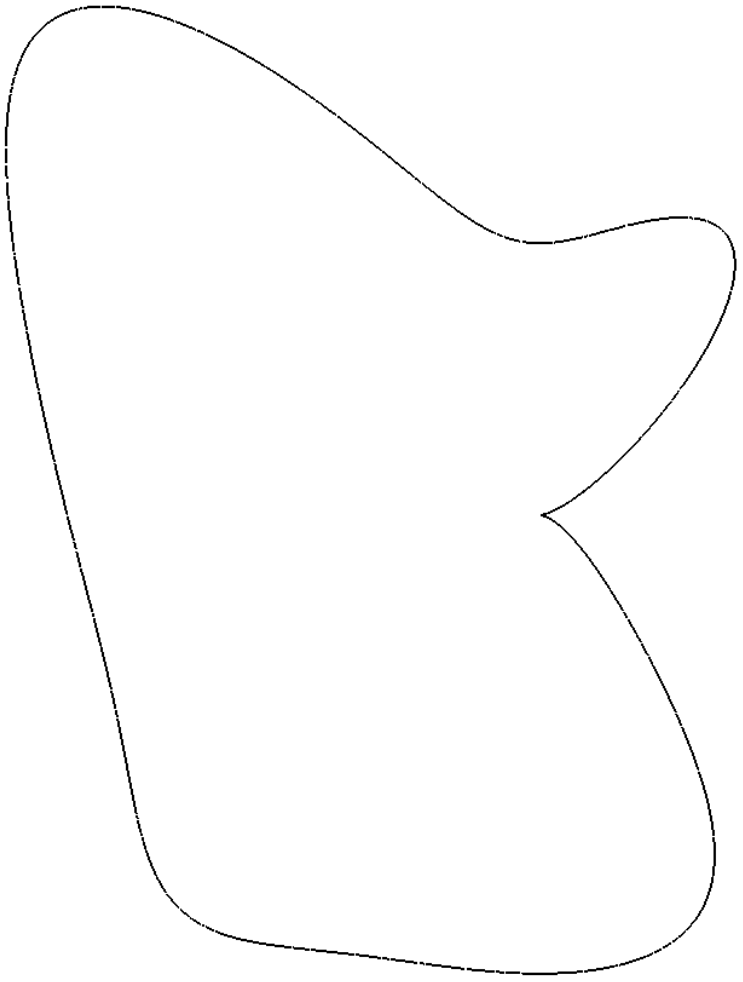
Adam Carrez

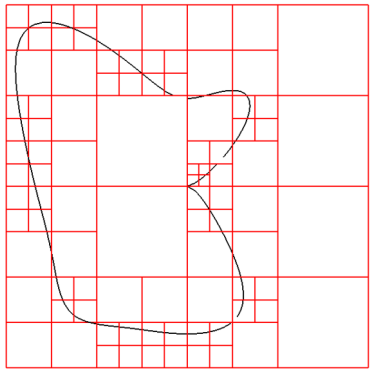
Complexités de représentations de fonctions mathématiques en temps réel

Introduction - Objectif

$$P(x,y) = 2x^6 + y^6 + 2y^2 + y^3 + 4x^3 - 3x^2y - 3y^2 = 0$$

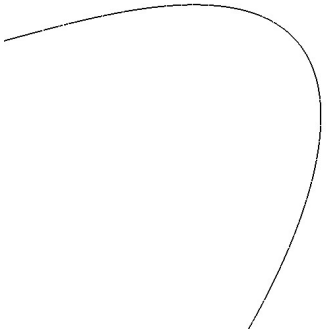


Introduction - Présentation du problème



Représentation en 2D

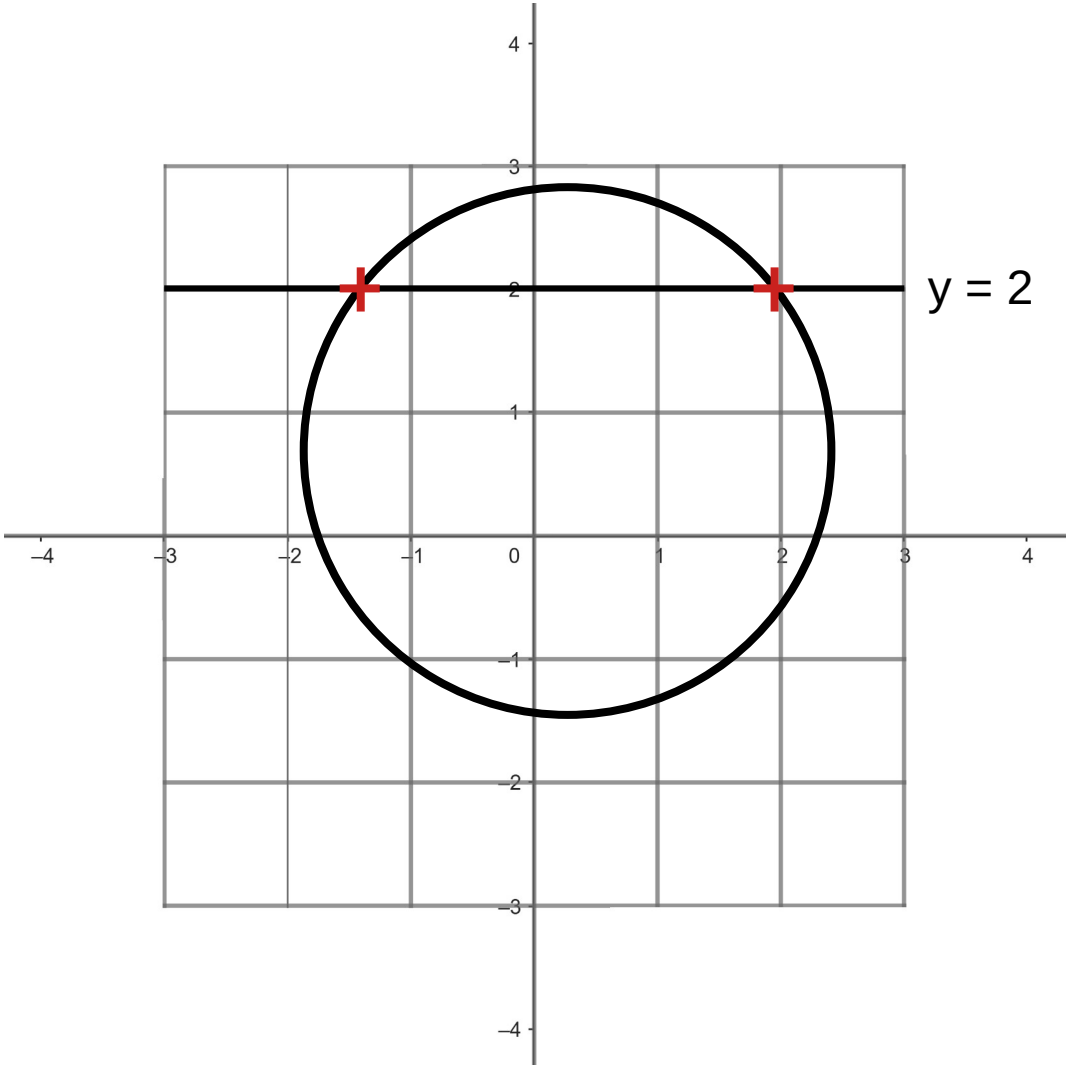
- Créer les points
- Comment les stocker
- Comment les relier



Améliorations dynamique du maillage

- Augmenter la qualité lors du zoom
- Pondérer la densité de points par la courbure

Représentation 2D – Construction de points



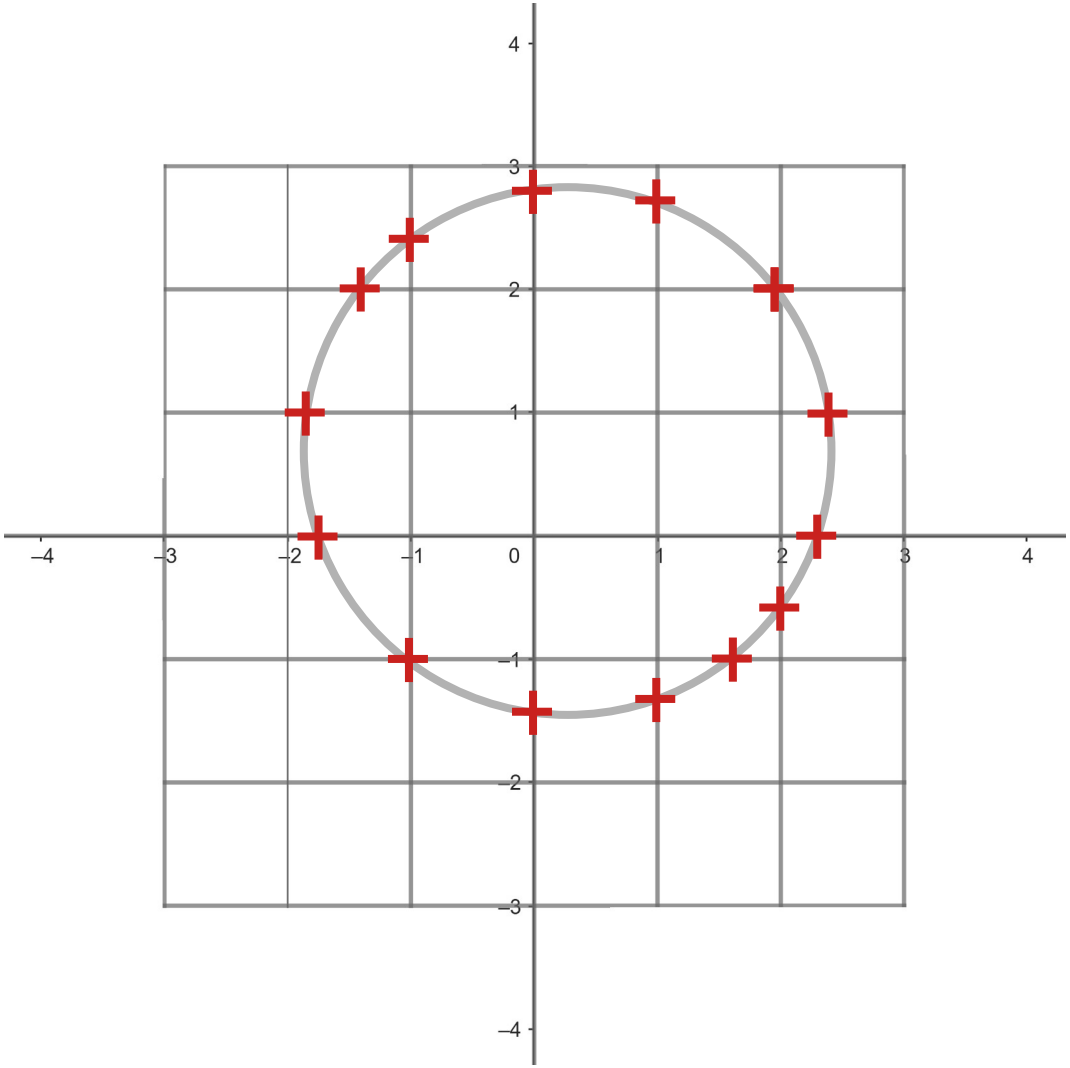
Méthode de Newton

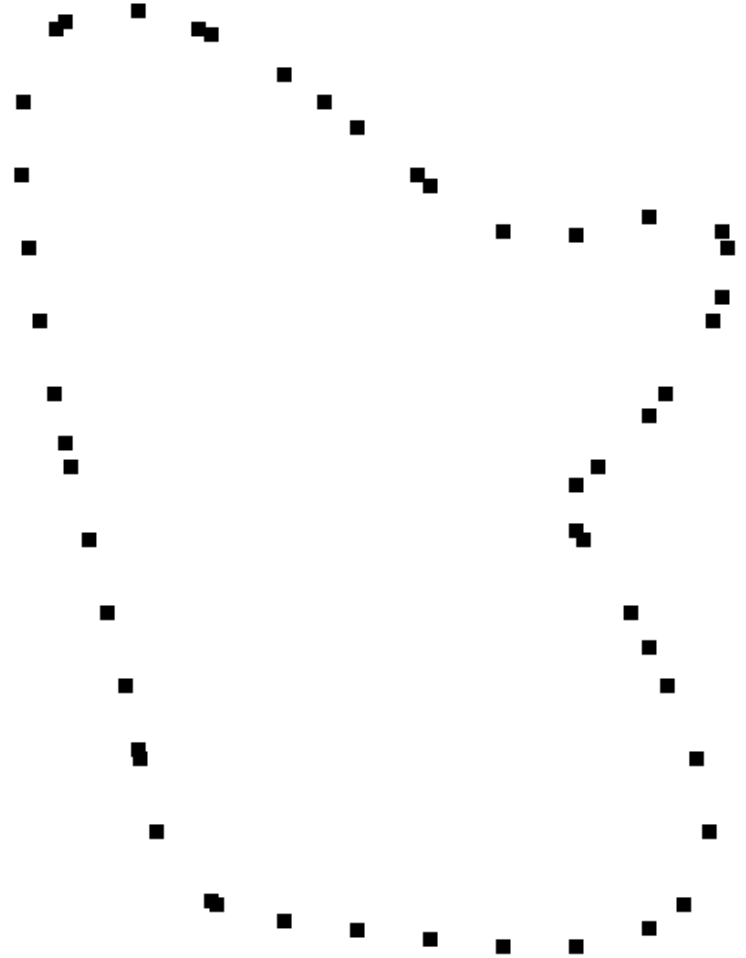
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Dans notre cas :

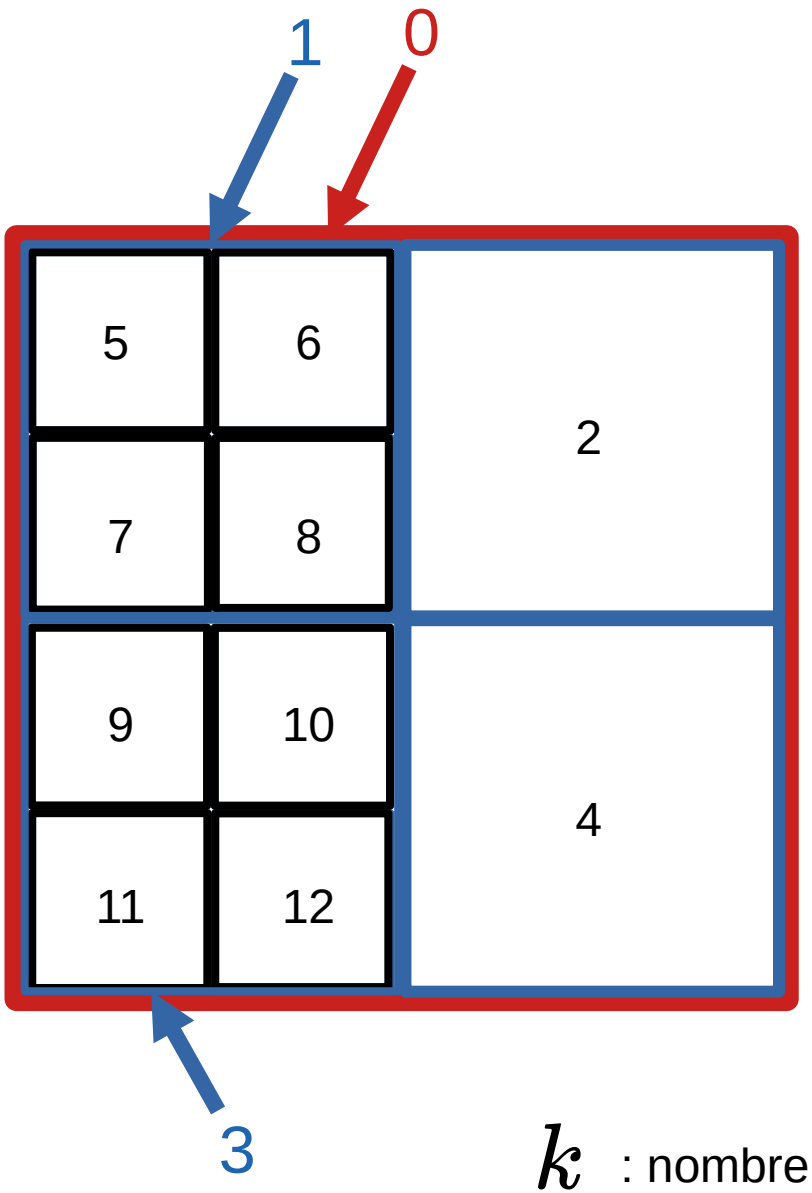
$$x_{n+1} = x_n - \frac{P(x_n, y)}{\frac{\partial P(x_n, y)}{\partial x}}$$

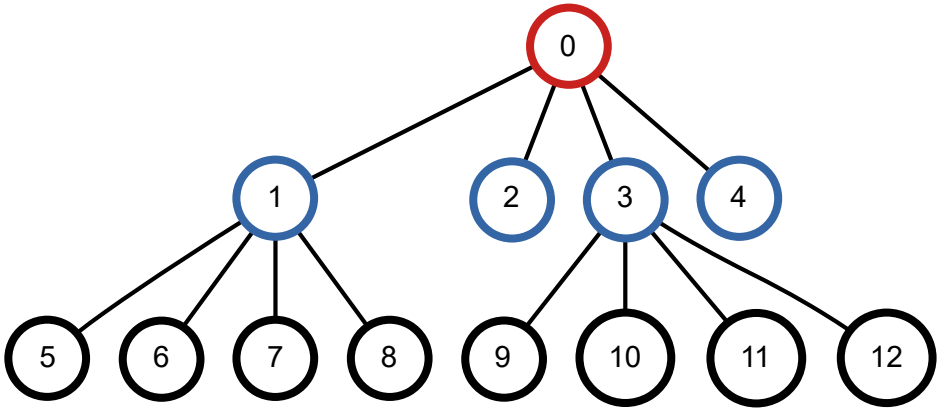
Représentation 2D – Construction de points





Représentation 2D - Quadtree

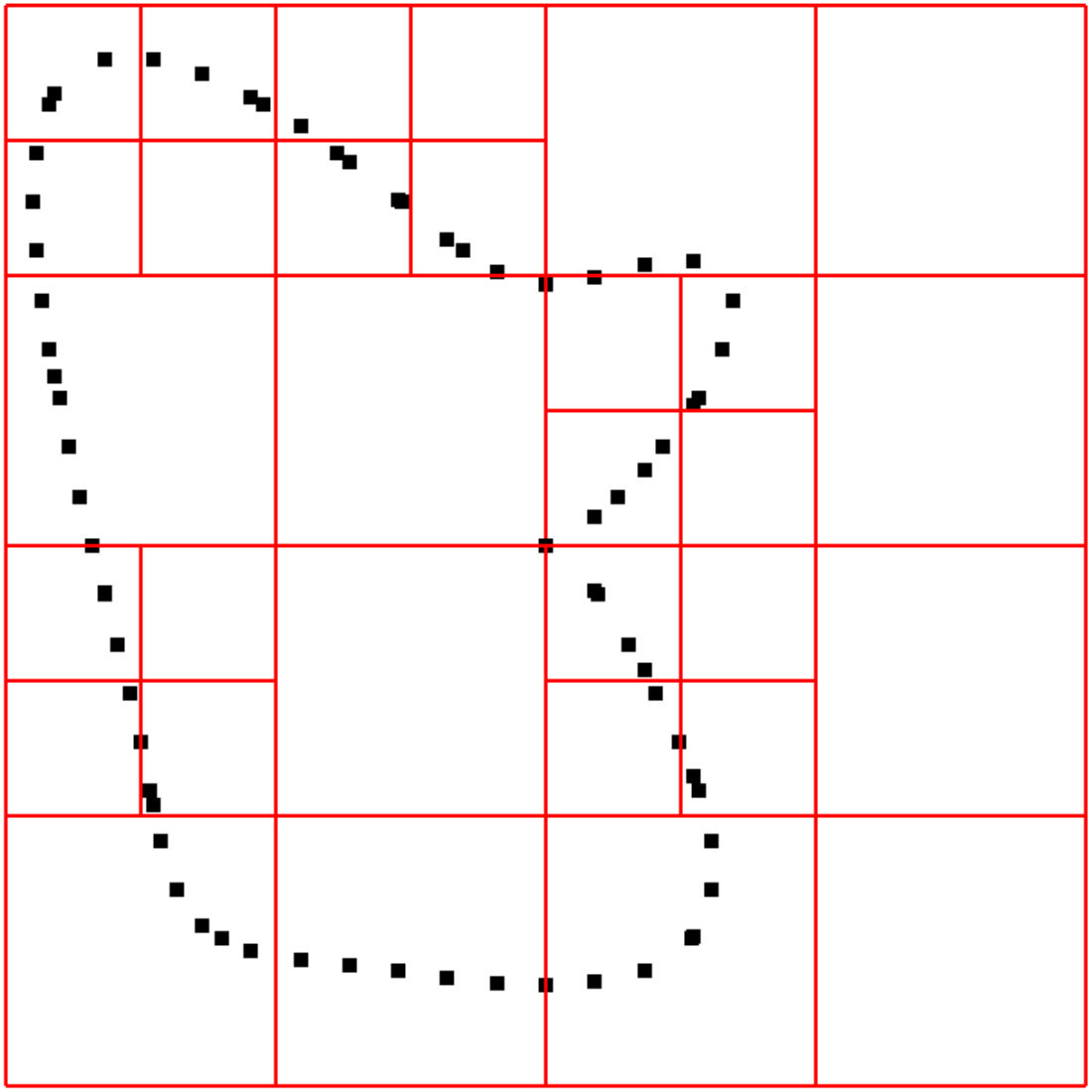




Structure arborescente du quadtree : réalisé avec Graphviz

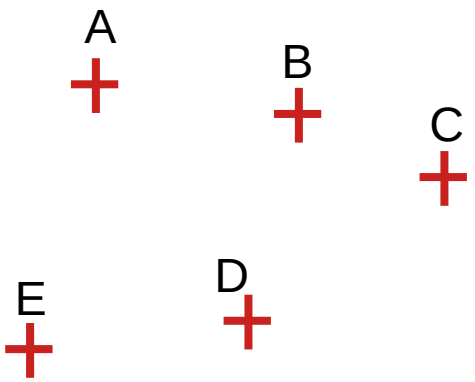
k : nombre maximum de points par cellule

Représentation 2D - Quadtree



Représentation 2D - Principe

Algorithme de Kruskal : $[A:0,\ B:1,\ C:2,\ D:3,\ E:4]$

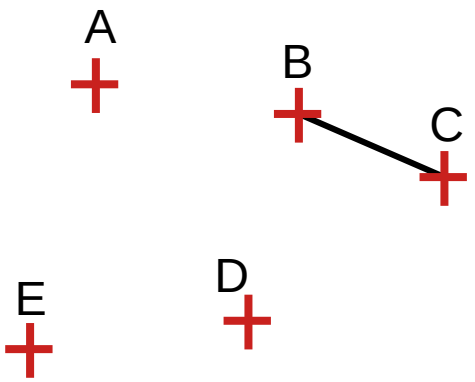


Représentation 2D - Principe

Algorithme de Kruskal :

$$[A:0,\ \ B:1,\ \ C:2,\ \ D:3,\ \ E:4]$$

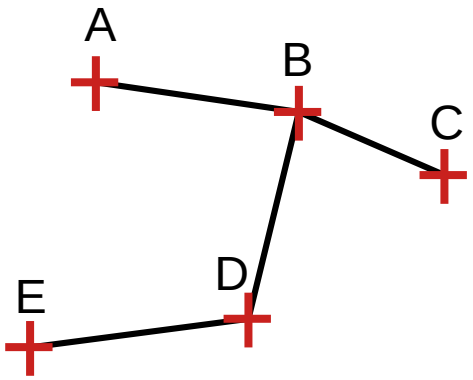
$$[A:0,\ \ B:1,\ \ C:1,\ \ D:3,\ \ E:4]$$



Représentation 2D - Principe

Algorithme de Kruskal :

$$[A:0,\ \ B:1,\ \ C:2,\ \ D:3,\ \ E:4]$$



$$[A:0,\ \ B:1,\ \ C:1,\ \ D:3,\ \ E:4]$$

$$\vdots$$

$$\vdots$$

$$[A:1,\ \ B:1,\ \ C:1,\ \ D:1,\ \ E:1]$$

Représentation 2D - Principe

Algorithme de Kruskal :

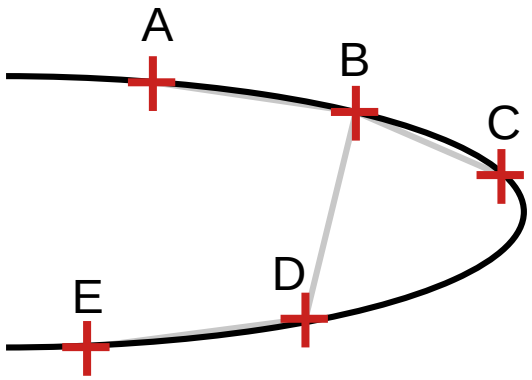
$$[A:0,\ B:1,\ C:2,\ D:3,\ E:4]$$

$$[A:0,\ B:1,\ C:1,\ D:3,\ E:4]$$

$$\vdots$$

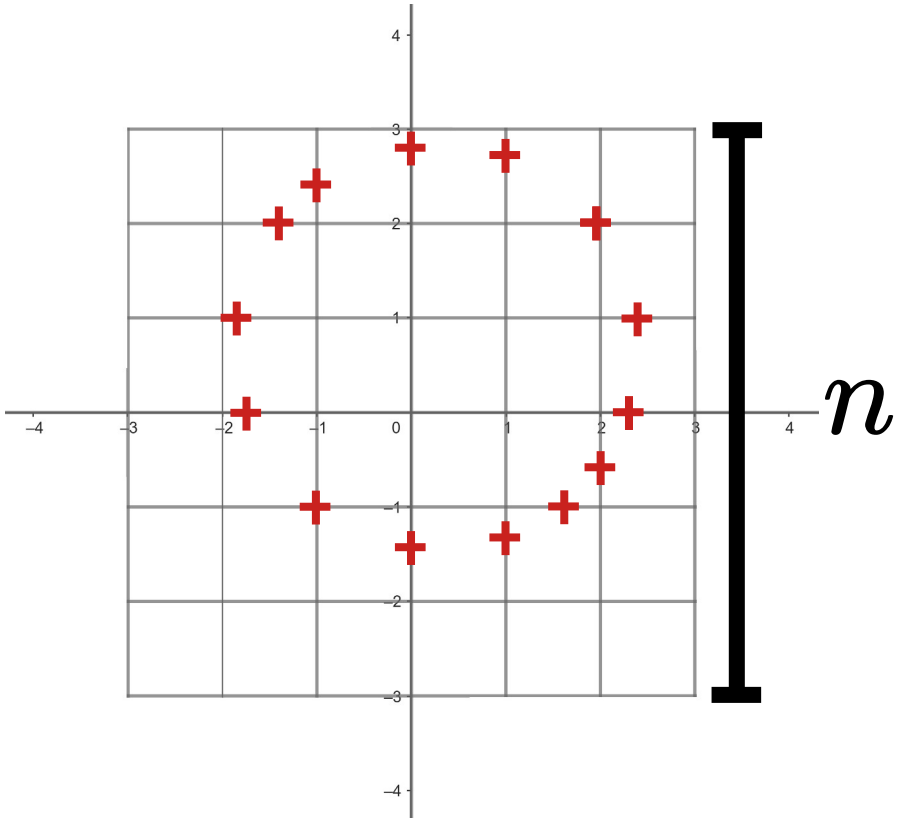
$$\vdots$$

$$[A:1,\ B:1,\ C:1,\ D:1,\ E:1]$$



lier_points n’est pas correcte

Représentation 2D – Complexités Théoriques



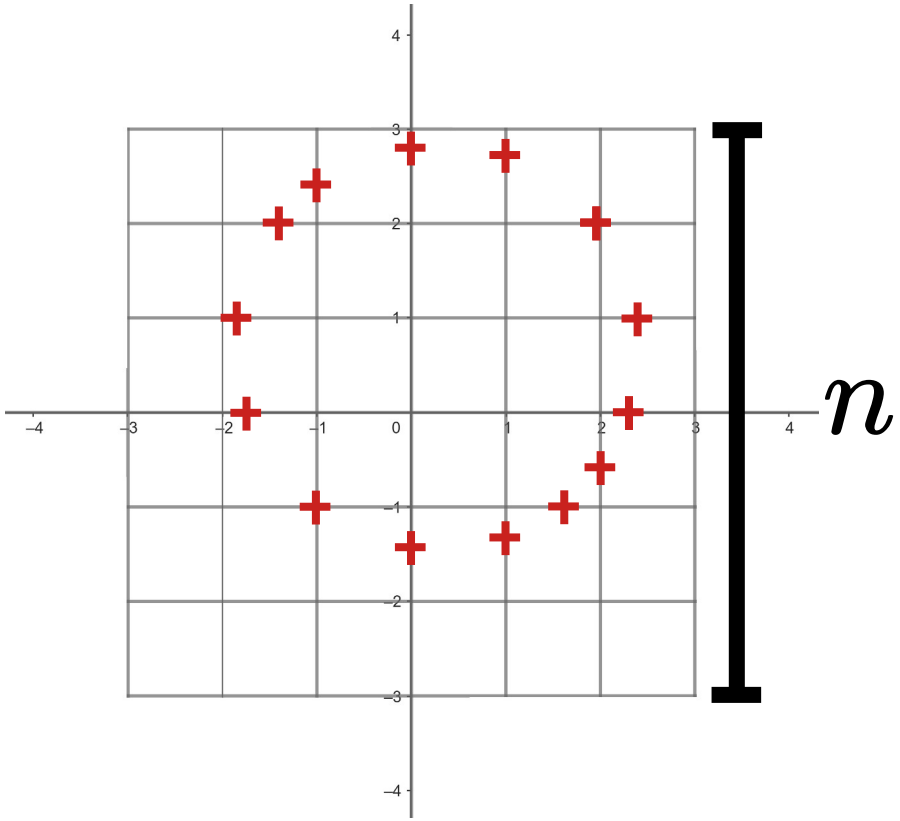
m : nombre d'itérations de la méthode de Newton

k : nombre de points par quadtree

Générer les points via *trouver_racine :*

$$O(n^2 \times m)$$

Représentation 2D – Complexités Théoriques



m : nombre d'itérations de la méthode de Newton

k : nombre de points par quadtree

Générer les points via *trouver_racine :*

$$O(n^2 \times m)$$

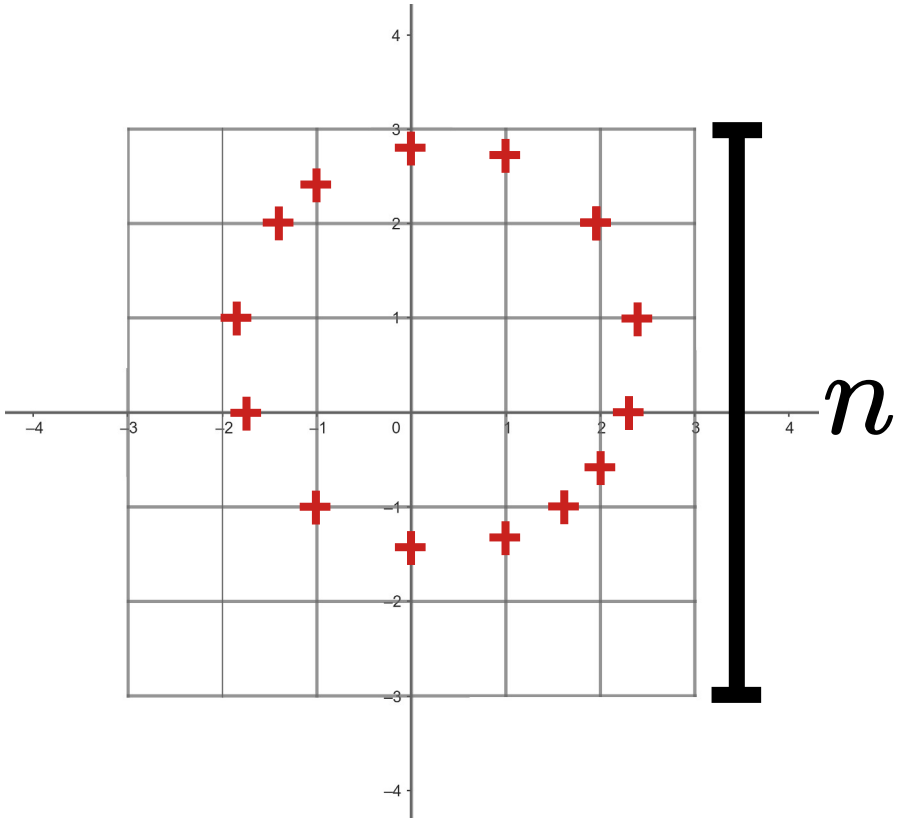
Nettoyer les doublons

$$O\left(n^4\right)$$

En pratique :

$$O\left(n^3\right)$$

Représentation 2D – Complexités Théoriques



m : nombre d'itérations de la méthode de Newton

k : nombre de points par quadtree

Générer les points via *trouver_racine :*

$$O(n^2 \times m)$$

Nettoyer les doublons

$$O\left(n^4\right)$$

En pratique :

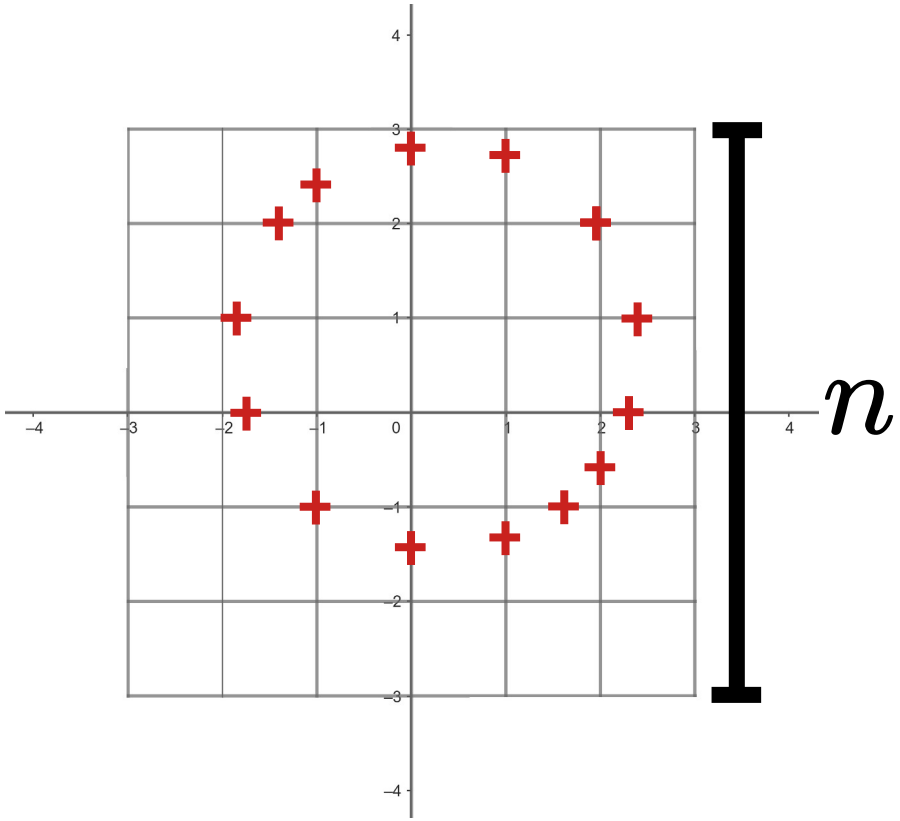
$$O\left(n^3\right)$$

Diviser l'espace en quadtrees

$$O\left(n \times \log\left(\frac{n}{k}\right)\right)$$

$$c(n) = 4 \times c\left(\frac{n}{4}\right) + n$$

Représentation 2D – Complexités Théoriques



m : nombre d'itérations de la méthode de Newton

k : nombre de points par quadtree

Générer les points via *trouver_racine :*

$$O(n^2 \times m)$$

Nettoyer les doublons

$$O\left(n^4\right)$$

En pratique :

$$O\left(n^3\right)$$

Diviser l'espace en quadtrees

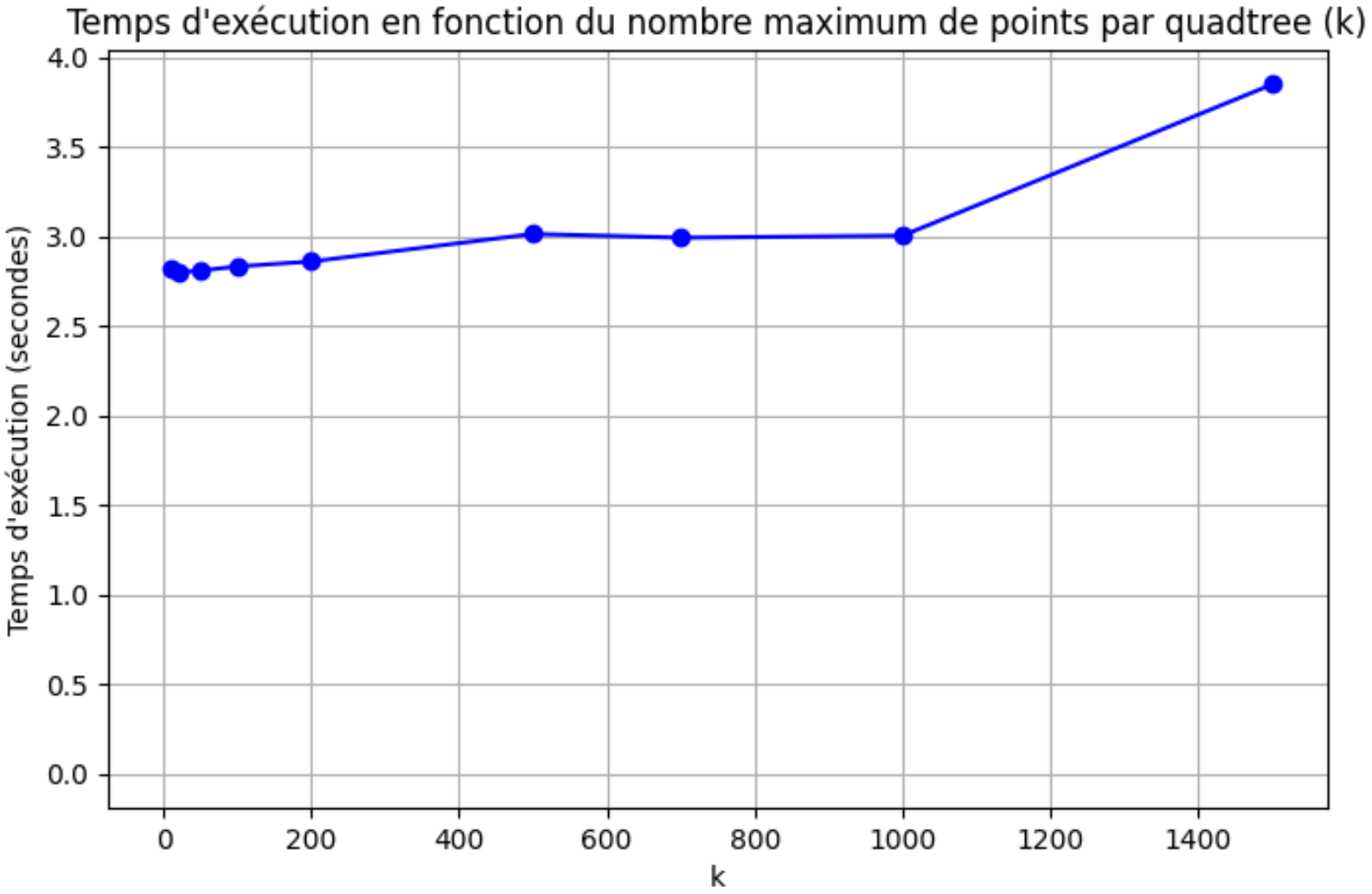
$$O\left(n \times \log\left(\frac{n}{k}\right)\right)$$

$$c(n) = 4 \times c\left(\frac{n}{4}\right) + n$$

Lier les points

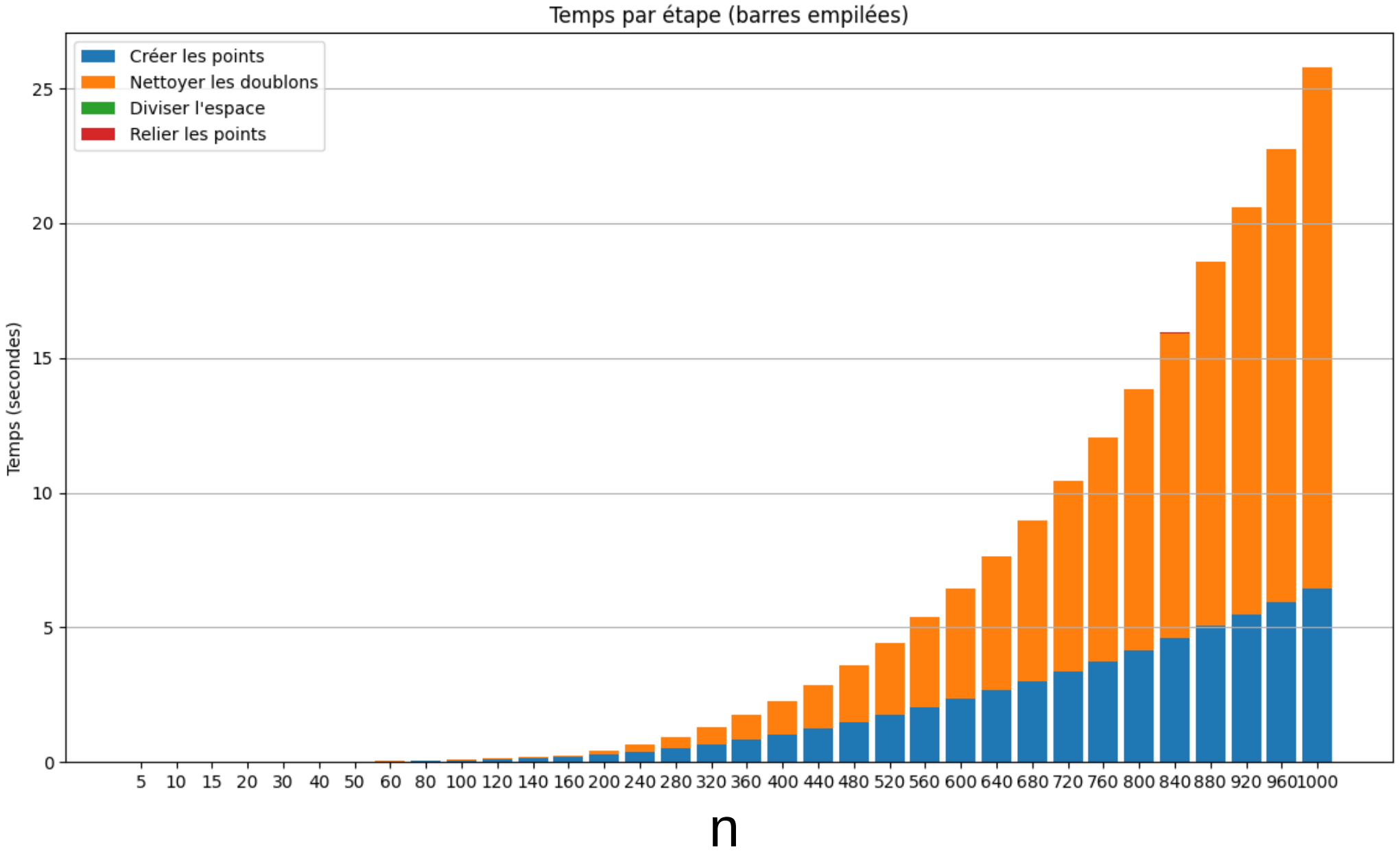
$$O\left(n \times k^2 \log k\right)$$

Représentation 2D – Analyses réelles

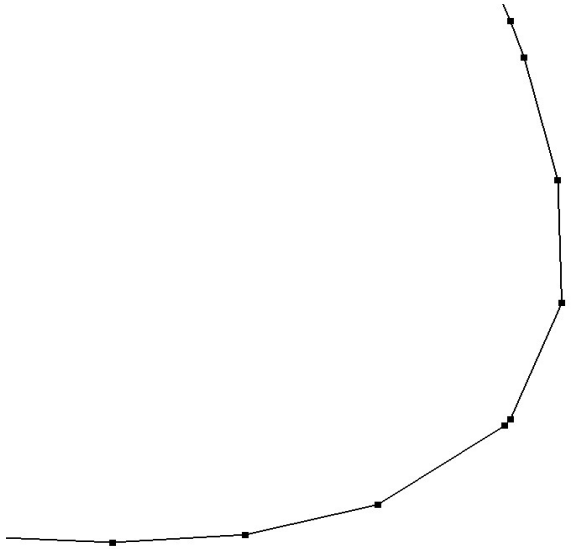


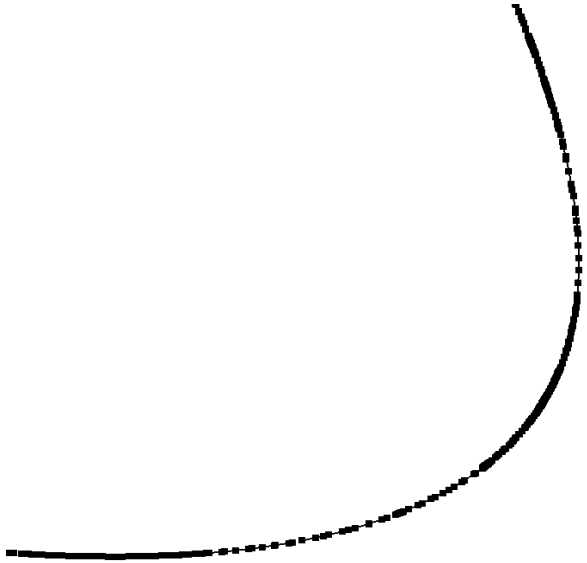
Générer la forme : $O\left(n^4 + n^2 m\right)$

Représentation 2D – Analyses réelles



Maillage dynamique – Principe





Plusieurs méthodes explorées

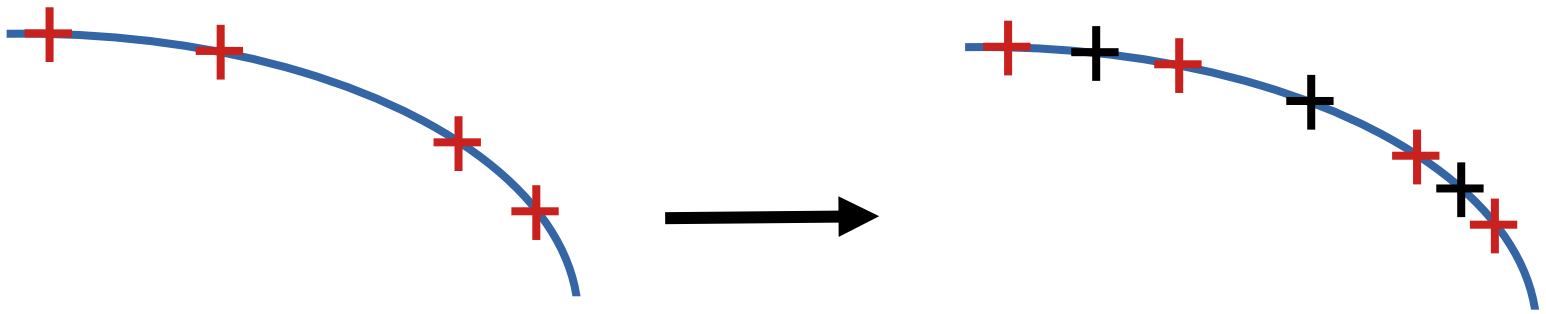
1- Par ajout de points

2- Par recréation des cellules visibles

2.2- Sans optimisation par la courbure

2.1- Avec optimisation

Maillage dynamique – Ajout de point



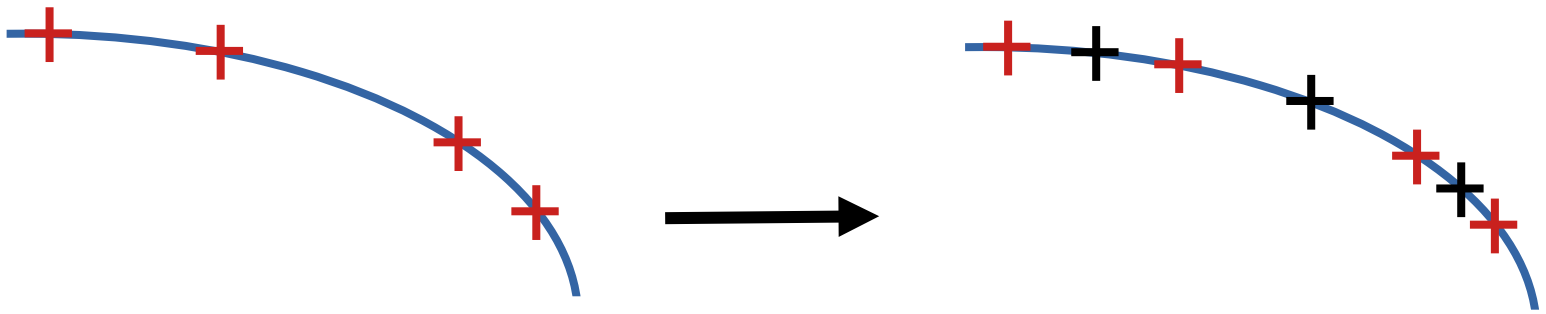
m : nombre d'itérations de Newton

N : nombre de points visibles à l'écran

Complexité temporelle : $O(Nm)$

Complexité spatiale : $O(N)$

Maillage dynamique – Ajout de point



m : nombre d'itérations de Newton

N : nombre de points visibles à l'écran

Complexité temporelle : $O(Nm)$

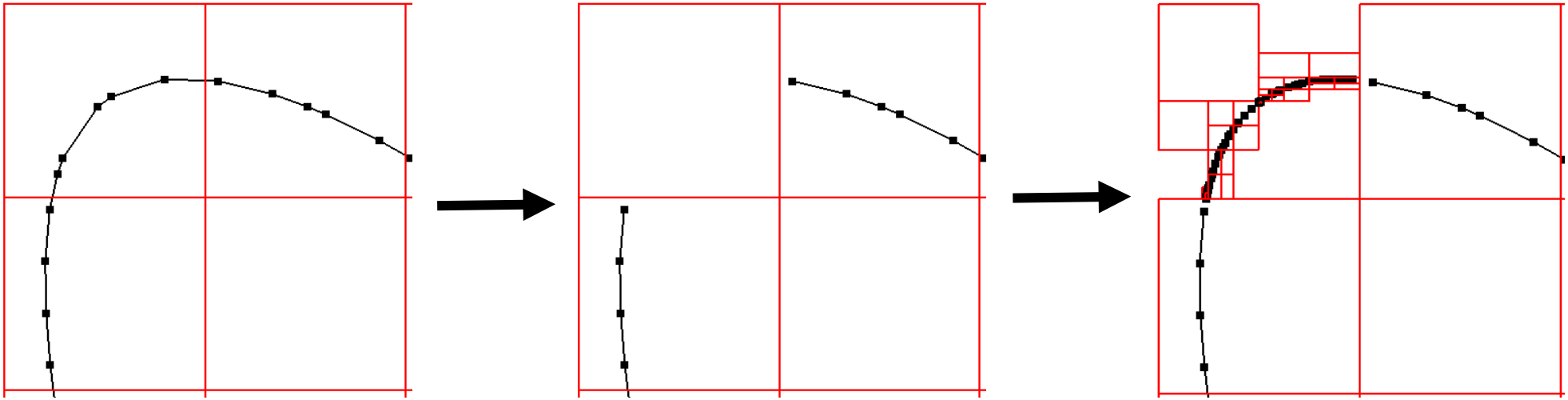
Complexité spatiale : $O(N)$

Limitations :

- trouver_racine_newton échoue lors d'un zoom fort

- Densifie de façon irrégulière

Maillage dynamique – Recréation du quadtree



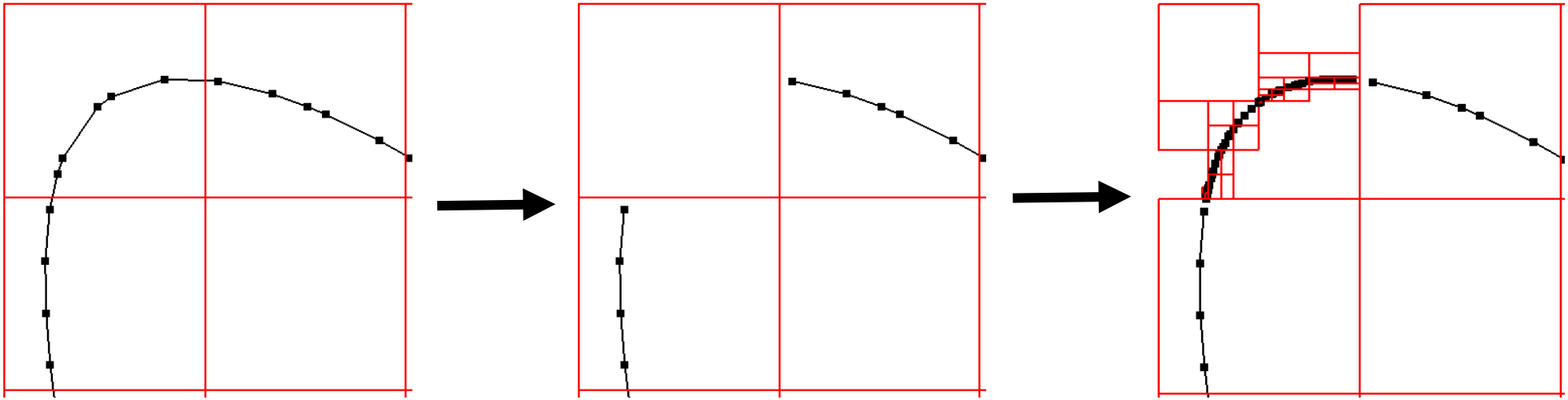
Rappel

k : nombre de points maximum par quadtree

Complexité temporelle : $O\left(k^4\right)$

Complexité spatiale : $O\left(k^2\right)$

Maillage dynamique – Recréation du quadtree



Rappel

k : nombre de points maximum par quadtree

Complexité temporelle : $O\left(k^4\right)$

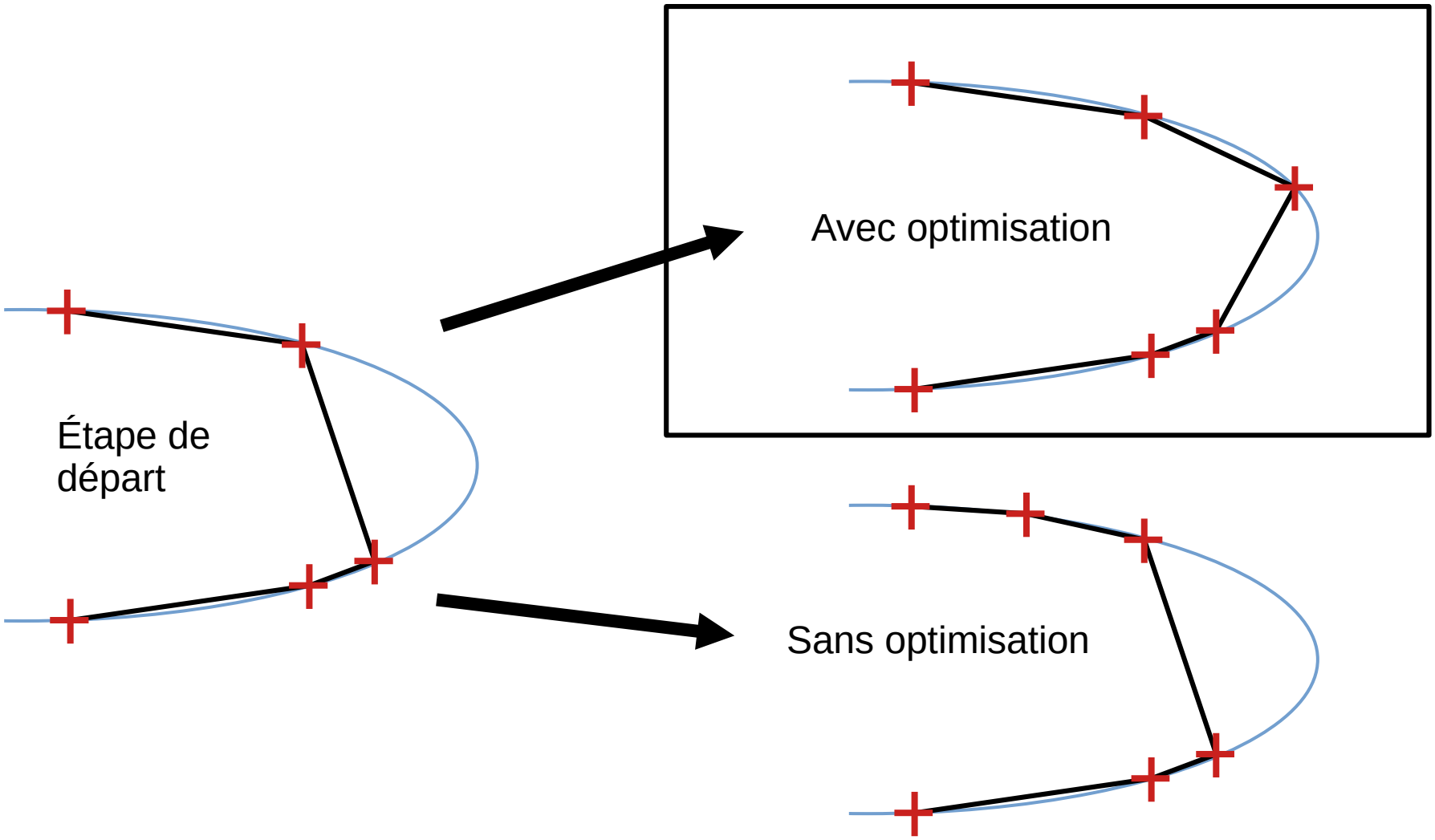
Complexité spatiale : $O\left(k^2\right)$

Limitations résolues :

- Moins d'échecs

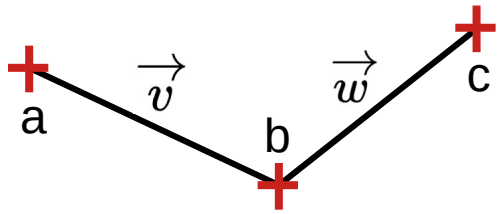
- Ne dépend plus de la densité initiale

Optimisation par la courbure - Principe

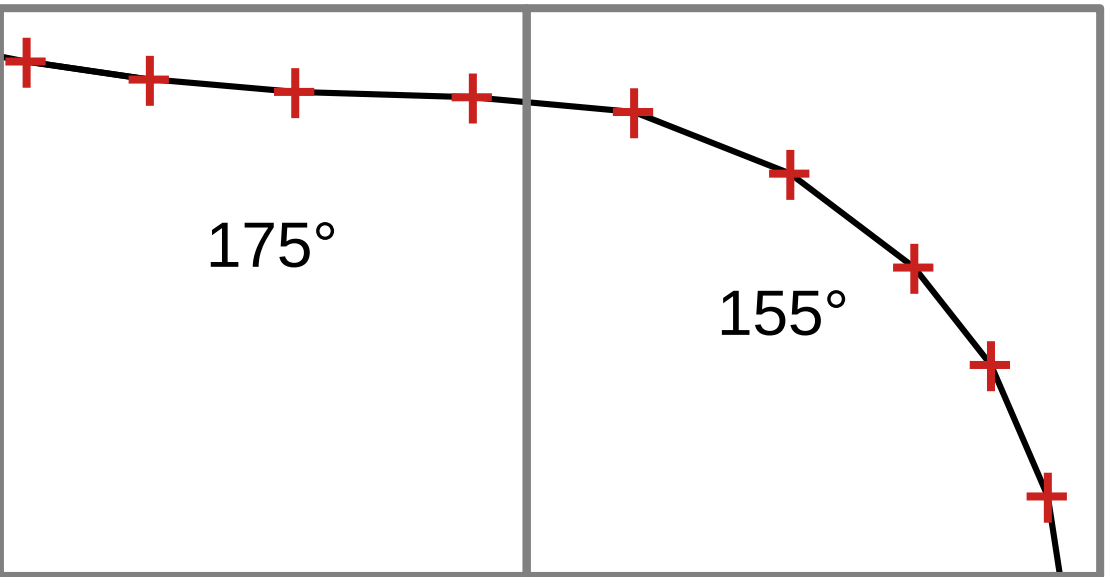


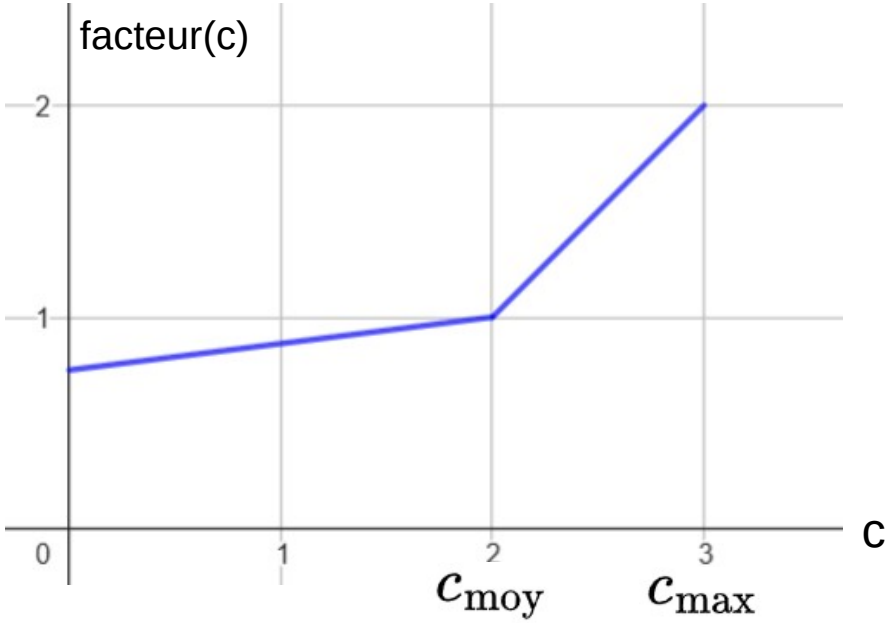
Optimisation par la courbure - Principe

Courbure moyenne
d'une cellule



$$\underset{(a,\,b,\,c)\,\in\,Q}{\text{moy}}\left(\widehat{abc}\right) = \boxed{\underset{(a,\,b,\,c)\,\in\,Q}{\text{moy}}\left(\arctan\frac{|v_x w_y - v_y w_x|}{v_x w_x + v_y w_y}\right)}$$





Optimisation par la courbure - Complexité

k : nombre de points maximum par quadtree

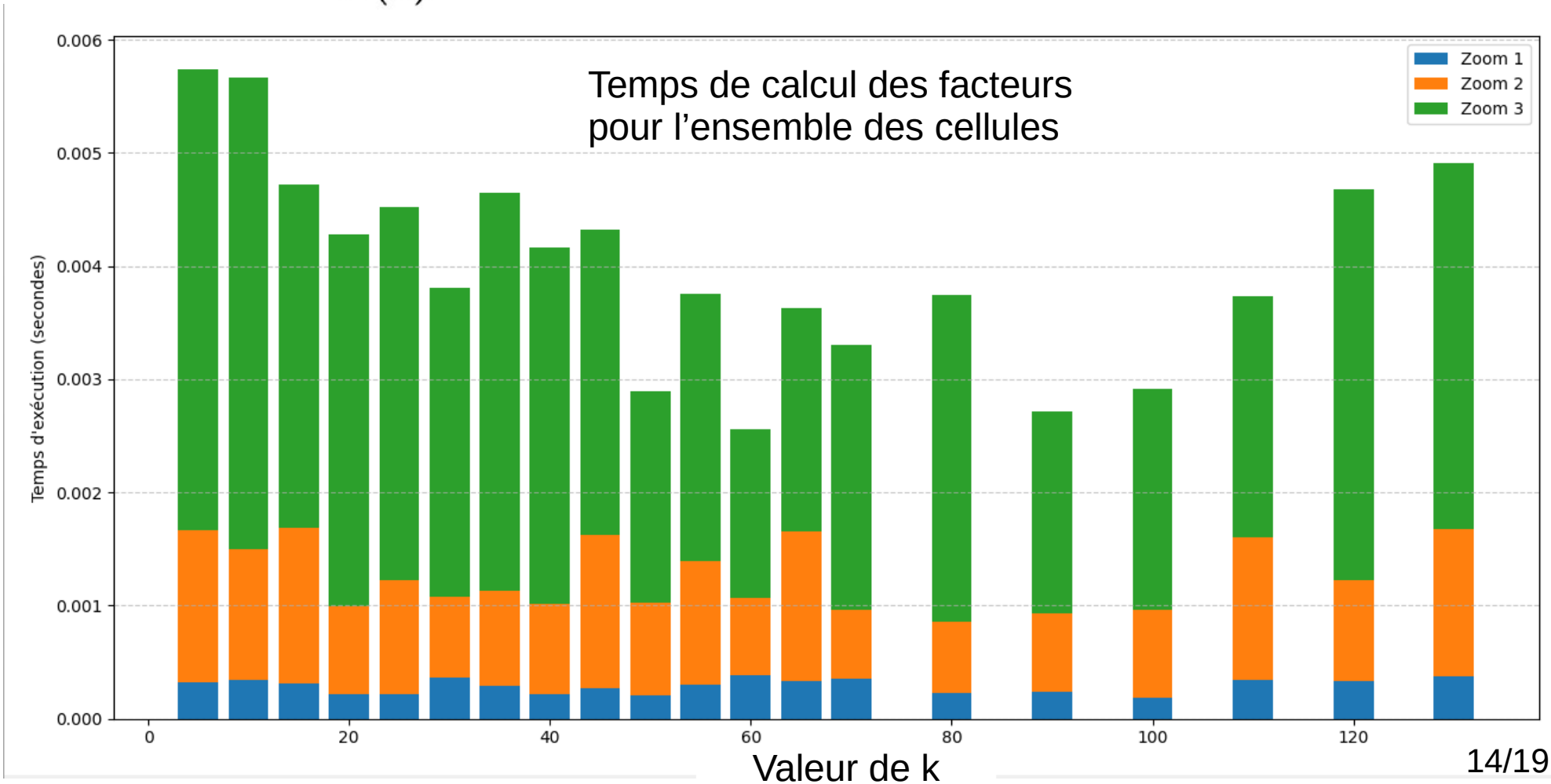
Courbure moyenne d'un Quadtree

Temporelle $O(k)$

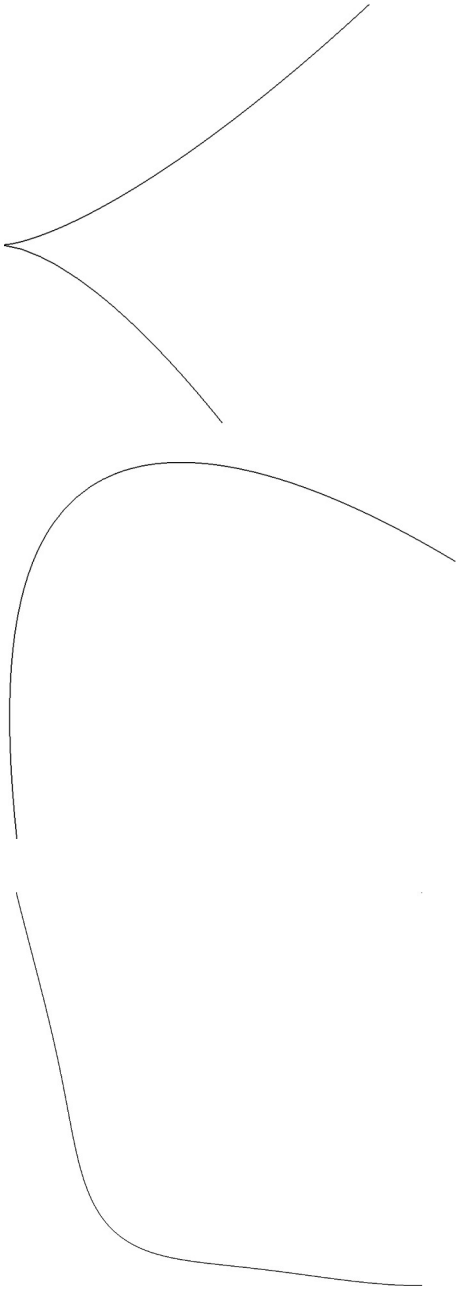
Spatiale $O(1)$

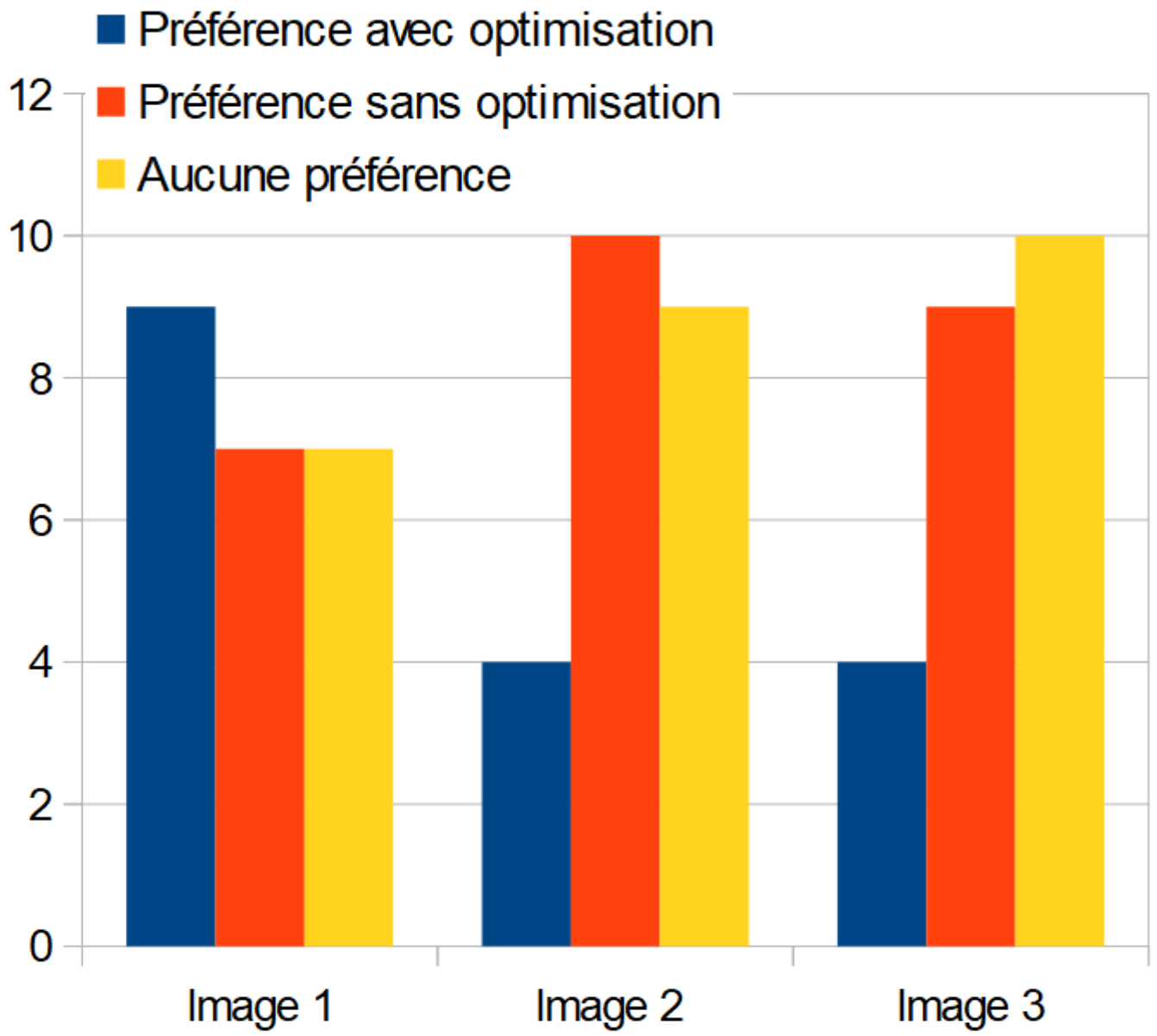
Pour N points visibles

Temporelle $O(N)$

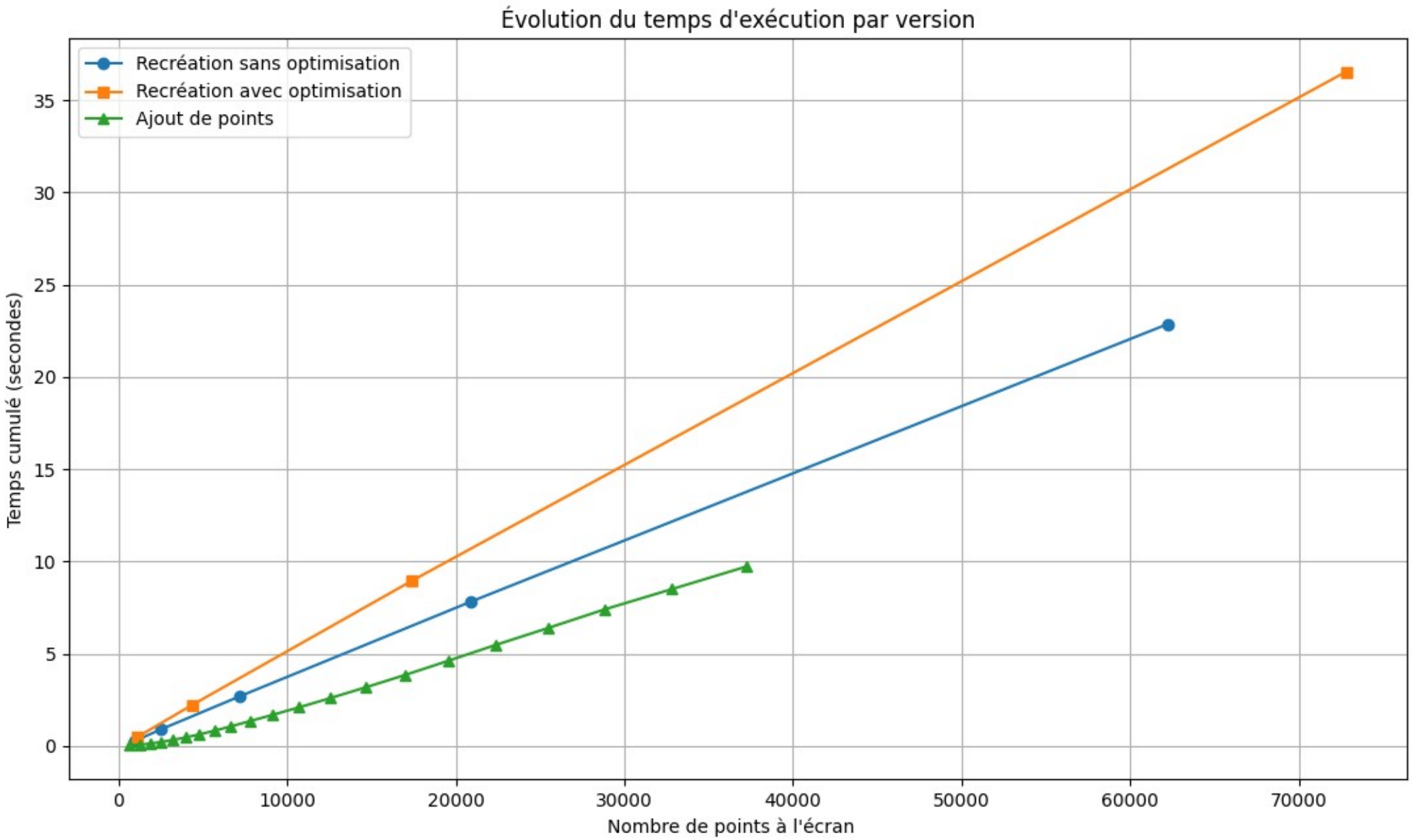


Optimisation par la courbure - Intérêt

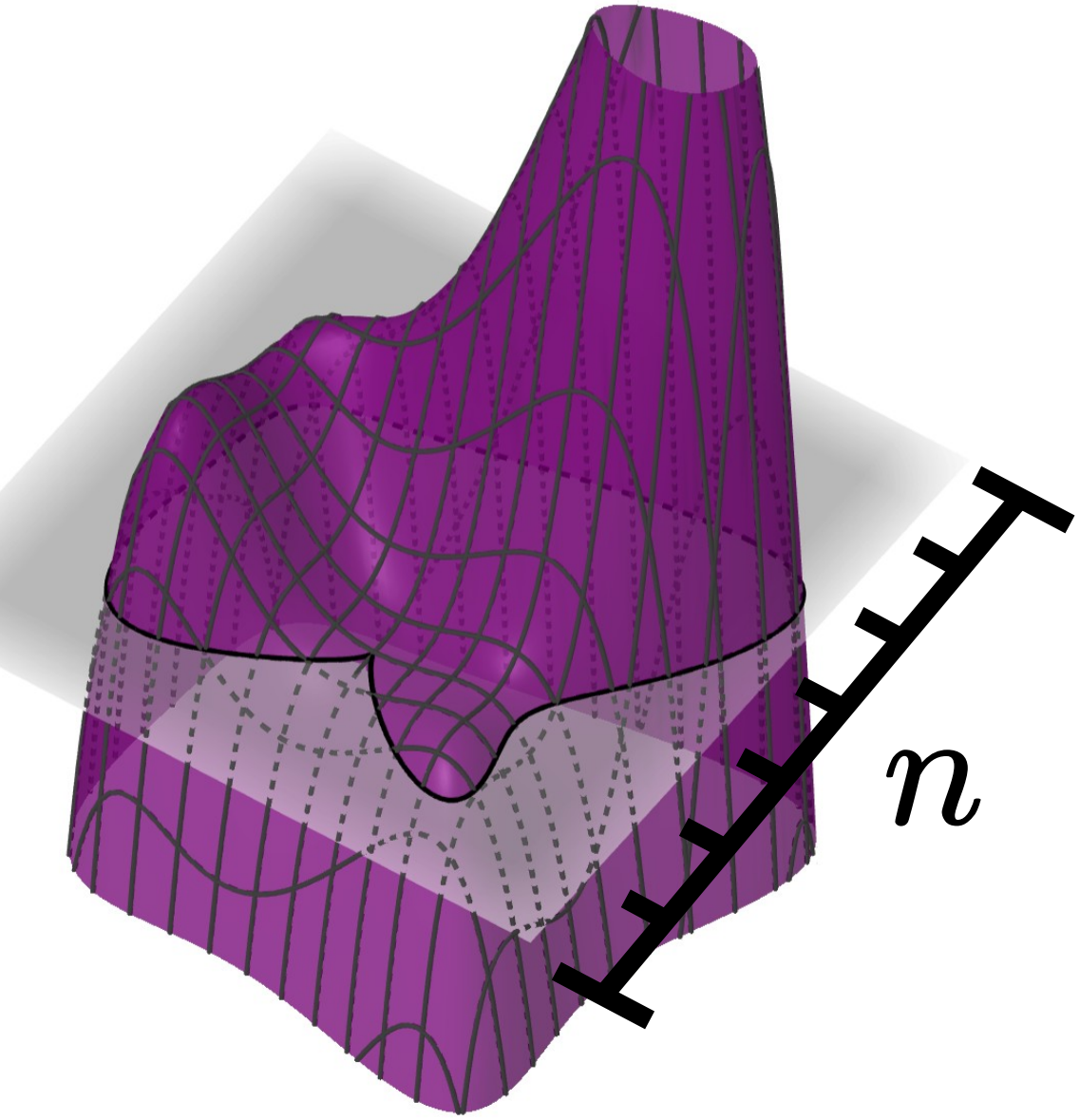




Comparaison des versions et optimisation



Ouverture – Analyse théorique 3D



Générée sur GeoGebra 3D

Calcul des $(x,\ y,\ P(x,y))$

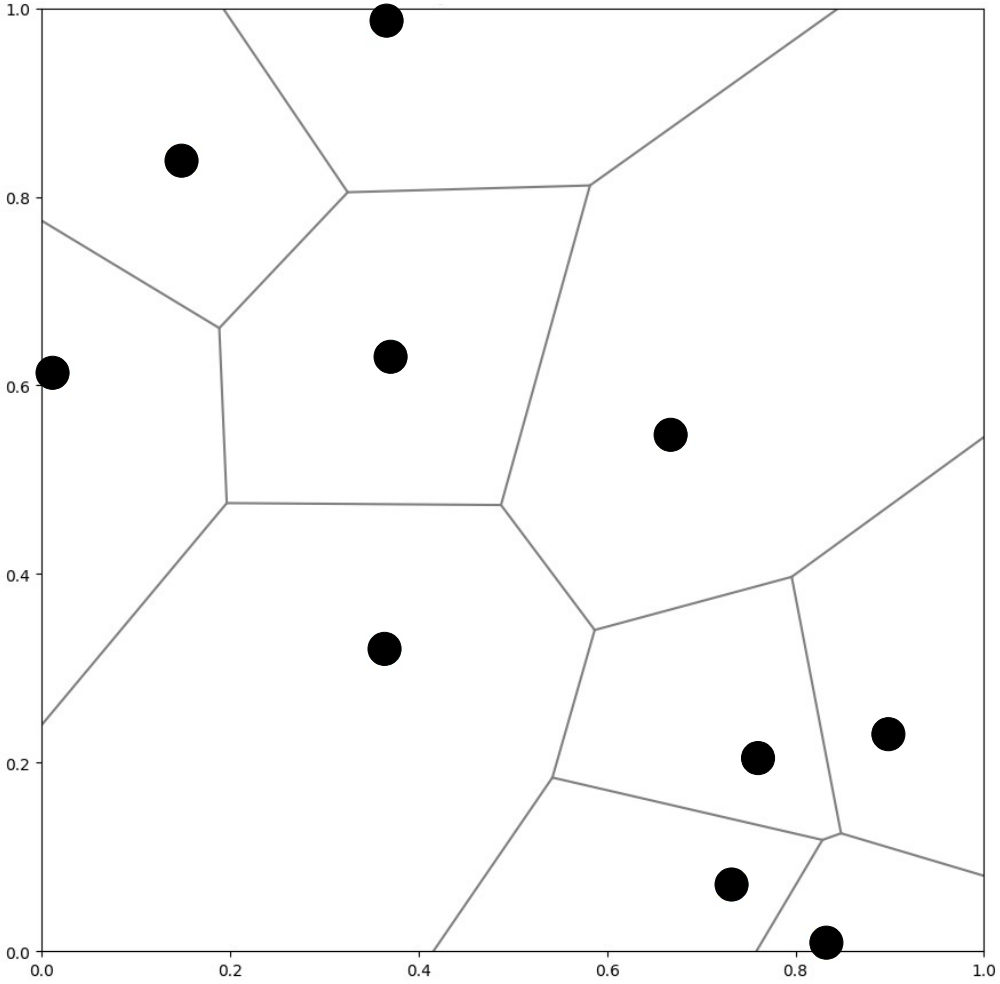
Complexité temporelle : $O(n^2)$

spatiale : $O(n^2)$

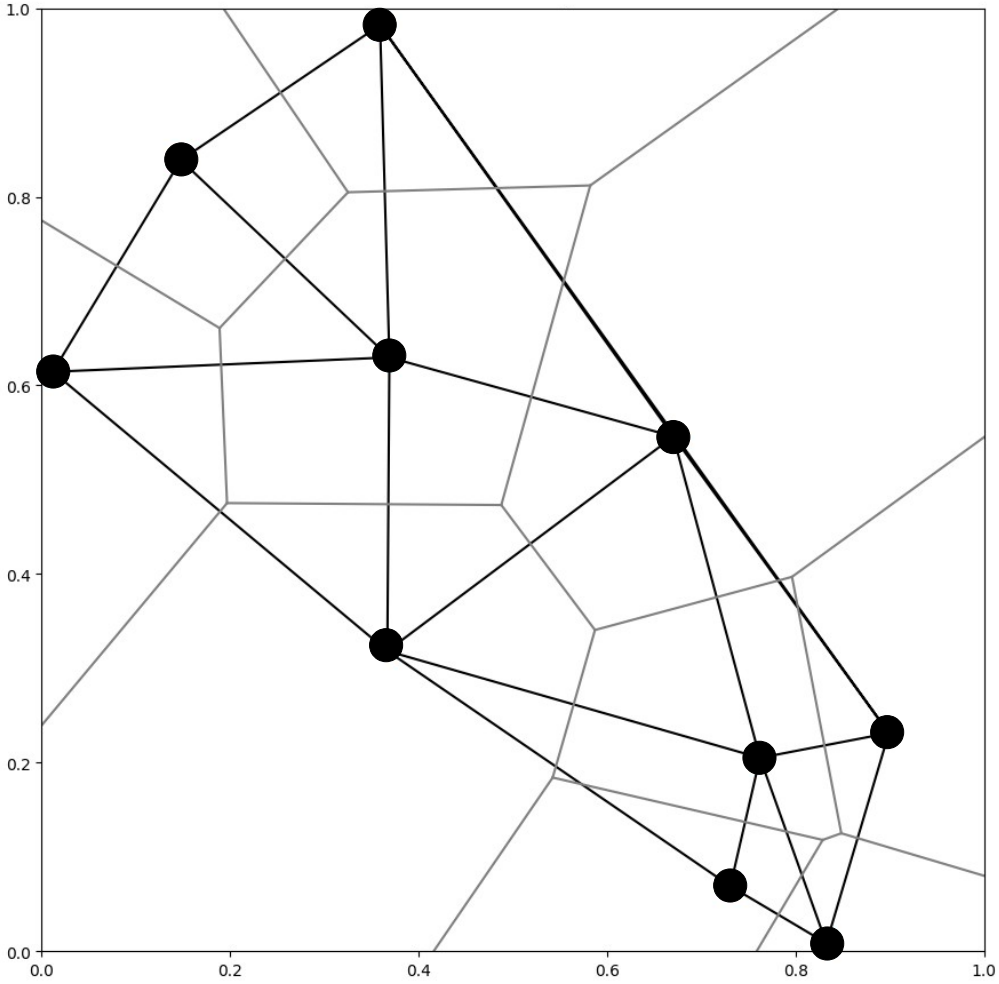
⚠ Il faut n plus grand que pour la 2D

Ouverture – Analyse théorique 3D

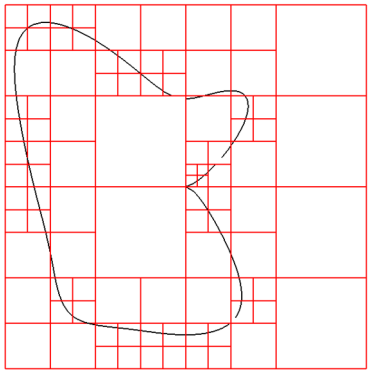
Diagramme de Voronoi



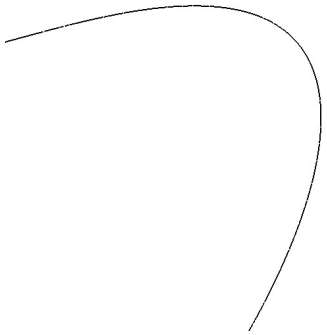
Triangulation de Delaunay



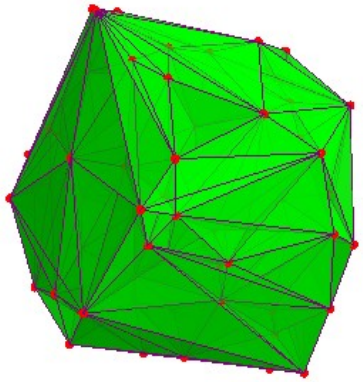
Questions



Représentation en 2D
- Points
- Quadtree
- Algorithme de Kruskal



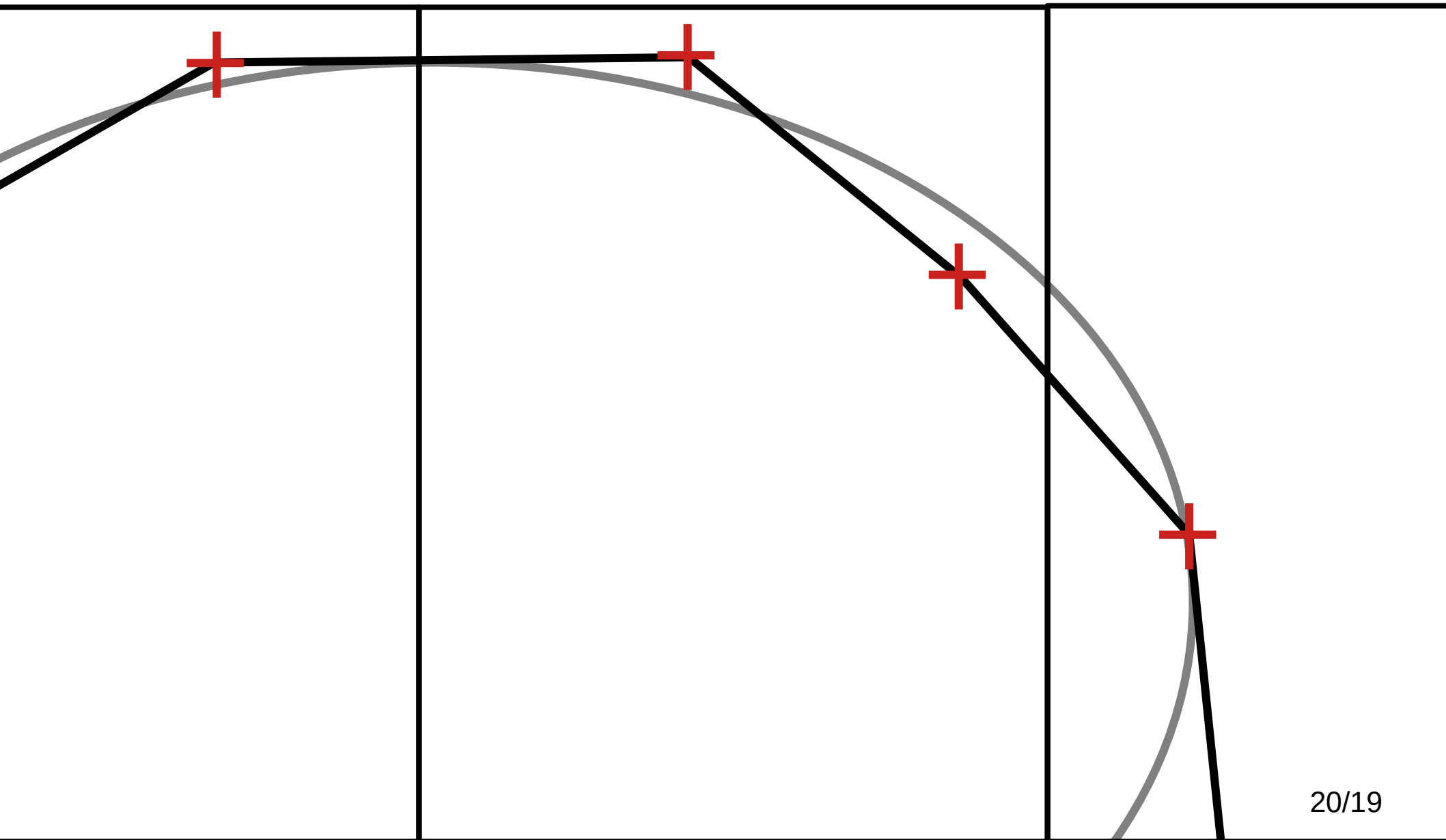
Améliorations dynamique du maillage
- Ajout de point
- Recréation des quadtrees
- Optimisation par la courbure



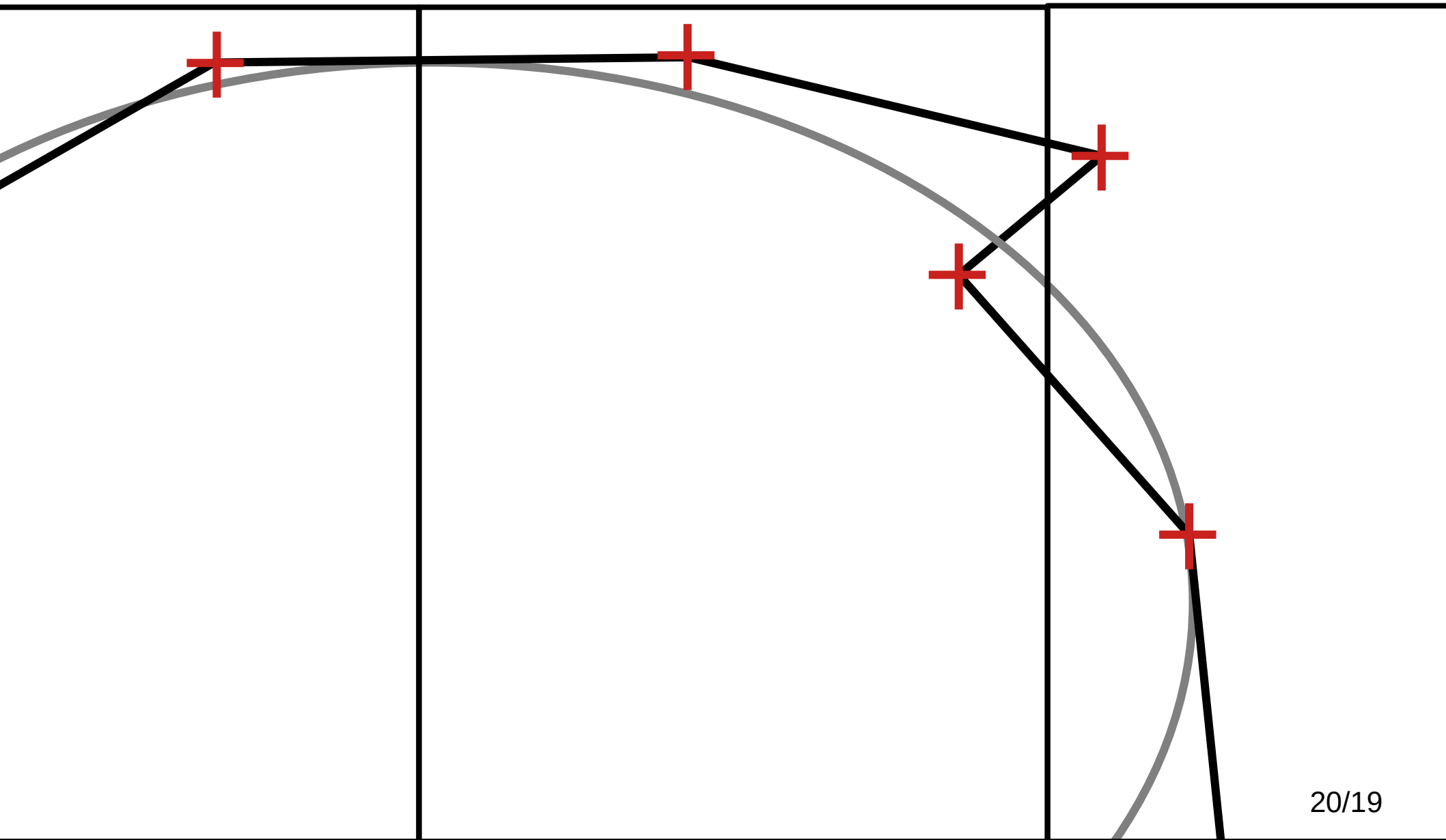
Créée via DelaunayMesh

Analyse théorique en 3D
- Algorithme de Crust

Annexe – Échec méthode de Newton



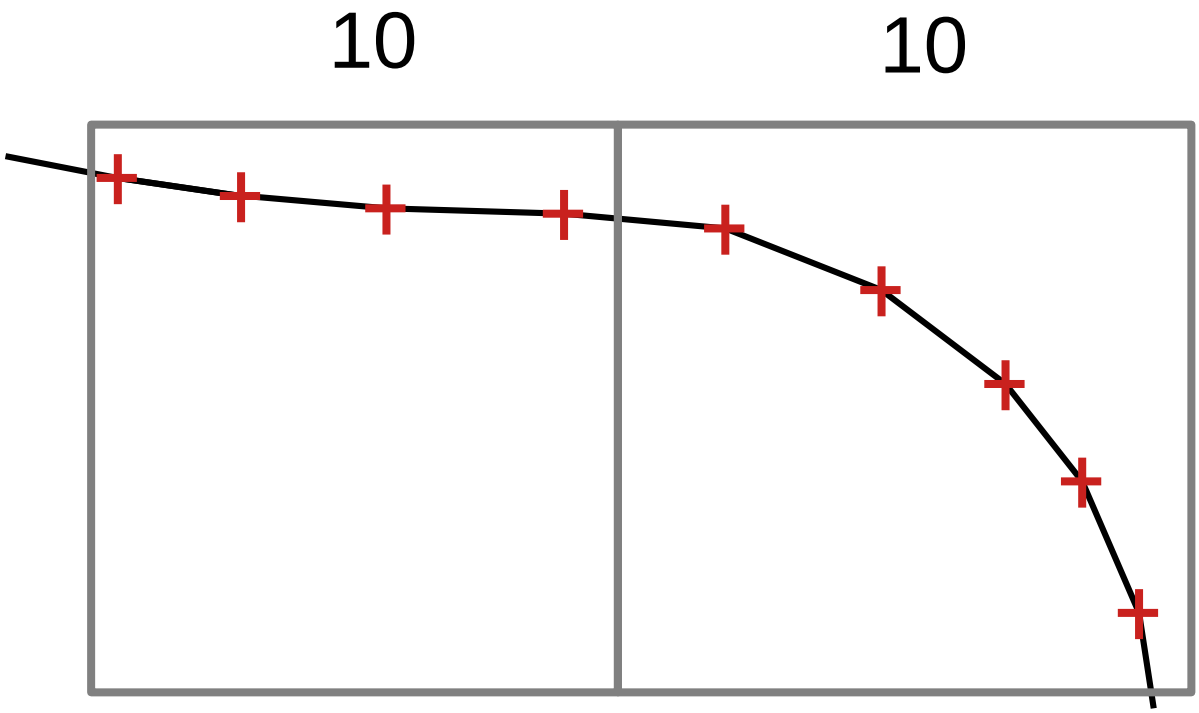
Annexe – Échec méthode de Newton



Annexe – Temps d’exécution sans optimisation

Complexité temporelle : $O\left(k^4\right)$

k = 10

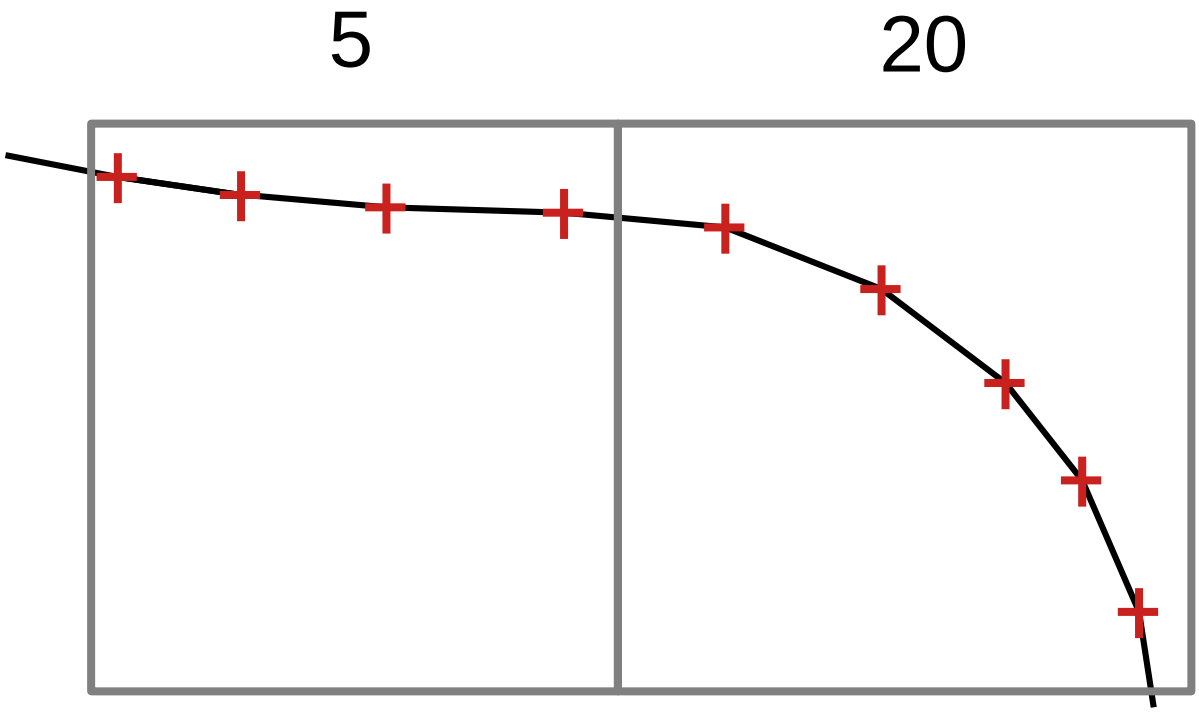


Nombre total d’opérations : 20.000

Annexe – Temps d’exécution avec optimisation

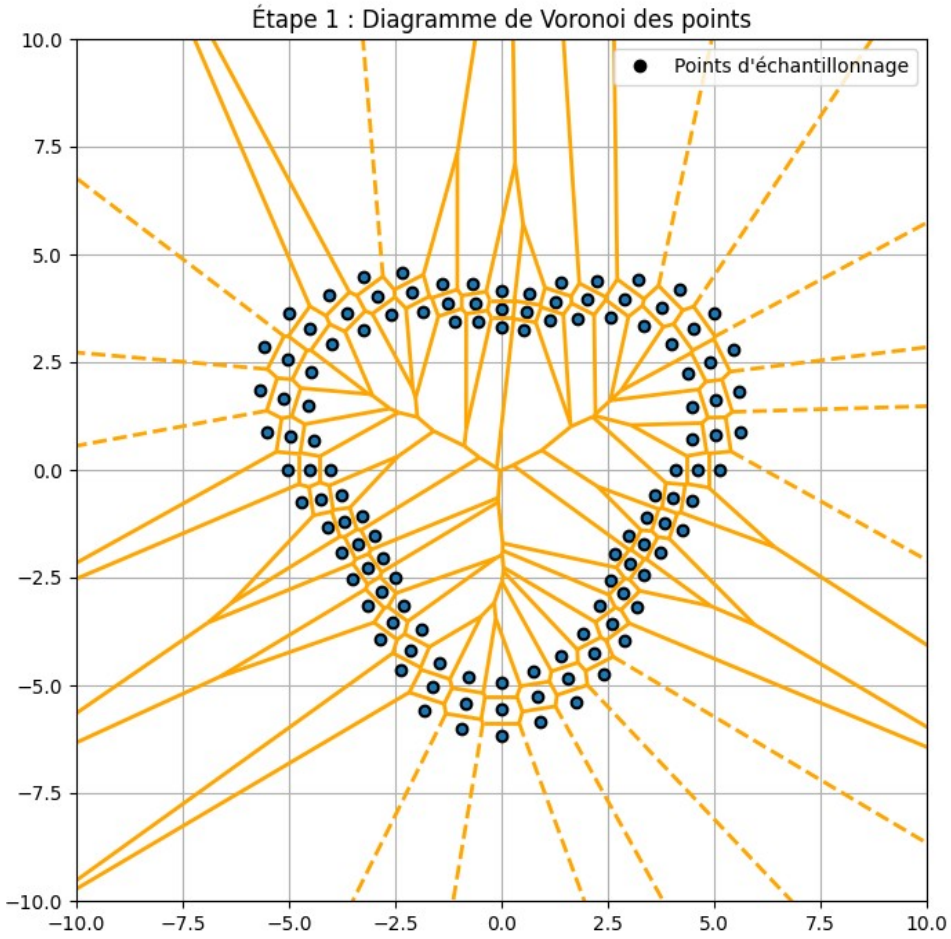
Complexité temporelle : $O\left(k^4\right)$

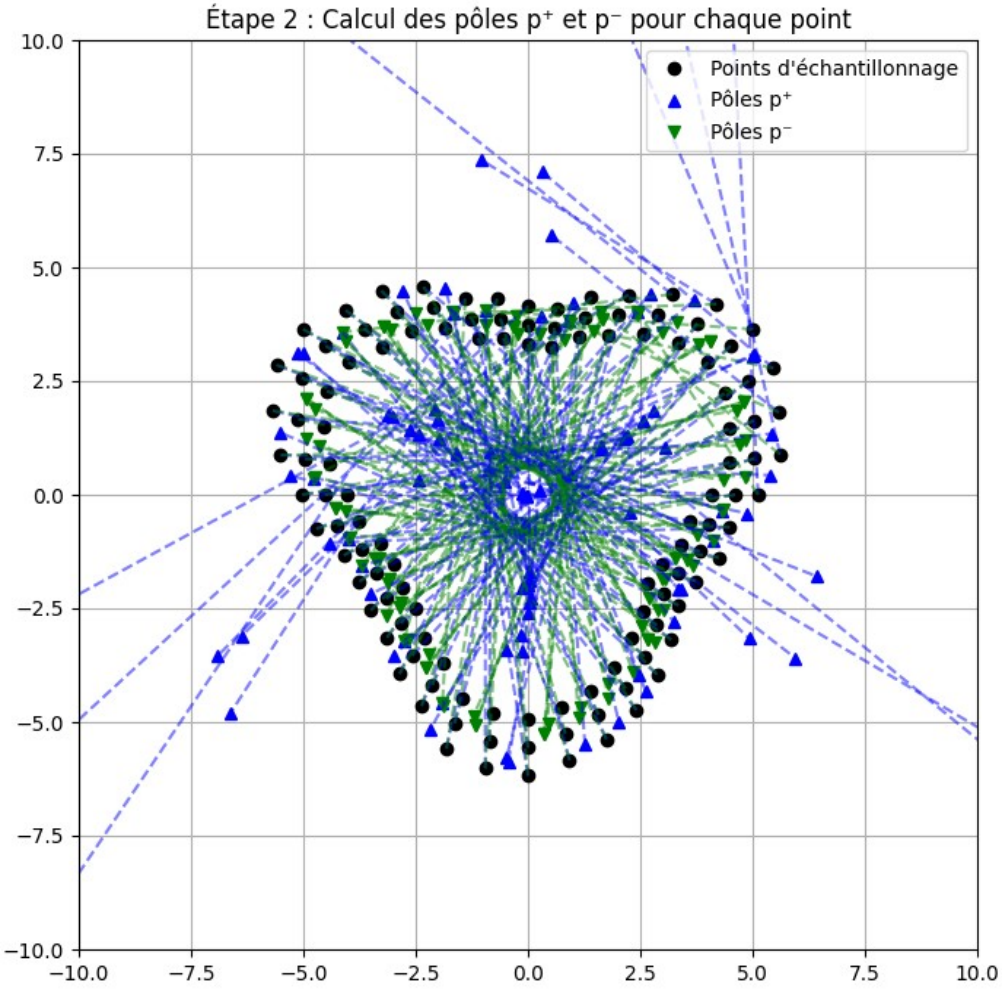
k = 10



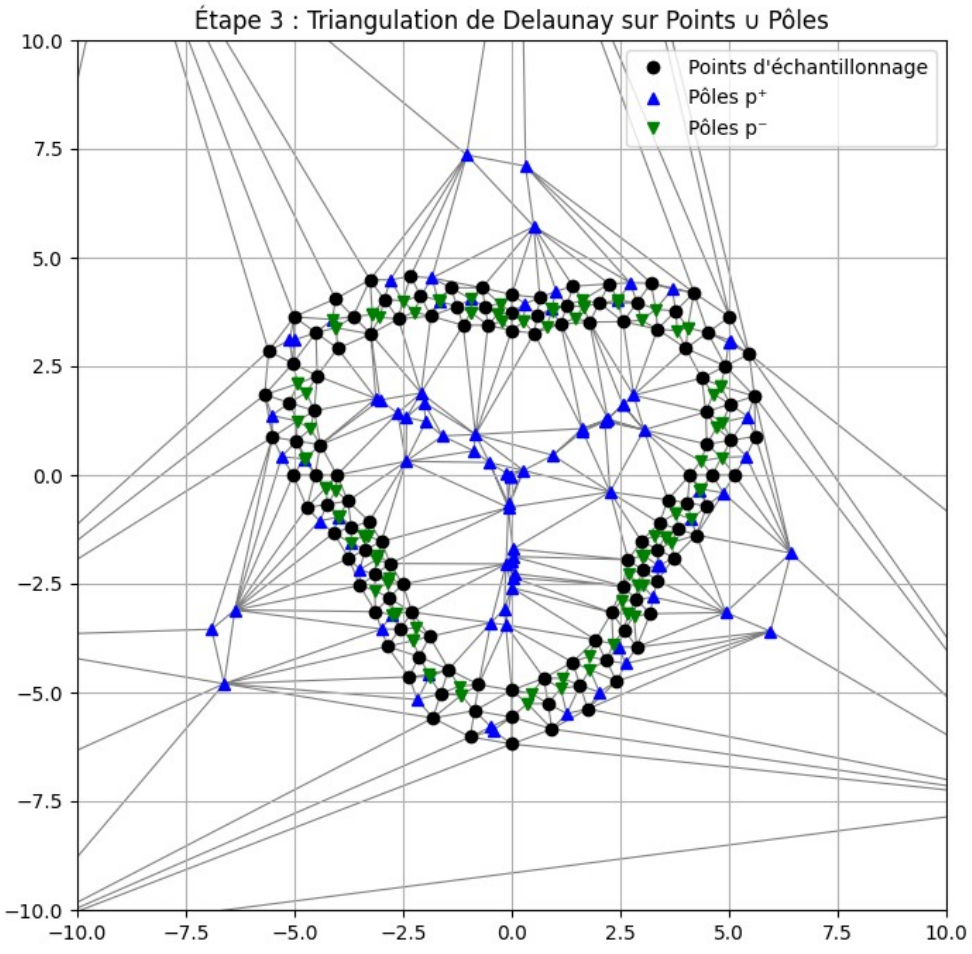
Nombre total d’opérations : 160.625

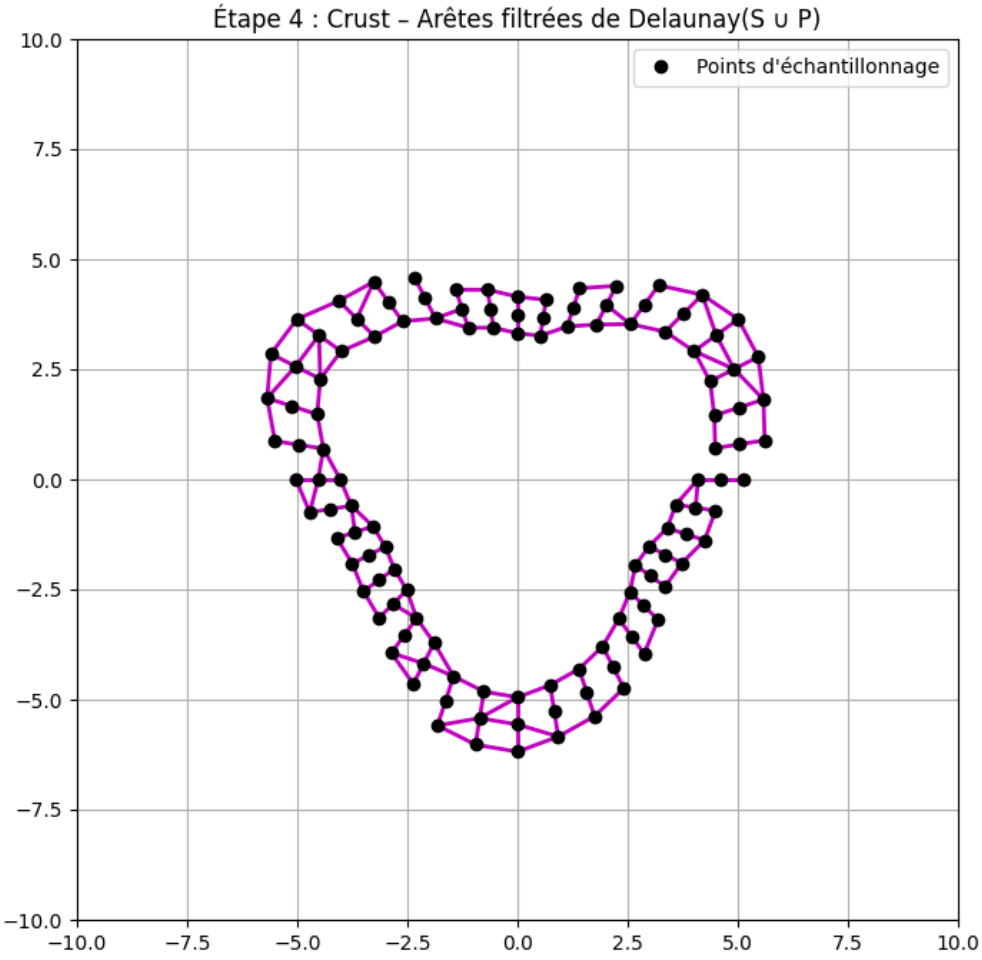
Annexe – Algorithme de Crust





Annexe – Algorithme de Crust





Ouverture – Algorithme de Crust – Complexité

$N = n^2$: Nombre de points

Diagramme de Voronoï : $O(N^2)$

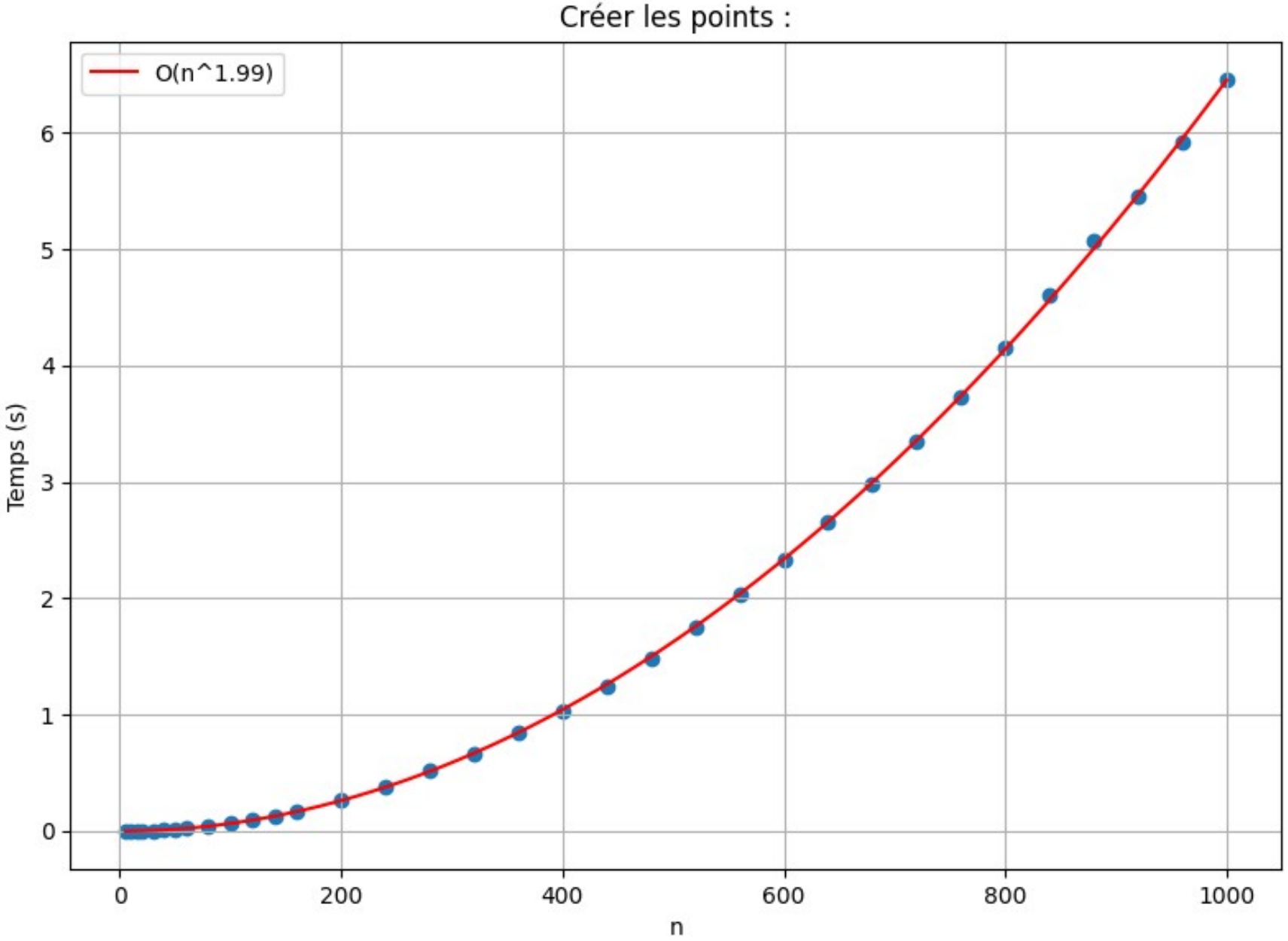
Calculer les pôles : $O(N)$

Triangulation de Delaunay : $O(N^2)$

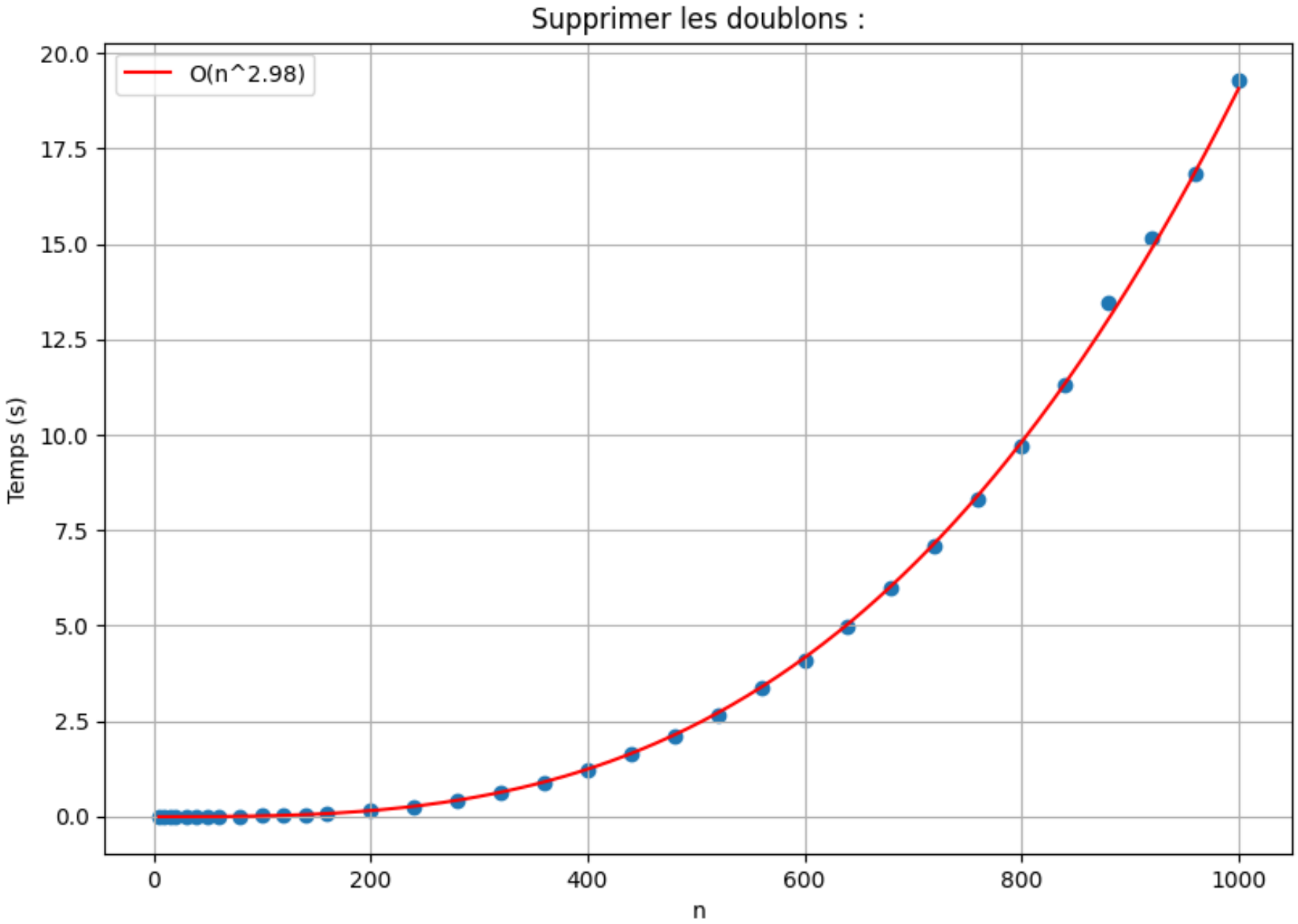
Filtrage des arêtes : $O(N^2)$

Complexité temporelle en $O(n^4)$

Annexe – Analyse pratique créer les points



Annexe – Analyse pratique supprimer les doublons



Annexe – Code – Makefile

```
# Compiler
CXX = g++

# Compiler flags
SAFE_CXXFLAGS = -std=c++11 -Wall -Wextra -fsanitize=address -g
CXXFLAGS = -std=c++11

# Include directories
INCLUDES = -I../glfw-3.4/include -I../glew-2.1.0/include

# Linker flags
LDFLAGS = -L../glew-2.1.0/lib -Wl,-rpath,../glew-2.1.0/lib

# Libraries to link
LIBS = -lGLEW -lglfw -lGL -lX11 -lpthread -lXrandr -lXi -ldl

# Source files (excluding main)
SRCS = point.cpp quadtree.cpp camera.cpp polynome.cpp

# Object files
OBJS = $(SRCS:.cpp=.o)

# Executable name
EXEC = plotmath

# Default target (non-secure build)
all: CXXFLAGS := $(CXXFLAGS)
all: $(EXEC)

# Secure build target
safe: CXXFLAGS := $(SAFE_CXXFLAGS)
safe: $(EXEC)

# Link the executable (include file.cpp directly)
$(EXEC): $(OBJS) file.cpp
```

Annexe – Code – Makefile

```
36          $(CXX) $(CXXFLAGS) $(INCLUDES) -g -O0 -o $@ file.cpp $(OBJS) $(LDFLAGS) $(LIBS)
37
38  # Compile source files into object files
39  %.o: %.cpp
40          $(CXX) $(CXXFLAGS) $(INCLUDES) -c $< -o $@
41
42  # Clean up build files
43  clean:
44          rm -f $(OBJS) $(EXEC)
45
46  .PHONY: all safe clean
```

Annexe – Code – camera.h

```
 1  #ifndef CAM
 2  #define CAM
 3
 4  #include <SFML/Graphics.hpp>
 5  #include <GL/glew.h>
 6  #include <GLFW/glfw3.h>
 7
 8  #include <iostream>
 9
10  #include "point.h"
11
12  class Quadtree;
13
14  class Camera{
15      public:
16          Camera(GLFWwindow* window, int window_width, int window_height);
17
18          void set_XY(double x, double y);
19          void set_dragXY(double x, double y);
20          void set_camera_dragXY(double x, double y);
21          void set_afficher_points(bool val);
22          void set_afficher_quadtrees(bool val);
23
24          std::shared_ptr<Point> get_XY();
25          std::shared_ptr<Point> get_dragXY();
26          std::shared_ptr<Point> get_cameraXY_drag();
27          bool get_flag_drag();
28          double get_zoom();
29          int get_mult();
30          int get_width();
31          int get_height();
32          bool get_afficher_points();
33          bool get_afficher_quadtrees();
34
35          double camera_pos_to_world_pos_x(double x);
```

Annexe – Code – camera.h

```
36          double camera_pos_to_world_pos_y(double y);
37          double world_pos_to_camera_pos_x(double x);
38          double world_pos_to_camera_pos_y(double y);
39
40          static void mouse_button_callback(GLFWwindow* window, int button, int action, int mods);
41          static void cursor_position_callback(GLFWwindow* window, double xpos, double ypos);
42          static void scroll_callback(GLFWwindow* window, double xoffset, double yoffset);
43          static void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods);
44
45          int render_points(Quadtree* quadtree, GLFWwindow* window);
46          int render_visible_points(Quadtree* quadtree, GLFWwindow* window, Point* last_point, bool*
   has_last_point);
47
48      private:
49          double zoom;
50
51          std::shared_ptr<Point> cameraXY;
52          std::shared_ptr<Point> cameraXY_drag;
53          std::shared_ptr<Point> dragXY;
54
55          bool flag_drag;
56
57          int MULT;
58          int WINDOW_WIDTH;
59          int WINDOW_HEIGHT;
60
61          bool afficher_points;
62          bool afficher_quadtrees;
63  };
64
65  #endif
```

Annexe – Code – camera.cpp

```
 1  #include "camera.h"
 2  #include "quadtree.h"
 3
 4  Camera::Camera(GLFWwindow* window, int window_width, int window_height)
 5  {
 6      zoom = 1.0f;
 7      cameraXY = std::make_shared<Point>(0, 0);
 8      cameraXY_drag = std::make_shared<Point>(0, 0);
 9      dragXY = std::make_shared<Point>(0, 0);
10      flag_drag = false;
11      afficher_points = false;
12      afficher_quadtrees = false;
13
14      MULT = 200;
15      WINDOW_WIDTH = window_width;
16      WINDOW_HEIGHT = window_height;
17
18      glfwSetWindowUserPointer(window, this);
19
20      glfwSetKeyCallback(window, key_callback);
21      glfwSetScrollCallback(window, scroll_callback);
22      glfwSetMouseButtonCallback(window, mouse_button_callback);
23      glfwSetCursorPosCallback(window, cursor_position_callback);
24  }
25
26  void Camera::set_XY(double x, double y){
27      cameraXY->set_x(x);
28      cameraXY->set_y(y);
29  }
30
31  void Camera::set_dragXY(double x, double y){
32      dragXY->set_x(x);
33      dragXY->set_y(y);
34  }
35
```

Annexe – Code – camera.cpp

```
36  void Camera::set_camera_dragXY(double x, double y){
37      cameraXY_drag->set_x(x);
38      cameraXY_drag->set_y(y);
39  }
40
41  void Camera::set_afficher_points(bool val){ afficher_points = val; }
42
43  void Camera::set_afficher_quadtrees(bool val){ afficher_quadtrees = val; }
44
45  std::shared_ptr<Point> Camera::get_XY(){ return cameraXY; }
46  std::shared_ptr<Point> Camera::get_dragXY(){ return dragXY; }
47  std::shared_ptr<Point> Camera::get_cameraXY_drag(){ return cameraXY_drag; }
48  bool Camera::get_flag_drag(){ return flag_drag; }
49
50  double Camera::get_zoom(){ return zoom; }
51
52  int Camera::get_mult(){ return MULT; }
53
54  int Camera::get_width(){ return WINDOW_WIDTH; }
55
56  int Camera::get_height(){ return WINDOW_HEIGHT; }
57
58  bool Camera::get_afficher_points(){ return afficher_points; }
59
60  bool Camera::get_afficher_quadtrees(){ return afficher_quadtrees; }
61
62
63  double Camera::camera_pos_to_world_pos_x(double c_point_x){
64      return (c_point_x/zoom)/MULT + cameraXY->x();
65  }
66
67  double Camera::camera_pos_to_world_pos_y(double c_point_y){
68      return (c_point_y/zoom)/MULT - cameraXY->y();
69  }
70
```

Annexe – Code – camera.cpp

```
 71  double Camera::world_pos_to_camera_pos_x(double point_x){
 72      return MULT*((point_x - cameraXY->x()) * zoom);
 73  }
 74
 75  double Camera::world_pos_to_camera_pos_y(double point_y){
 76      return MULT*((point_y + cameraXY->y()) * zoom);
 77  }
 78
 79  void Camera::mouse_button_callback(GLFWwindow* window, int button, int action, int /*mods*/) {
 80      Camera* camera = static_cast<Camera*>(glfwGetWindowUserPointer(window));
 81      if (camera) {
 82          if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS) {
 83
 84              double dragX, dragY;
 85              glfwGetCursorPos(window, &dragX, &dragY);
 86              camera->dragXY->set_x(dragX);
 87              camera->dragXY->set_y(dragY);
 88
 89              camera->cameraXY_drag->set_x(camera->cameraXY->x());
 90              camera->cameraXY_drag->set_y(camera->cameraXY->y());
 91              camera->flag_drag = true;
 92          }
 93          if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_RELEASE) {
 94              camera->flag_drag = false;
 95          }
 96      }
 97  }
 98
 99  void Camera::cursor_position_callback(GLFWwindow* window, double xpos, double ypos) {
100      Camera* camera = static_cast<Camera*>(glfwGetWindowUserPointer(window));
101      if (camera && camera->flag_drag == true) {
102          camera->cameraXY->set_x(camera->cameraXY_drag->x() + ((camera->dragXY->x() - xpos) /
    camera->zoom)/camera->get_mult());
103          camera->cameraXY->set_y(camera->cameraXY_drag->y() + ((camera->dragXY->y()- ypos) /
    camera->zoom)/camera->get_mult());
```

Annexe – Code – camera.cpp

```
104     }
105 }
106
107 void Camera::scroll_callback(GLFWwindow* window, double /*xoffset*/, double yoffset) {
108     Camera* camera = static_cast<Camera*>(glfwGetWindowUserPointer(window));
109     if (camera) {
110         const float zoomSpeed = 0.05f;
111         camera->zoom *= (1.0f + yoffset * zoomSpeed);
112     }
113 }
114
115 void actionQ() {
116     std::cout << "Touche Q pressée !" << std::endl;
117 }
118
119 void actionP() {
120     std::cout << "Touche P pressée !" << std::endl;
121 }
122
123 void Camera::key_callback(GLFWwindow* window, int key, int /*scancode*/, int action, int /*mods*/) {
124     Camera* camera = static_cast<Camera*>(glfwGetWindowUserPointer(window));
125     if (camera) {
126         const float moveSpeed = 10.0f / camera->zoom;
127
128         if (action == GLFW_PRESS || action == GLFW_REPEAT) {
129             switch (key) {
130                 case GLFW_KEY_LEFT:
131                     camera->cameraXY->set_x(camera->cameraXY->x() - moveSpeed);
132                     break;
133                 case GLFW_KEY_RIGHT:
134                     camera->cameraXY->set_x(camera->cameraXY->x() + moveSpeed);
135                     break;
136                 case GLFW_KEY_UP:
137                     camera->cameraXY->set_y(camera->cameraXY->y() + moveSpeed);
138                     break;
```

Annexe – Code – camera.cpp

```
139                 case GLFW_KEY_DOWN:
140                     camera->cameraXY->set_y(camera->cameraXY->y() - moveSpeed);
141                     break;
142             }
143         }
144 
145         if (action == GLFW_PRESS){
146             if (key == GLFW_KEY_A){//Touche Q car clavier inversé
147                 camera->set_afficher_quadtrees(! camera->get_afficher_quadtrees());
148             }
149             else if (key == GLFW_KEY_P){
150                 camera->set_afficher_points(! camera->get_afficher_points());
151             }
152         }
153     }
154 }
155 
156 int Camera::render_points(Quadtree* quadtree, GLFWwindow* window){
157     if (quadtree->divided){
158         int nb_pts = 0;
159         for (int i = 0; i < 4; i++){
160             nb_pts += render_points(quadtree->quadtrees[i], window);
161         }
162         return nb_pts;
163     }
164     else{
165         glPointSize(8.0f);
166         glLineWidth(2.0f);
167         glColor3f(0.0, 0.0, 0.0);
168         for (int i = 0; i < quadtree->point_counter; i++) {
169             auto p = quadtree->points->at(i);
170             float screenX = world_pos_to_camera_pos_x(p->x());
171             float screenY = world_pos_to_camera_pos_y(p->y());
172 
173             if (afficher_points){
```

Annexe – Code – camera.cpp

```
174                 glBegin(GL_POINTS);
175                     glVertex2f(screenX, screenY);
176                 glEnd();
177             }
178 
179 
180             for (size_t j = 0; j < p->suivants->size(); j++){
181                 std::shared_ptr<Point> suivant = p->suivants->at(j).lock();
182                 if (suivant != nullptr){
183                     float screenX_suivant = world_pos_to_camera_pos_x(suivant->x());
184                     float screenY_suivant = world_pos_to_camera_pos_y(suivant->y());
185                     glBegin(GL_LINES);
186                         glVertex2f(screenX, screenY);
187                         glVertex2f(screenX_suivant, screenY_suivant);
188                     glEnd();
189                 }
190             }
191         }
192 
193         if (afficher_quadtrees){
194             glColor3f(1.0, 0.0, 0.0);
195             glBegin(GL_LINES);
196                 glVertex2f(world_pos_to_camera_pos_x(quadtree->min_x),
    world_pos_to_camera_pos_y(quadtree->min_y));
197                 glVertex2f(world_pos_to_camera_pos_x(quadtree->min_x),
    world_pos_to_camera_pos_y(quadtree->max_y));
198             glEnd();
199 
200             glBegin(GL_LINES);
201                 glVertex2f(world_pos_to_camera_pos_x(quadtree->max_x),
    world_pos_to_camera_pos_y(quadtree->min_y));
202                 glVertex2f(world_pos_to_camera_pos_x(quadtree->max_x),
    world_pos_to_camera_pos_y(quadtree->max_y));
203             glEnd();
204 
```

Annexe – Code – camera.cpp

```
205             glBegin(GL_LINES);
206                 glVertex2f(world_pos_to_camera_pos_x(quadtree->min_x),
    world_pos_to_camera_pos_y(quadtree->min_y));
207                 glVertex2f(world_pos_to_camera_pos_x(quadtree->max_x),
    world_pos_to_camera_pos_y(quadtree->min_y));
208             glEnd();
209 
210             glBegin(GL_LINES);
211                 glVertex2f(world_pos_to_camera_pos_x(quadtree->min_x),
    world_pos_to_camera_pos_y(quadtree->max_y));
212                 glVertex2f(world_pos_to_camera_pos_x(quadtree->max_x),
    world_pos_to_camera_pos_y(quadtree->max_y));
213             glEnd();
214         }
215 
216         return quadtree->point_counter;
217     }
218  }
219 
220  int Camera::render_visible_points(Quadtree* quadtree, GLFWwindow* window, Point* last_point, bool*
    has_last_point){
221     double window_left_pos = camera_pos_to_world_pos_x(0);
222     double window_up_pos = camera_pos_to_world_pos_y(0);
223     double window_right_pos = camera_pos_to_world_pos_x(get_width());
224     double window_down_pos = camera_pos_to_world_pos_y(get_height());
225 
226     if (quadtree->divided){
227         int nb_pts = 0;
228         if (window_left_pos >= quadtree->center->x()){
229             if (window_up_pos >= quadtree->center->y()){
230                 nb_pts += render_visible_points(quadtree->quadtrees[1], window, last_point,
    has_last_point);
231             }
232             else if (window_down_pos <= quadtree->center->y()){
233                 nb_pts += render_visible_points(quadtree->quadtrees[3], window, last_point,
```

Annexe – Code – camera.cpp

```
    has_last_point);
234             }
235             else {
236                 nb_pts += render_visible_points(quadtree->quadtrees[1], window, last_point,
    has_last_point);
237                 nb_pts += render_visible_points(quadtree->quadtrees[3], window, last_point,
    has_last_point);
238             }
239         }
240         else if (window_right_pos <= quadtree->center->x()){
241             if (window_up_pos >= quadtree->center->y()){
242                 nb_pts += render_visible_points(quadtree->quadtrees[0], window, last_point,
    has_last_point);
243             }
244             else if (window_down_pos <= quadtree->center->y()){
245                 nb_pts += render_visible_points(quadtree->quadtrees[2], window, last_point,
    has_last_point);
246             }
247             else {
248                 nb_pts += render_visible_points(quadtree->quadtrees[0], window, last_point,
    has_last_point);
249                 nb_pts += render_visible_points(quadtree->quadtrees[2], window, last_point,
    has_last_point);
250             }
251         }
252         else{
253             nb_pts += render_visible_points(quadtree->quadtrees[0], window, last_point,
    has_last_point);
254             nb_pts += render_visible_points(quadtree->quadtrees[1], window, last_point,
    has_last_point);
255             nb_pts += render_visible_points(quadtree->quadtrees[2], window, last_point,
    has_last_point);
256             nb_pts += render_visible_points(quadtree->quadtrees[3], window, last_point,
    has_last_point);
257         }
```

Annexe – Code – camera.cpp

```
258             return nb_pts;
259         }
260         else{
261             return render_points(quadtree, window);
262         }
263     }
```

Annexe – Code – point.h

```
 1  #ifndef POINT
 2  #define POINT
 3
 4  #include <vector>
 5  #include <memory>
 6
 7  class Point : public std::enable_shared_from_this<Point> {
 8      public:
 9          Point(double x, double y);
10          Point();
11          ~Point();
12          double dist(std::shared_ptr<Point> point);
13
14          double x();
15          double y();
16
17          void set_x(double x);
18          void set_y(double y);
19
20          std::vector<std::weak_ptr<Point>>* suivants;
21
22      protected:
23          double coord_x;
24          double coord_y;
25  };
26
27  #endif
```

Annexe – Code – point.cpp

```cpp
#include "point.h"
#include <cstdlib>
#include <cmath>
#include <cstdio>

Point::Point(double x, double y){
    coord_x = x;
    coord_y = y;
    suivants = new std::vector<std::weak_ptr<Point>>;
}

Point::Point(){
    coord_x = 0;
    coord_y = 0;
    suivants = new std::vector<std::weak_ptr<Point>>;
}

Point::~Point(){
    delete suivants;
}

double Point::dist(std::shared_ptr<Point> point){
    if (!point) { return -1; }
    return sqrt(std::pow(point->x() - coord_x, 2) + std::pow(point->y() - coord_y, 2));
}

double Point::x(){ return coord_x; }

double Point::y(){ return coord_y; }

void Point::set_x(double x_){ coord_x = x_; }

void Point::set_y(double y_){ coord_y = y_; }
```

Annexe – Code – polynome.h

```
 1  #ifndef POLY
 2  #define POLY
 3
 4  #include <vector>
 5  #include <memory>
 6  #include "point.h"
 7  class Quadtree;
 8
 9  class Terme {
10      public:
11          Terme(int pow_x, int pow_y, double coeff);
12          std::unique_ptr<Terme> deriver_x_terme();
13          std::unique_ptr<Terme> deriver_y_terme();
14
15          int pow_x;
16          int pow_y;
17          double coeff;
18  };
19
20  class Polynome {
21      public:
22          Polynome();
23          ~Polynome();
24
25          std::vector<std::unique_ptr<Terme>>* termes;
26          void ajouter_terme(std::unique_ptr<Terme> terme);
27
28          Polynome* deriver_x();
29          Polynome* deriver_y();
30          Polynome* evaluer_x(double x);
31          Polynome* evaluer_y(double y);
32          double evaluer_xy(double x, double y);
33          double evaluer_y_valeur(double y);
34          double evaluer_x_valeur(double x);
35
```

Annexe – Code – polynome.h

```
36          void print(char* name);
37
38          std::vector<std::shared_ptr<Point>>* trouver_racine_y(double x, double y_min, double y_max,
    int nb_iter, int nb_pts, double precision);
39          std::vector<std::shared_ptr<Point>>* trouver_racine_x(double y, double x_min, double x_max,
    int nb_iter, int nb_pts, double precision);
40          double trouver_racine_newton_y(int nb_iter, double y_0, Polynome* poly_deriv, double
    precision);
41          double trouver_racine_newton_x(int nb_iter, double x_0, Polynome* poly_deriv, double
    precision);
42
43          Quadtree* generer_quadtree_poly(double x_max, double x_min, double y_max, double y_min, int
    nb_iter, int nb_pts, int nb_pts_par_quadtree,
44                                                  int nb_section_doublons, double precision, Polynome*
    self, std::vector<std::shared_ptr<Point>>* non_lies);
45  };
46
47  #endif
```

Annexe – Code – polynome.cpp

```
 1  #include "polynome.h"
 2  #include "quadtree.h"
 3  #include <cstdio>
 4  #include <cmath>
 5  #include <vector>
 6  #include <iostream>
 7  #include <memory>
 8  #include <cstdlib>
 9
10  // Implementation Terme
11  Terme::Terme(int pow_x_, int pow_y_, double coeff_) {
12      pow_x = pow_x_;
13      pow_y = pow_y_;
14      coeff = coeff_;
15  }
16
17  std::unique_ptr<Terme> Terme::deriver_x_terme() {
18      if (pow_x == 0) {
19          return std::unique_ptr<Terme>(new Terme(0, 0, 0));
20      }
21      return std::unique_ptr<Terme>(new Terme(pow_x - 1, pow_y, coeff * pow_x));
22  }
23
24  std::unique_ptr<Terme> Terme::deriver_y_terme() {
25      if (pow_y == 0) {
26          return std::unique_ptr<Terme>(new Terme(0, 0, 0));
27      }
28      return std::unique_ptr<Terme>(new Terme(pow_x, pow_y - 1, coeff * pow_y));
29  }
30  // Fin implementation Terme
31
32  // Implementation Polynome
33  Polynome::Polynome() {
34      termes = new std::vector<std::unique_ptr<Terme>>;
35  }
```

Annexe – Code – polynome.cpp

```
36
37  Polynome::~Polynome() {
38      delete termes;
39  }
40
41  void Polynome::print(char* name) {
42      printf("%s = ", name);
43      for (size_t i = 0; i < termes->size(); i++) {
44          auto& terme = (*termes)[i];
45          printf("%f X^%d Y^%d + ", terme->coeff, terme->pow_x, terme->pow_y);
46      }
47      printf("\n");
48  }
49
50  void Polynome::ajouter_terme(std::unique_ptr<Terme> terme) {
51      if (terme->coeff != 0) {
52          termes->push_back(std::move(terme));
53      }
54  }
55
56  Polynome* Polynome::deriver_x() {
57      Polynome* poly = new Polynome();
58      for (size_t i = 0; i < termes->size(); i++) {
59          auto& t = (*termes)[i];
60          auto t_derive = t->deriver_x_terme();
61          poly->ajouter_terme(std::move(t_derive));
62      }
63      return poly;
64  }
65
66  Polynome* Polynome::deriver_y() {
67      Polynome* poly = new Polynome();
68      for (size_t i = 0; i < termes->size(); i++) {
69          auto& t = (*termes)[i];
70          auto t_derive = t->deriver_y_terme();
```

Annexe – Code – polynome.cpp

```
71          poly->ajouter_terme(std::move(t_derive));
72      }
73      return poly;
74  }
75
76  Polynome* Polynome::evaluer_x(double val_x) {
77      Polynome* p = new Polynome();
78      size_t nb_termes = termes->size();
79      for (size_t i = 0; i < nb_termes; i++) {
80          std::unique_ptr<Terme> terme(new Terme(
81              0,
82              (*termes)[i]->pow_y,
83              (*termes)[i]->coeff * std::pow(val_x, (*termes)[i]->pow_x)
84          ));
85          p->ajouter_terme(std::move(terme));
86      }
87      return p;
88  }
89
90  Polynome* Polynome::evaluer_y(double val_y) {
91      Polynome* p = new Polynome();
92      size_t nb_termes = termes->size();
93      for (size_t i = 0; i < nb_termes; i++) {
94          std::unique_ptr<Terme> terme(new Terme(
95              (*termes)[i]->pow_x,
96              0,
97              (*termes)[i]->coeff * std::pow(val_y, (*termes)[i]->pow_y)
98          ));
99          p->ajouter_terme(std::move(terme));
100     }
101     return p;
102 }
103
104 double Polynome::evaluer_xy(double x, double y) {
105     double val = 0;
```

Annexe – Code – polynome.cpp

```
106      size_t n = termes->size();
107      for (size_t i = 0; i < n; i++) {
108          val += (*termes)[i]->coeff * std::pow(y, (*termes)[i]->pow_y) * std::pow(x,
    (*termes)[i]->pow_x);
109      }
110      return val;
111  }
112
113  double Polynome::evaluer_y_valeur(double val_y) {
114      double val = 0;
115      size_t n = termes->size();
116      for (size_t i = 0; i < n; i++) {
117          if (!(*termes)[i]) {
118              std::cerr << "Terme[" << i << "] est nul" << std::endl;
119              continue; // ou return NAN selon le contexte
120          }
121
122          val += (*termes)[i]->coeff * std::pow(val_y, (*termes)[i]->pow_y);
123      }
124      return val;
125  }
126
127  double Polynome::evaluer_x_valeur(double val_x) {
128      double val = 0;
129      size_t n = termes->size();
130      for (size_t i = 0; i < n; i++) {
131          if (!(*termes)[i]) {
132              std::cerr << "Terme[" << i << "] est nul" << std::endl;
133              continue; // ou return NAN selon le contexte
134          }
135          val += (*termes)[i]->coeff * std::pow(val_x, (*termes)[i]->pow_x);
136      }
137      return val;
138  }
139
```

Annexe – Code – polynome.cpp

```
140  double Polynome::trouver_racine_newton_y(int nb_iter, double y_0, Polynome* poly_deriv, double
    precision) {
141      double y_n = y_0;
142      double y_n1 = y_0;
143      for (int i = 0; i < nb_iter; i++) {
144          y_n = y_n1;
145          double f_yn = evaluer_y_valeur(y_n);
146          double f_deriv_yn = poly_deriv->evaluer_y_valeur(y_n);
147          if (f_deriv_yn == 0) {
148              return NAN; // Evite la division par zéro
149          }
150          y_n1 = y_n - (f_yn / f_deriv_yn);
151
152          if (std::abs(y_n1 - y_n) < precision) {
153              return y_n1; // convergé, on peut sortir plus tôt
154          }
155      }
156      return NAN;
157  }
158
159  double Polynome::trouver_racine_newton_x(int nb_iter, double x_0, Polynome* poly_deriv, double
    precision) {
160      double x_n = x_0;
161      double x_n1 = x_0;
162      for (int i = 0; i < nb_iter; i++) {
163          x_n = x_n1;
164          double f_xn = evaluer_x_valeur(x_n);
165          double f_deriv_xn = poly_deriv->evaluer_x_valeur(x_n);
166          if (f_deriv_xn == 0) {
167              return NAN; // Evite la division par zéro
168          }
169          x_n1 = x_n - (f_xn / f_deriv_xn);
170      }
171
172      if (std::abs(x_n - x_n1) < precision) {
```

Annexe – Code – polynome.cpp

```
173             return x_n1;
174         }
175         return NAN;
176     }
177
178     std::vector<std::shared_ptr<Point>>* Polynome::trouver_racine_y(double x, double y_min, double
    y_max,
179                                                                    int nb_iter, int nb_pts, double
    precision) {
180         auto racines = new std::vector<std::shared_ptr<Point>>;
181         std::unique_ptr<Polynome> p_eval(evaluer_x(x));
182         std::unique_ptr<Polynome> p_deriv(p_eval->deriver_y());
183
184         double pas = (y_max - y_min) / nb_pts;
185         for (double y = y_min; y <= y_max; y += pas) {
186             double racine = p_eval->trouver_racine_newton_y(nb_iter, y, p_deriv.get(), precision);
187             if (racine > y_min && racine < y_max) {
188                 racines->push_back(std::make_shared<Point>(x, racine));
189             }
190         }
191         return racines;
192     }
193
194     std::vector<std::shared_ptr<Point>>* Polynome::trouver_racine_x(double y, double x_min, double
    x_max,
195                                                                    int nb_iter, int nb_pts, double
    precision) {
196         auto racines = new std::vector<std::shared_ptr<Point>>;
197         std::unique_ptr<Polynome> p_eval(evaluer_y(y));
198         std::unique_ptr<Polynome> p_deriv(p_eval->deriver_x());
199
200         double pas = (x_max - x_min) / nb_pts;
201         for (double x = x_min; x <= x_max; x += pas) {
202             double racine = p_eval->trouver_racine_newton_x(nb_iter, x, p_deriv.get(), precision);
203             if (racine > x_min && racine < x_max) {
```

Annexe – Code – polynome.cpp

```
204             racines->push_back(std::make_shared<Point>(racine, y));
205         }
206     }
207     return racines;
208 }
209
210 Quadtree* Polynome::generer_quadtree_poly(double x_max, double x_min, double y_max, double y_min,
211                                           int nb_iter, int nb_pts, int nb_pts_par_quadtree,
212                                           int nb_section_doublons, double precision,
213                                           Polynome* self, std::vector<std::shared_ptr<Point>>*
    non_lies){
214     Quadtree* quadtree = new Quadtree(4, nb_pts_par_quadtree, self);
215     quadtree->max_x = x_max;
216     quadtree->min_x = x_min;
217     quadtree->max_y = y_max;
218     quadtree->min_y = y_min;
219     quadtree->center->set_x((x_min + x_max)/2);
220     quadtree->center->set_y((y_min + y_max)/2);
221
222     double pas_y = (y_max - y_min) / nb_pts;
223     for (double y = y_min; y <= y_max; y += pas_y) {
224         auto racines = trouver_racine_x(y, x_min, x_max, nb_iter, nb_pts, precision);
225         quadtree->push_vector(racines);
226         delete racines;
227     }
228
229     double pas_x = (x_max - x_min) / nb_pts;
230     for (double x = x_min; x <= x_max; x += pas_x) {
231         auto racines = trouver_racine_y(x, y_min, y_max, nb_iter, nb_pts, precision);
232         quadtree->push_vector(racines);
233         delete racines;
234     }
235
236     quadtree->nettoyer_double(x_min, x_max, y_min, y_max, nb_section_doublons, precision*0.1);
237
```

Annexe – Code – polynome.cpp

```
238       quadtree->divide_space();
239
240       auto partition = new std::vector<int>;
241       int partition_id = 0;
242       quadtree->lier_points(non_lies, partition, partition_id);
243       delete partition;
244
245       return quadtree;
246   }
```

Annexe – Code – quadtree.h

```
 1  #ifndef QUAD
 2  #define QUAD
 3  #include "point.h"
 4
 5  class Polynome;
 6  class Camera;
 7  class Quadtree;
 8
 9  typedef struct int_pile{int val; int_pile* next;} int_pile;
10  typedef struct quad_courb { Quadtree* quad;
11                              double courb;} quad_courb;
12  quad_courb* quad_courb_creer(Quadtree* q, double c);
13
14  class Quadtree{
15      public:
16          Quadtree(int subdiv, int nb_max_point, Polynome* poly);
17          ~Quadtree();
18          void copier(Quadtree* quadtree);
19
20          bool appartient(std::shared_ptr<Point> point);
21          void add_point(std::shared_ptr<Point> point);
22          void push_vector(std::vector<std::shared_ptr<Point>>* points);
23          void divide_space();
24          int get_point_count();
25
26          std::shared_ptr<Point> get_center();
27          bool is_divided();
28          void nettoyer_double(double x_min, double x_max, double y_min, double y_max, int n, double
    precision);
29          void lier_points(std::vector<std::shared_ptr<Point>>* non_lies, std::vector<int>*
partition,
     int& partition_id);
30          int zone_non_vide(double x_min, double x_max, double y_min, double y_max);
31          bool zone_recherche(double& x_min, double& x_max, double& y_min, double& y_max);
32          double courbure_moy_quadtree();
33
```

Annexe – Code – quadtree.h

```
34          void augmenter_qualite_visible(Camera* camera, int nb_iter, double precision, int version,
   Quadtree* self,
35                                                  bool qualite, std::vector<quad_courb*>*
tableau_quad_courb);
36          void augmenter_qualite_V1_ajout(int nb_iter, double precision);
37          void augmenter_qualite_V2_creation(int nb_iter, double precision, double facteur);
38
39          Quadtree** quadtrees;
40          std::vector<std::shared_ptr<Point>>* points;
41          Polynome* forme;
42          double min_x, min_y, max_x, max_y;
43          bool divided;
44          std::shared_ptr<Point> center;
45          int subdivision;
46          int nb_pts_par_quadtree;
47          int point_counter;
48  };
49  #endif
```

Annexe – Code – quadtree.cpp

```
 1  #include "quadtree.h"
 2  #include "polynome.h"
 3  #include "camera.h"
 4
 5  #include <cstdlib>
 6  #include <iostream>
 7  #include <float.h>
 8  #include <algorithm>
 9  #include <cmath>
10  #include <vector>
11
12  Quadtree::Quadtree(int subdiv, int nb_pts_max, Polynome* poly){
13      quadtrees = (Quadtree**)malloc(sizeof(Quadtree*)*subdiv);
14
15      divided = false;
16      max_x = -DBL_MAX;
17      max_y = -DBL_MAX;
18      min_x = DBL_MAX;
19      min_y = DBL_MAX;
20      center = std::make_shared<Point>(0, 0);
21      point_counter = 0;
22      subdivision = subdiv;
23      nb_pts_par_quadtree = nb_pts_max;
24      points = new std::vector<std::shared_ptr<Point>>();
25      forme = poly;
26  }
27
28  Quadtree::~Quadtree(){
29      if (divided){
30          for (int i = 0; i < subdivision; i++){
31              delete quadtrees[i];
32          }
33      }
34      else{
35          if (points != nullptr){
```

Annexe – Code – quadtree.cpp

```
36              points->clear();
37          }
38      }
39      free(quadtrees);
40      delete points;
41  }
42
43  void Quadtree::add_point(std::shared_ptr<Point> point){
44      point_counter++;
45      points->push_back(point);
46  }
47
48  void Quadtree::push_vector(std::vector<std::shared_ptr<Point>>* points){
49      for (size_t i = 0; i < points->size(); i++){
50          add_point((*points)[i]);
51      }
52  }
53
54  void Quadtree::divide_space(){
55      if (point_counter >= nb_pts_par_quadtree){
56          divided = true;
57          for (int i = 0; i < subdivision; i++){
58              quadtrees[i] = new Quadtree(subdivision, nb_pts_par_quadtree, forme);
59          }
60          quadtrees[0]->min_x = min_x;
61          quadtrees[0]->max_y = max_y;
62          quadtrees[0]->max_x = center->x();
63          quadtrees[0]->min_y = center->y();
64          quadtrees[0]->center->set_x((quadtrees[0]->min_x + quadtrees[0]->max_x)/2);
65          quadtrees[0]->center->set_y((quadtrees[0]->min_y + quadtrees[0]->max_y)/2);
66
67
68          quadtrees[1]->min_x = center->x();
69          quadtrees[1]->max_y = max_y;
70          quadtrees[1]->max_x = max_x;
```

Annexe – Code – quadtree.cpp

```
 71          quadtrees[1]->min_y = center->y();
 72          quadtrees[1]->center->set_x((quadtrees[1]->min_x + quadtrees[1]->max_x)/2);
 73          quadtrees[1]->center->set_y((quadtrees[1]->min_y + quadtrees[1]->max_y)/2);
 74
 75          quadtrees[2]->min_x = min_x;
 76          quadtrees[2]->max_y = center->y();
 77          quadtrees[2]->max_x = center->x();
 78          quadtrees[2]->min_y = min_y;
 79          quadtrees[2]->center->set_x((quadtrees[2]->min_x + quadtrees[2]->max_x)/2);
 80          quadtrees[2]->center->set_y((quadtrees[2]->min_y + quadtrees[2]->max_y)/2);
 81
 82          quadtrees[3]->min_x = center->x();
 83          quadtrees[3]->max_y = center->y();
 84          quadtrees[3]->max_x = max_x;
 85          quadtrees[3]->min_y = min_y;
 86          quadtrees[3]->center->set_x((quadtrees[3]->min_x + quadtrees[3]->max_x)/2);
 87          quadtrees[3]->center->set_y((quadtrees[3]->min_y + quadtrees[3]->max_y)/2);
 88
 89          for (size_t i = 0; i < points->size(); i++){
 90              if ((*points)[i]->x() < center->x()){
 91                  if ((*points)[i]->y() < center->y()){
 92                      quadtrees[2]->add_point((*points)[i]);
 93                  }
 94                  else{
 95                      quadtrees[0]->add_point((*points)[i]);
 96                  }
 97              }
 98              else{
 99                  if ((*points)[i]->y() > center->y()){
100                      quadtrees[1]->add_point((*points)[i]);
101                  }
102                  else{
103                      quadtrees[3]->add_point((*points)[i]);
104                  }
105              }
```

Annexe – Code – quadtree.cpp

```
106          }
107          points->clear();
108          quadtrees[0]->divide_space();
109          quadtrees[1]->divide_space();
110          quadtrees[2]->divide_space();
111          quadtrees[3]->divide_space();
112      }
113  }
114
115  bool doublon(int i, int j, std::vector<std::shared_ptr<Point>>* points, double precision){
116      if (std::abs((*points)[i]->x() - (*points)[j]->x()) < precision){
117          if (std::abs((*points)[i]->y() - (*points)[j]->y()) < precision){
118              return true;
119          }
120      }
121      return false;
122  }
123
124  void Quadtree::nettoyer_double(double x_min, double x_max, double y_min, double y_max, int
    nb_section_doublons, double precision){
125      int n = nb_section_doublons;
126      double section_x = std::fmax(std::abs(x_min), std::abs(x_max))/n;
127      double section_y = std::fmax(std::abs(y_min), std::abs(y_max))/n;
128
129      // Création de la grille
130      std::vector<int>*** grille = (std::vector<int>***)malloc(sizeof(std::vector<int>**)*n);
131      for (int i = 0; i < n; i++){
132          grille[i] = (std::vector<int>**)malloc(sizeof(std::vector<int>*) * n);
133          for (int j = 0; j < n; j++){
134              grille[i][j] = new std::vector<int>;
135          }
136      }
137
138      bool* a_supprimer = (bool*)malloc(sizeof(bool)*point_counter);
139      for (int i = 0; i < point_counter; i++){
```

Annexe – Code – quadtree.cpp

```
140         a_supprimer[i] = false;
141     }
142
143     // Remplissage
144     for (int i = 0; i < point_counter; i++){
145         std::shared_ptr<Point> point = (*points)[i];
146
147         //On peut avoir x = nb_sections_doublons si |x| = max(|x_min|,|x_max|)
148         int x = std::min((int)(std::abs(point->x()) / section_x), nb_section_doublons-1);
149         int y = std::min((int)(std::abs(point->y()) / section_y), nb_section_doublons-1);
150         grille[x][y]->push_back(i);
151     }
152
153     // Comparaison
154     for (int x = 0; x < n; x++){
155         for (int y = 0; y < n; y++){
156             if (grille[x][y] != NULL){
157                 std::vector<int>* pile = grille[x][y];
158
159
160                 for (size_t i = 0; i < pile->size(); i++){
161                     int x = pile->at(i);
162                     for (size_t j = i+1; j < pile->size(); j++){
163                         int y = pile->at(j);
164                         if (doublon(x, y, points, precision)){
165                             a_supprimer[x] = true;
166                             break;
167                         }
168
169                     }
170                 }
171                 delete pile;
172             }
173         }
174     }
```

Annexe – Code – quadtree.cpp

```
175
176     // Suppression des doublons
177     for (int i = point_counter - 1; i >= 0; i--){
178         if (a_supprimer[i]){
179             points->erase(points->begin() + i);
180             point_counter--;
181         }
182     }
183
184     // Suppression des objets
185     free(a_supprimer);
186     for(int x = 0; x < n; x++){
187         free(grille[x]);
188     }
189     free(grille);
190 }
191
192 //Structure unir et trouver
193 std::vector<int>* creer_partition(int n){
194     std::vector<int>* p = new std::vector<int>;
195     for(int i = 0; i < n; i++){
196         p->push_back(0);
197     }
198
199     for(int i = 0; i < n; i++){
200         p->at(i) = i;
201     }
202
203     return p;
204 }
205
206 int trouver(std::vector<int>* p, int i){
207     return p->at(i);
208 }
209
```

Annexe – Code – quadtree.cpp

```
210  void unir(std::vector<int>* p, int i, int j, int n){
211      int repr_i = trouver(p, i);
212      int repr_j = trouver(p, j);
213
214      for (int k = 0; k < n; k++){
215          if (p->at(k) == repr_i){
216              p->at(k) = repr_j;
217          }
218      }
219  }
220  //Fin structure unir et trouver
221
222  typedef struct {double dist;
223                  int i;
224                  int j;
225              } arete;
226
227  arete* creer_arete(double d, int i, int j){
228      arete* a = (arete*)malloc(sizeof(arete));
229      a->dist = d;
230      a->i = i;
231      a->j = j;
232      return a;
233  }
234
235  int compare_aretes(arete* a, arete* b){ return a->dist < b->dist; }
236
237  std::vector<arete*>* trier_distance(std::vector<std::shared_ptr<Point>>* points){
238      int n = points->size();
239      std::vector<arete*>* distances = new std::vector<arete*>;
240      for (int i = 0; i < n; i++){
241          for (int j = i+1; j < n; j++){
242              double d = points->at(i)->dist(points->at(j));
243              distances->push_back(creer_arete(d, i, j));
244          }
```

Annexe – Code – quadtree.cpp

```
245      }
246      std::sort(distances->begin(), distances->end(), compare_aretes);
247      return distances;
248  }
249
250  bool est_uni(std::vector<int>* partition){
251      int n = partition->size();
252      for(int i = 0; i < n-1; i++){
253          if (partition->at(i) != partition->at(i+1)){
254              return false;
255          }
256      }
257      return true;
258  }
259
260  void Quadtree::lier_points(std::vector<std::shared_ptr<Point>>* non_lies, std::vector<int>*
    partition, int& partition_id){
261      if (divided){
262          for (int i = 0; i < 4; i++){
263              quadtrees[i]->lier_points(non_lies, partition, partition_id);
264              partition_id++;
265          }
266
267          int n = non_lies->size();
268          std::vector<arete*>* distances = trier_distance(non_lies);
269          int* vus = (int*)malloc(sizeof(int)*n);
270          for (int i = 0; i < n; i++){
271              vus[i] = 0;
272          }
273
274          //kruskal, mais certains points sont déjà relié par la récursion
275          bool flag = true;
276          int indice = 0;
277          while (flag && indice < n){
278              arete* a = distances->at(indice);
```

Annexe – Code – quadtree.cpp

```
279             int i = a->i;
280             int j = a->j;
281             if (trouver(partition, i) !=  trouver(partition, j)){
282                 unir(partition, i, j, n);
283                 vus[i]++;
284                 vus[j]++;
285                 non_lies->at(i)->suivants->push_back(non_lies->at(j));
286                 non_lies->at(j)->suivants->push_back(non_lies->at(i));
287             }
288             if (est_uni(partition)){
289                 flag = false;
290             }
291             indice++;
292         }
293 
294         for (size_t i = 0; i < distances->size(); i++){
295             free(distances->at(i));
296         }
297         delete distances;
298 
299         std::vector<std::shared_ptr<Point>>* nouv_non_lies = new
    std::vector<std::shared_ptr<Point>>;
300         std::vector<int>* nouv_partition = new std::vector<int>;
301         partition_id++;
302         for (int i = 0; i < n; i++){
303             if (vus[i] == 0){
304                 nouv_non_lies->push_back(non_lies->at(i));
305                 nouv_partition->push_back(partition_id);
306             }
307         }
308         free(vus);
309 
310         non_lies->clear();
311         non_lies->insert(non_lies->end(), nouv_non_lies->begin(), nouv_non_lies->end());
312         delete nouv_non_lies;
```

Annexe – Code – quadtree.cpp

```
313
314         partition->clear();
315         partition->insert(partition->end(), nouv_partition->begin(), nouv_partition->end());
316         delete nouv_partition;
317     }
318     else{
319         //kruskal
320         int n = points->size();
321
322         //on veut savoir combien de fois chaque points sera lié pour l'ajouter ou non à non_lies
323         int* vus = (int*)malloc(sizeof(int)*n);
324         for (int i = 0; i < n; i++){
325             vus[i] = 0;
326         }
327
328         std::vector<int>* p = creer_partition(n);
329         std::vector<arete*>* distances = trier_distance(points);
330         int compteur = 0;
331         int indice = 0;
332         while (compteur < n-1){
333             arete* a = distances->at(indice);
334             int i = a->i;
335             int j = a->j;
336             if (trouver(p, i) !=  trouver(p, j)){
337                 unir(p, i, j, n);
338                 vus[i]++;
339                 vus[j]++;
340                 points->at(i)->suivants->push_back(points->at(j));
341                 points->at(j)->suivants->push_back(points->at(i));
342                 compteur++;
343             }
344             indice++;
345         }
346
347         delete p;
```

Annexe – Code – quadtree.cpp

```
348         for (int i = (int)distances->size()-1 ; i >=0 ; i--){
349             free(distances->at(i));
350         }
351         delete distances;
352 
353         partition_id++;
354         for (int i = 0; i < n; i++){
355             if (vus[i] == 1){
356                 //on indique que points[i] est dans la composante "numero_quadtree"
357                 non_lies->push_back(points->at(i));
358                 partition->push_back(partition_id);
359             }
360         }
361         free(vus);
362 
363     }
364 }
365 
366 int Quadtree::zone_non_vide(double x_min, double x_max, double y_min, double y_max){
367     if (forme->evaluer_xy(x_min, y_min) > 0 &&
368         forme->evaluer_xy(x_min, y_max) > 0 &&
369         forme->evaluer_xy(x_max, y_min) > 0 &&
370         forme->evaluer_xy(x_max, y_max) > 0){
371             return 0;
372         }
373     if (forme->evaluer_xy(x_min, y_min) < 0 &&
374         forme->evaluer_xy(x_min, y_max) < 0 &&
375         forme->evaluer_xy(x_max, y_min) < 0 &&
376         forme->evaluer_xy(x_max, y_max) < 0){
377             return 0;
378         }
379     return 1;
380 }
381 
382 bool Quadtree::zone_recherche(double& x_min, double& x_max, double& y_min, double& y_max){
```

Annexe – Code – quadtree.cpp

```
383     //On suppose que le quadtree est une feuille
384     if (!divided){
385         if (zone_non_vide(x_min, x_max, y_min, y_max) == 0){
386             return false;
387         }
388 
389         double centre_x = (x_min + x_max)/2;
390         double centre_y = (y_min + y_max)/2;
391 
392         int cadran0 = zone_non_vide(x_min, centre_x, centre_y, y_max);
393         int cadran1 = zone_non_vide(centre_x, x_max, centre_y, y_max);
394         int cadran2 = zone_non_vide(x_min, centre_x, y_min, centre_y);
395         int cadran3 = zone_non_vide(centre_x, x_max, y_min, centre_y);
396 
397         int somme = cadran0 + cadran1 + cadran2 + cadran3;
398 
399         if (somme == 3 || somme == 4){
400             return true;
401         }
402         else if (somme == 1){
403             if (cadran0 == 1){
404                 x_max = centre_x;
405                 y_min = centre_y;
406                 return zone_recherche(x_min, x_max, y_min, y_max);
407             }
408             else if (cadran1 == 1){
409                 x_min = centre_x;
410                 y_min = centre_y;
411                 return zone_recherche(x_min, x_max, y_min, y_max);
412             }
413             else if (cadran2 == 1){
414                 x_max = centre_x;
415                 y_max = centre_y;
416                 return zone_recherche(x_min, x_max, y_min, y_max);
417             }
```

Annexe – Code – quadtree.cpp

```
418             else if (cadran3 == 1){
419                 x_min = centre_x;
420                 y_max = centre_y;
421                 return zone_recherche(x_min, x_max, y_min, y_max);
422             }
423         }
424         else if(somme==2){
425             if (cadran0 == 1){
426                 if (cadran1 == 1){ // haut
427                     y_min = centre_y;
428                     return true;
429                 }
430                 else if(cadran2 == 1){ // gauche
431                     x_max = centre_x;
432                     return true;
433                 }
434             }
435             else if (cadran3 == 1){
436                 if (cadran1 == 1){ // droite
437                     x_min = centre_x;
438                     return true;
439                 }
440                 else if(cadran2 == 1){
441                     y_max = centre_y;
442                     return true;
443                 }
444             }
445             else {
446                 return true;
447             }
448         }
449     }
450     return false;
451     //n'arrive jamais
452 }
```

Annexe – Code – quadtree.cpp

```
453
454  void Quadtree::copier(Quadtree* quadtree){
455      //this n'est pas divisé, les quadtrees enfants n'étaient pas initialisé
456      delete points;
457
458      free(quadtrees);
459      quadtrees = quadtree->quadtrees;
460      points = quadtree->points;
461      forme = quadtree->forme;
462
463      min_x = quadtree->min_x;
464      min_y = quadtree->min_y;
465      max_x = quadtree->max_x;
466      max_y = quadtree->max_y;
467      center = quadtree->center;
468      divided = quadtree->divided;
469
470      subdivision = quadtree->subdivision;
471      nb_pts_par_quadtree = quadtree->nb_pts_par_quadtree;
472      point_counter = quadtree->point_counter;
473
474
475      quadtree->divided = false;
476      quadtree->quadtrees = nullptr;
477      quadtree->points = nullptr;
478
479      delete quadtree;
480  }
481
482  bool Quadtree::appartient(std::shared_ptr<Point> point){
483      if (std::isnan(point->x()) || std::isnan(point->x())){
484          return false;
485      }
486      if (point->x() > max_x || point->x() < min_x ||
487          point->y() > max_y || point->y() < min_y){
```

Annexe – Code – quadtree.cpp

```
488         return false;
489     }
490     return true;
491 }
492
493 typedef struct triplet{ std::shared_ptr<Point> p1;
494     std::shared_ptr<Point> p2;
495     std::shared_ptr<Point> nouv_point;
496
497 } triplet;
498
499 std::shared_ptr<triplet> creer_triplet(std::shared_ptr<Point> p1, std::shared_ptr<Point> p2,
    std::shared_ptr<Point> nouv_point){
500     std::shared_ptr<triplet> t = std::make_shared<triplet>();
501     t->p1 = p1;
502     t->p2 = p2;
503     t->nouv_point = nouv_point;
504     return t;
505 }
506
507 void lier_triplets(std::shared_ptr<Point> p1, std::shared_ptr<Point> p2, std::shared_ptr<Point>
    p_m){
508     for(int i = p1->suivants->size()-1; i >=0 ; i--){
509         if (p1->suivants->at(i).lock() == p2 || p1->suivants->at(i).expired()){
510             p1->suivants->erase(p1->suivants->begin() + i);
511             p1->suivants->push_back(p2);
512             break;
513         }
514     }
515     for(int i = p2->suivants->size()-1; i >=0 ; i--){
516         if (p2->suivants->at(i).lock() == p1 || p2->suivants->at(i).expired()){
517             p2->suivants->erase(p2->suivants->begin() + i);
518             p2->suivants->push_back(p1);
519             break;
520         }
```

Annexe – Code – quadtree.cpp

```
521      }
522      p_m->suivants->push_back(p1);
523      p_m->suivants->push_back(p2);
524  }
525
526  void Quadtree::augmenter_qualite_V1_ajout(int nb_iter, double precision){
527      //On suppose qu'on est arrivé dans une feuille
528      if (!divided){
529          std::vector<std::shared_ptr<triplet>>* pile = new std::vector<std::shared_ptr<triplet>>;
530          for (size_t i = 0; i < points->size(); i++){
531              std::shared_ptr<Point> point = points->at(i);
532
533              std::shared_ptr<Point> p1 = nullptr;
534              if (point->suivants->size() >= 1){
535                  p1 = point->suivants->at(0).lock();
536              }
537
538              std::shared_ptr<Point> p2 = nullptr;
539              if (point->suivants->size() >= 2){
540                  p2 = point->suivants->at(1).lock();
541              }
542
543              if (p1 != nullptr){
544                  double diff_x = std::abs(point->x() -  p1->x());
545                  double diff_y = std::abs(point->y() -  p1->y());
546
547                  double moy_x = (point->x() + p1->x())/2;
548                  double moy_y = (point->y() + p1->y())/2;
549
550                  std::shared_ptr<Point> nouv_point = std::make_shared<Point>(max_x+1, 0);
551                  //On l'initialise à une valeur en dehors du quadtree
552
553                  if (diff_x > diff_y){
554                      Polynome* p_eval = forme->evaluer_x(moy_x);
555                      Polynome* p_deriv = p_eval->deriver_y();
```

Annexe – Code – quadtree.cpp

```
  556                     double y = p_eval->trouver_racine_newton_y(nb_iter, moy_y, p_deriv,
precision);
  557                     nouv_point->set_x(moy_x);
  558                     nouv_point->set_y(y);
  559                     delete p_eval;
  560                     delete p_deriv;
  561                 }
  562                 else{
  563                     Polynome* p_eval = forme->evaluer_y(moy_y);
  564                     Polynome* p_deriv = p_eval->deriver_x();
  565                     double x = p_eval->trouver_racine_newton_x(nb_iter, moy_x, p_deriv,
precision);
  566                     nouv_point->set_x(x);
  567                     nouv_point->set_y(moy_y);
  568                     delete p_eval;
  569                     delete p_deriv;
  570                 }
  571
  572                 if (appartient(nouv_point)){
  573                     std::shared_ptr<triplet> t = creer_triplet(point, p1, nouv_point);
  574                     pile->push_back(t);
  575                 }
  576             }
  577
  578             if (p2 != nullptr){
  579                 double diff_x = std::abs(point->x() -  p2->x());
  580                 double diff_y = std::abs(point->y() -  p2->y());
  581
  582                 double moy_x = (point->x() + p2->x())/2;
  583                 double moy_y = (point->y() + p2->y())/2;
  584
  585                 std::shared_ptr<Point> nouv_point = std::make_shared<Point>(max_x+1, 0);
  586                 //On l'initialise à une valeur en dehors du quadtree
  587
  588                 if (diff_x > diff_y){
  589                     Polynome* p_eval = forme->evaluer_x(moy_x);
  590                     Polynome* p_deriv = p_eval->deriver_y();
```

Annexe – Code – quadtree.cpp

```
591                     double y = p_eval->trouver_racine_newton_y(nb_iter, moy_y, p_deriv,
precision);
592                     nouv_point->set_x(moy_x);
593                     nouv_point->set_y(y);
594                     delete p_eval;
595                     delete p_deriv;
596                 }
597                 else{
598                     Polynome* p_eval = forme->evaluer_y(moy_y);
599                     Polynome* p_deriv = p_eval->deriver_x();
600                     double x = p_eval->trouver_racine_newton_x(nb_iter, moy_x, p_deriv,
precision);
601                     nouv_point->set_x(x);
602                     nouv_point->set_y(moy_y);
603                     delete p_eval;
604                     delete p_deriv;
605                 }
606 
607                 if (appartient(nouv_point)){
608                     std::shared_ptr<triplet> t = creer_triplet(point, p2, nouv_point);
609                     pile->push_back(t);
610                 }
611             }
612 
613         }
614 
615         //Ajout des nouv_points
616         for (size_t i = 0; i < pile->size(); i++){
617             std::shared_ptr<triplet> t = pile->at(i);
618 
619             lier_triplets(t->p1, t->p2, t->nouv_point);
620             points->push_back(t->nouv_point);
621             point_counter++;
622         }
623         delete pile;
624 
625         this->nettoyer_double(min_x, max_x, min_y, max_y, 10, precision);
```

Annexe – Code – quadtree.cpp

```
626             this->divide_space();
627         }
628     }
629
630     void Quadtree::augmenter_qualite_V2_creation(int nb_iter, double precision, double facteur){
631         if (divided){
632             for (int i = 0; i < 4; i++){
633                 quadtrees[i]->augmenter_qualite_V2_creation(nb_iter, precision, facteur);
634             }
635         }
636         else{
637             int nb_pts = nb_pts_par_quadtree * 1 * facteur;
638             // int nb_pts = point_counter * 4 * facteur;
639
640
641             double x_min = min_x;
642             double x_max = max_x;
643             double y_min = min_y;
644             double y_max = max_y;
645
646             bool non_vide = zone_recherche(x_min, x_max, y_min, y_max);
647             if (non_vide){
648                 int nb_section_doublons = 10; //arbitraire
649                 std::vector<std::shared_ptr<Point>>* non_lies = new
std::vector<std::shared_ptr<Point>>;
650                 Quadtree* q = forme->generer_quadtree_poly(x_max, x_min, y_max, y_min,
651                     nb_iter, nb_pts, nb_pts_par_quadtree, nb_section_doublons, precision,
652                     forme, non_lies);
653                 delete non_lies;
654                 copier(q);
655             }
656
657         }
658     }
659
660     quad_courb* quad_courb_creer(Quadtree* q, double c){
```

Annexe – Code – quadtree.cpp

```
661       quad_courb* qc = (quad_courb*)malloc(sizeof(quad_courb));
662       qc->quad = q;
663       qc->courb = c;
664       return qc;
665   }
666
667
668   void Quadtree::augmenter_qualite_visible(Camera* c, int nb_iter, double precision, int version,
669                                            Quadtree* self, bool qualite,
std::vector<quad_courb*>*
      tableau_quad_courb){
670       double window_left_pos = c->camera_pos_to_world_pos_x(0);
671       double window_up_pos = c->camera_pos_to_world_pos_y(0);
672       double window_right_pos = c->camera_pos_to_world_pos_x(c->get_width());
673       double window_down_pos = c->camera_pos_to_world_pos_y(c->get_height());
674
675       if (divided){
676           if (window_left_pos >= center->x()){
677               if (window_up_pos >= center->y()){
678                   quadtrees[1]->augmenter_qualite_visible(c, nb_iter, precision, version,
      quadtrees[1], qualite, tableau_quad_courb);
679               }
680               else if (window_down_pos <= center->y()){
681                   quadtrees[3]->augmenter_qualite_visible(c, nb_iter, precision, version,
      quadtrees[3], qualite, tableau_quad_courb);
682               }
683               else {
684                   quadtrees[1]->augmenter_qualite_visible(c, nb_iter, precision, version,
      quadtrees[1], qualite, tableau_quad_courb);
685                   quadtrees[3]->augmenter_qualite_visible(c, nb_iter, precision, version,
      quadtrees[3], qualite, tableau_quad_courb);
686               }
687           }
688           else if (window_right_pos <= center->x()){
689               if (window_up_pos >= center->y()){
690                   quadtrees[0]->augmenter_qualite_visible(c, nb_iter, precision, version,
```

Annexe – Code – quadtree.cpp

```
    quadtrees[0], qualite, tableau_quad_courb);
691             }
692             else if (window_down_pos <= center->y()){
693                 quadtrees[2]->augmenter_qualite_visible(c, nb_iter, precision, version,
    quadtrees[2], qualite, tableau_quad_courb);
694             }
695             else {
696                 quadtrees[0]->augmenter_qualite_visible(c, nb_iter, precision, version,
    quadtrees[0], qualite, tableau_quad_courb);
697                 quadtrees[2]->augmenter_qualite_visible(c, nb_iter, precision, version,
    quadtrees[2], qualite, tableau_quad_courb);
698             }
699         }
700         else{
701             quadtrees[0]->augmenter_qualite_visible(c, nb_iter, precision, version, quadtrees[0],
    qualite, tableau_quad_courb);
702             quadtrees[1]->augmenter_qualite_visible(c, nb_iter, precision, version, quadtrees[1],
    qualite, tableau_quad_courb);
703             quadtrees[2]->augmenter_qualite_visible(c, nb_iter, precision, version, quadtrees[2],
    qualite, tableau_quad_courb);
704             quadtrees[3]->augmenter_qualite_visible(c, nb_iter, precision, version, quadtrees[3],
    qualite, tableau_quad_courb);
705         }
706     }
707     else{
708         if (version == 1){
709             this->augmenter_qualite_V1_ajout(nb_iter, precision);
710 
711         }
712         else if (version == 2){
713             if (qualite){
714                 tableau_quad_courb->push_back(quad_courb_creer(self, courbure_moy_quadtree()));
715             }
716             else{
717                 this->augmenter_qualite_V2_creation(nb_iter, precision, 1);
```

Annexe – Code – quadtree.cpp

```
718             }
719         }
720     }
721 }
722
723 double courb(std::shared_ptr<Point> p1, std::shared_ptr<Point> p, std::shared_ptr<Point> p2){
724     //p est le point au milieu
725     double x_v1 = p1->x() - p->x();
726     double y_v1 = p1->y() - p->y();
727
728     double x_v2 = p2->x() - p->x();
729     double y_v2 = p2->y() - p->y();
730
731     double cross = x_v1 * y_v2 - y_v1 * x_v2; // produit vectoriel en 2D
732     double dot = x_v1 * x_v2 + y_v1 * y_v2;
733
734     return std::atan2(std::abs(cross), dot); // angle non orienté
735 }
736
737
738 double Quadtree::courbure_moy_quadtree(){
739     int n = points->size();
740     if (n > 0){
741         double courbure = 0;
742         for (int i = 0; i < n; i++){
743             auto point = points->at(i);
744             if (point->suivants->size() == 2){
745                 if(!point->suivants->at(0).expired() && !point->suivants->at(1).expired()){
746                     courbure += courb(point->suivants->at(0).lock(), point,
    point->suivants->at(1).lock());
747                 }
748             }
749         }
750         return courbure/points->size();
751     }
752     return 0;
753 }
```

Annexe – Code – file.cpp

```
 1  #include "camera.h"
 2  #include "quadtree.h"
 3  #include "polynome.h"
 4
 5  #include <SFML/Graphics.hpp>
 6  #include <GL/glew.h>
 7  #include <GLFW/glfw3.h>
 8
 9  #include <iostream>
10  #include <cstdlib>
11  #include <cmath>
12  #include <vector>
13  #include <random>
14  #include <memory>
15
16
17  const int NB_MINI_PTS = 100;
18  const int NB_MAX_PTS = 500;
19
20  const double FACTEUR_MIN = 0.75;
21  const double FACTEUR_MAX = 2;
22
23
24  double courb_moy(std::vector<quad_courb*>* tableau_quad_courb){
25      double courbure = 0;
26      int nb_pts = 0;
27      for (size_t i = 0; i < tableau_quad_courb->size(); i++){
28          quad_courb* qc = tableau_quad_courb->at(i);
29          courbure += qc->courb * qc->quad->point_counter;
30          nb_pts += qc->quad->point_counter;
31      }
32      if (nb_pts > 0){
33          return courbure/nb_pts;
34      }
35
```

Annexe – Code – file.cpp

```
36      printf("Problème courb_moy()\n");
37      return NAN;
38  }
39
40  double courb_max(std::vector<quad_courb*>* tableau_quad_courb){
41      double courbure_max = 0;
42      for (size_t i = 0; i < tableau_quad_courb->size(); i++){
43          quad_courb* qc = tableau_quad_courb->at(i);
44          if (qc->courb > courbure_max){
45              courbure_max = qc->courb;
46          }
47      }
48      if (courbure_max > 0){
49          return courbure_max;
50      }
51
52      printf("Problème courb_max()\n");
53      return NAN;
54  }
55
56  double facteur(double courbure_moyenne, double courbure, double courbure_max){
57      if (courbure_moyenne == courbure_max){
58          return 1;
59      }
60      if (courbure < courbure_moyenne){
61          return FACTEUR_MIN + courbure * ((1-FACTEUR_MIN)/(courbure_moyenne));
62      }
63      //else
64      return 1 + (courbure - courbure_moyenne) * ((FACTEUR_MAX-1)/(courbure_max-courbure_moyenne));
65  }
66
67  int main(int /*argc*/, char *argv[]) {
68      std::random_device rd; // obtain a random number from hardware
69      std::mt19937 gen(rd()); // seed the generator
70      std::uniform_int_distribution<> distr(0, 255); // define the range
```

Annexe – Code – file.cpp

```
71
72      // Initialize GLFW
73      if (!glfwInit()) {
74          std::cerr << "Failed to initialize GLFW" << std::endl;
75          return -1;
76      }
77
78      // Create a windowed mode window and its OpenGL context
79      int WINDOW_WIDTH = 1000;
80      int WINDOW_HEIGHT = 960;
81      GLFWwindow* window = glfwCreateWindow(WINDOW_WIDTH, WINDOW_HEIGHT, "My Title", NULL, NULL);
82      if (!window) {
83          std::cerr << "Failed to create GLFW window!" << std::endl;
84          glfwTerminate();
85          return -1;
86      }
87
88      // Make the window's context current
89      glfwMakeContextCurrent(window);
90
91      // Initialize GLEW
92      if (glewInit() != GLEW_OK) {
93          std::cerr << "Failed to initialize GLEW!" << std::endl;
94          return -1;
95      }
96
97      // Set the viewport to match the window dimensions
98      glViewport(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT);
99      glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
100
101     // Set up the projection matrix
102     glMatrixMode(GL_PROJECTION);
103     glLoadIdentity();
104     glOrtho(0.0, WINDOW_WIDTH, 0.0, WINDOW_HEIGHT, -1.0, 1.0);
105
```

Annexe – Code – file.cpp

```
106     // Set up the model-view matrix
107     glMatrixMode(GL_MODELVIEW);
108     glLoadIdentity();
109 
110     // Create the camera assigned to the current window
111     Camera* camera = new Camera(window, WINDOW_WIDTH, WINDOW_HEIGHT);
112 
113     Polynome* forme = new Polynome();
114 
115     //Polynome lisse
116     forme->ajouter_terme(std::unique_ptr<Terme>(new Terme(6, 0, 2)));
117     forme->ajouter_terme(std::unique_ptr<Terme>(new Terme(0, 6, 1)));
118     forme->ajouter_terme(std::unique_ptr<Terme>(new Terme(0, 2, 2)));
119     forme->ajouter_terme(std::unique_ptr<Terme>(new Terme(0, 3, 1)));
120     forme->ajouter_terme(std::unique_ptr<Terme>(new Terme(3, 0, 4)));
121     forme->ajouter_terme(std::unique_ptr<Terme>(new Terme(2, 1, -3)));
122     forme->ajouter_terme(std::unique_ptr<Terme>(new Terme(0, 2, -3)));
123 
124     double precision = 10e-7;
125     int nb_pts = std::stoi(argv[3]);
126     int nb_iter = 5;
127     int nb_pts_par_quadtree = std::stoi(argv[4]);
128     int nb_section_doublons = 1 + (int)nb_pts*(3/4);
129 
130     auto non_lies = new std::vector<std::shared_ptr<Point>>;
131     Quadtree* quadtree = forme->generer_quadtree_poly(1.5, -1.5, 1.5, -1.5, nb_iter, nb_pts,
    nb_pts_par_quadtree, nb_section_doublons, precision, forme, non_lies);
132 
133     if (non_lies->size() == 2){
134         non_lies->at(0)->suivants->push_back(non_lies->at(1));
135         non_lies->at(0)->suivants->push_back(non_lies->at(1));
136     }
137     std::cout << "nb_points :" << quadtree->point_counter << "\n";
138     delete non_lies;
139 
```

Annexe – Code – file.cpp

```
140      // Boucle principale
141
142      //render_visible_points a besoin d'un point de départ inutile
143      bool has_last_point = false;
144      Point* last_point = new Point(0,0);
145      int nb_pts_visible = 0;
146      int nb_frames = 0;
147      while (!glfwWindowShouldClose(window)) {
148          nb_frames++;
149          if (nb_frames % 60 == 0){ //on affiche le nombre de points toutes les 2 secondes
150              std::cout << "nb_pts_visible : " << nb_pts_visible << "\n";
151          }
152
153          // Nettoyer l'écran
154          glClear(GL_COLOR_BUFFER_BIT);
155
156          // Rendre les points
157          nb_pts_visible = camera->render_visible_points(quadtree, window, last_point,
    &has_last_point);
158
159          // augmenter_qualite
160          if (nb_pts_visible < NB_MINI_PTS){
161              if (std::stoi(argv[1]) == 1){
162                  quadtree->augmenter_qualite_visible(camera, nb_iter+10, precision*0.1,
    std::stoi(argv[1]), nullptr, std::stoi(argv[2]), nullptr);
163              }
164              else if (std::stoi(argv[1]) == 2){
165                  if (std::stoi(argv[2]) == 1){
166                      auto tableau_quad_courb = new std::vector<quad_courb*>;
167                      quadtree->augmenter_qualite_visible(camera, nb_iter+10, precision*0.1,
    std::stoi(argv[1]), quadtree, std::stoi(argv[2]), tableau_quad_courb);
168                      double courbure_moyenne = courb_moy(tableau_quad_courb);
169                      double courbure_max = courb_max(tableau_quad_courb);
170                      for (size_t i = 0; i < tableau_quad_courb->size(); i++){
171                          quad_courb* qc = tableau_quad_courb->at(i);
```

Annexe – Code – file.cpp

```
172                         qc->quad->augmenter_qualite_V2_creation(nb_iter+10, precision*0.1,
    facteur(courbure_moyenne, qc->courb, courbure_max));
173                     }
174
175                     for (size_t i = 0; i < tableau_quad_courb->size(); i++){
176                         free(tableau_quad_courb->at(i));
177                     }
178                     delete tableau_quad_courb;
179                 }
180                 else{
181                     quadtree->augmenter_qualite_visible(camera, nb_iter+10, precision*0.1,
    std::stoi(argv[1]), quadtree, std::stoi(argv[2]), nullptr);
182                 }
183             }
184         }
185
186         // Swap front and back buffers
187         glfwSwapBuffers(window);
188
189         // Poll for and process events
190         glfwPollEvents();
191     }
192     std::cout << "nb_frames : " << nb_frames << "\n";
193     delete last_point;
194
195     // Clean up and close
196     glfwDestroyWindow(window);
197     glfwTerminate();
198
199     delete camera;
200     delete quadtree;
201     delete forme;
202     return 0;
203 }
```

Annexe – Code – Graphe calcul facteur courbure

```
 1  import matplotlib.pyplot as plt
 2  import pandas as pd
 3  import numpy as np
 4
 5  # Données brutes sous forme de tuples (k, Zoom1, Zoom2, Zoom3)
 6  raw_data = [
 7      (5, 319, 1343, 4079),
 8      (10, 344, 1148, 4173),
 9      (15, 310, 1372, 3043),
10      (20, 215, 775, 3285),
11      (25, 218, 1005, 3295),
12      (30, 367, 710, 2733),
13      (35, 285, 841, 3523),
14      (40, 216, 794, 2351),
15      (40, 260, 1287, 2617),
16      (45, 273, 1346, 2705),
17      (50, 207, 816, 1873),
18      (55, 299, 1095, 2364),
19      (60, 386, 684, 1483),
20      (65, 335, 1324, 1970),
21      (70, 349, 615, 2335),
22      (80, 224, 634, 2889),
23      (90, 237, 696, 1784),
24      (100, 185, 779, 1949),
25      (110, 343, 1258, 2131),
26      (120, 331, 893, 3459),
27      (130, 368, 1307, 3237)
28  ]
29
30  # Création du DataFrame
31  df = pd.DataFrame(raw_data, columns=["k", "Zoom 1", "Zoom 2", "Zoom 3"])
32  df = df.sort_values("k")
33
34  # Création du graphique empilé
35  plt.figure(figsize=(12, 8))
```

Annexe – Code – Graphe calcul facteur courbure

```
36  plt.bar(df["k"], np.array(df["Zoom 1"])*10**(-6), width = 4, label="Zoom 1")
37  plt.bar(df["k"], np.array(df["Zoom 2"])*10**(-6), bottom=np.array(df["Zoom 1"])*10**(-6), width = 4,
    label="Zoom 2")
38  plt.bar(df["k"], np.array(df["Zoom 3"])*10**(-6), bottom=np.array(df["Zoom 1"])*10**(-6) +
   np.array(df["Zoom 2"])*10**(-6), width = 4, label="Zoom 3")
39
40  plt.title("Temps de calculer des facteurs f(c) pour l'ensemble des cellules")
41  plt.xlabel("Valeur de k")
42  plt.ylabel("Temps d'exécution (secondes)")
43  plt.legend()
44  plt.grid(True, axis='y', linestyle='--', alpha=0.7)
45
46  plt.tight_layout()
47  plt.show()
```

Annexe – Code – Temps exécutions 3 versions

```
import numpy as np
# Version 2.0
x_2_0 = [959, 2541, 7209, 20859, 62208]
y_2_0 = np.array([259541, 259541 + 644713, 259541 + 644713 + 1786882,
          259541 + 644713 + 1786882 + 5111019,
          259541 + 644713 + 1786882 + 5111019 + 15058695])/1000000

# Version 2.1
x_2_1 = [1093, 4356, 17365, 72784]
y_2_1 = np.array([466578, 466578 + 1743199,
          466578 + 1743199 + 6721697,
          466578 + 1743199 + 6721697 + 27607691])/1000000

# Version 1.0
x_1_0 = [667, 1303, 1912, 2541, 3219, 3959, 4782, 5668, 6661, 7776,
          9140, 10723, 12543, 14632, 16986, 19552, 22350, 25493, 28813,
          32822, 37272]

y_1_0 = []
temps = np.array([23740, 39523, 61765, 83667, 109023, 136374, 161678, 200029,
          235520, 284958, 343469, 410881, 492346, 580365, 675122,
          770734, 854842, 928509, 1001166, 1103293, 1232776])/1000000
cumul = 0
for t in temps:
    cumul += t
    y_1_0.append(cumul)


import matplotlib.pyplot as plt

# Tracer les courbes
plt.plot(x_2_0, y_2_0, marker='o', label='Recréation sans optimisation')
plt.plot(x_2_1, y_2_1, marker='s', label='Recréation avec optimisation')
plt.plot(x_1_0, y_1_0, marker='^', label='Ajout de points')

```

Annexe – Code – Temps exécutions 3 versions

```
36  # Ajouter les labels et le titre
37  plt.xlabel("Nombre de points à l'écran()")
38  plt.ylabel("Temps cumulé (secondes)")
39  plt.title("Évolution du temps d'exécution par version")
40  plt.legend()
41  plt.grid(True)
42  plt.tight_layout()
43
44  # Afficher le graphique
45  plt.show()
```

Annexe – Code – Temps exéc création quadtree

```
 1  import matplotlib.pyplot as plt
 2  import numpy as np
 3  import pandas as pd
 4
 5  # Données
 6  data = [
 7      [5, 317, 816, 0, 43],
 8      [10, 761, 532, 13, 72],
 9      [15, 2121, 686, 18, 224],
10      [20, 3181, 821, 26, 290],
11      [30, 6541, 1833, 60, 428],
12      [40, 13467, 3245, 95, 628],
13      [50, 19256, 3310, 163, 472],
14      [60, 27385, 5238, 209, 538],
15      [80, 44272, 11211, 204, 931],
16      [100, 67358, 21959, 464, 960],
17      [120, 96946, 34979, 451, 1141],
18      [140, 130364, 55316, 321, 1377],
19      [160, 168559, 83139, 364, 1560],
20      [200, 267842, 165715, 456, 2017],
21      [240, 378675, 266235, 603, 2497],
22      [280, 518957, 420273, 786, 2817],
23      [320, 668071, 628917, 1167, 3264],
24      [360, 841317, 896895, 1261, 3773],
25      [400, 1033045, 1221015, 1297, 4062],
26      [440, 1247638, 1624191, 1530, 4541],
27      [480, 1485175, 2105292, 1705, 5350],
28      [520, 1747740, 2675516, 1902, 5580],
29      [560, 2028434, 3363632, 2143, 6341],
30      [600, 2328364, 4106677, 2317, 6160],
31      [640, 2648760, 4985117, 2374, 7099],
32      [680, 2983153, 5978095, 2577, 7966],
33      [720, 3344892, 7092642, 2709, 8459],
34      [760, 3725723, 8318046, 2964, 9645],
35      [800, 4152817, 9688634, 2980, 10354],
```

Annexe – Code – Temps exéc création quadtree

```
36      [840, 4602174, 11307325, 4272, 14515],
37      [880, 5072571, 13482467, 3922, 11546],
38      [920, 5457076, 15135374, 3551, 13494],
39      [960, 5918432, 16820830, 3730, 17169],
40      [1000, 6460320, 19301085, 4001, 18971],
41  ]
42
43  # Création du DataFrame
44  df = pd.DataFrame(data, columns=["n", "Créer", "Nettoyer", "Diviser", "Relier"])
45
46  # Calcul du temps total en secondes
47  df["Total_s"] = (df["Créer"] + df["Nettoyer"] + df["Diviser"] + df["Relier"]) / 1e6
48
49  # Tracé colonnes empilées
50  bar_width = 25
51  fig, ax = plt.subplots(figsize=(14, 7))
52
53  # Positions des barres - centrées
54  positions = np.arange(len(df)) * (bar_width + 5)
55
56  # Empilement des barres
57  ax.bar(positions, df["Créer"]/1e6, width=bar_width, label="Créer les points")
58  ax.bar(positions, df["Nettoyer"]/1e6, width=bar_width, bottom=df["Créer"]/1e6, label="Nettoyer les
    doublons")
59  bottom2 = (df["Créer"] + df["Nettoyer"])/1e6
60  ax.bar(positions, df["Diviser"]/1e6, width=bar_width, bottom=bottom2, label="Diviser l'espace")
61  bottom3 = bottom2 + df["Diviser"]/1e6
62  ax.bar(positions, df["Relier"]/1e6, width=bar_width, bottom=bottom3, label="Relier les points")
63
64  # Ajustements
65  ax.set_xticks(positions)
66  ax.set_xticklabels(df["n"])
67  ax.set_xlabel("n")
68  ax.set_ylabel("Temps (secondes)")
69  ax.set_title("Temps par étape (barres empilées)")
```

Annexe – Code – Temps exéc création quadtree

```
70  ax.legend()
71  ax.grid(True, axis='y')
72
73  plt.tight_layout()
74  plt.show()
```

Annexe – Code – Temps exéc en fonction de k

```
 1  import matplotlib.pyplot as plt
 2
 3  # Données initiales
 4  x = [10, 20, 50, 100, 200, 500, 700, 1000, 1500]
 5  y_microsec = [2821669, 2799708, 2809713, 2832789, 2860385, 3014369, 2993573, 3004664, 3849849]
 6
 7  # Conversion en secondes
 8  y_sec = [t / 1_000_000 for t in y_microsec]
 9
10  x = x
11  y_sec = y_sec
12
13  plt.figure(figsize=(8,5))
14  plt.plot(x, y_sec, marker='o', linestyle='-', color='b')
15  plt.plot(0, 0)
16
17  plt.title("Temps d'exécution en fonction du nombre maximum de points par quadtree (k)")
18  plt.xlabel("k")
19  plt.ylabel("Temps d'exécution (secondes)")
20
21
22  plt.grid(True)
23  plt.show()
```

Annexe – Code – Graphes algorithme de Crust

```
 1  import numpy as np
 2  import matplotlib.pyplot as plt
 3  from scipy.spatial import Voronoi, Delaunay, voronoi_plot_2d
 4  from itertools import combinations
 5  import numpy as np
 6
 7  n_points = 40
 8
 9  # Courbe principale (rayon ≈ 5)
10  theta = np.linspace(0, 2 * np.pi, n_points, endpoint=False)
11
12  # Courbe extérieure (grande)
13  r_outer = 5 + np.sin(3 * theta) + 0.3 * np.random.rand(n_points)
14  x_outer = r_outer * np.cos(theta)
15  y_outer = r_outer * np.sin(theta)
16
17  # Courbe intermédiaire (proche de l'extérieure)
18  r_middle = r_outer*0.9
19  x_middle = r_middle * np.cos(theta)
20  y_middle = r_middle * np.sin(theta)
21
22  # Courbe intérieure (proche aussi)
23  r_inner = r_outer*0.8
24  x_inner = r_inner * np.cos(theta)
25  y_inner = r_inner * np.sin(theta)
26
27  # Concaténer tous les points
28  x_all = np.concatenate([x_outer, x_middle, x_inner])
29  y_all = np.concatenate([y_outer, y_middle, y_inner])
30  points = np.column_stack((x_all, y_all))
31
32  # Étape 1 : Diagramme de Voronoi
33  vor = Voronoi(points)
34
35  # Visualisation Étape 1 : Diagramme de Voronoi
```

Annexe – Code – Graphes algorithme de Crust

```
36  fig, ax = plt.subplots(figsize=(8, 8))
37  ax.plot(points[:, 0], points[:, 1], 'ko', label='Points d\'échantillonnage')
38  voronoi_plot_2d(vor, ax=ax, show_vertices=False, line_colors='orange', line_width=2)
39
40  ax.set_title("Étape 1 : Diagramme de Voronoi des points")
41  ax.legend(loc='upper right')
42  ax.set_aspect('equal')
43  ax.set_xlim(-10, 10)
44  ax.set_ylim(-10, 10)
45
46  plt.grid(True)
47  plt.show()
48
49
50  # Étape 2 : Calcul des pôles p+ et p-
51  def find_poles(points, vor):
52      poles_plus = []
53      poles_minus = []
54
55      for i, s in enumerate(points):
56          region_index = vor.point_region[i]
57          region = vor.regions[region_index]
58          if -1 in region or len(region) == 0:
59              continue
60          vor_vertices = vor.vertices[region]
61          vectors = vor_vertices - s
62
63          dists = np.linalg.norm(vectors, axis=1)
64          idx_plus = np.argmax(dists)
65          p_plus = vor_vertices[idx_plus]
66          poles_plus.append(p_plus)
67
68          dot_products = np.dot(vectors, vectors[idx_plus])
69          dot_products[idx_plus] = 1  # pour exclure p_plus
70          idx_minus = np.argmin(dot_products)
```

Annexe – Code – Graphes algorithme de Crust

```
71          p_minus = vor_vertices[idx_minus]
72          poles_minus.append(p_minus)
73
74      return np.array(poles_plus), np.array(poles_minus)
75
76  poles_plus, poles_minus = find_poles(points, vor)
77
78  # Visualisation Étape 2 : Pôles p+ et p-
79  fig, ax = plt.subplots(figsize=(8, 8))
80  ax.plot(points[:, 0], points[:, 1], 'ko', label='Points d\'échantillonnage')
81  ax.plot(poles_plus[:, 0], poles_plus[:, 1], 'b^', label='Pôles p⁺')
82  ax.plot(poles_minus[:, 0], poles_minus[:, 1], 'gv', label='Pôles p⁻')
83
84  # Lignes reliant chaque point à ses pôles
85  for i, p in enumerate(points):
86      if i < len(poles_plus):
87          ax.plot([p[0], poles_plus[i, 0]], [p[1], poles_plus[i, 1]], 'b--', alpha=0.5)
88          ax.plot([p[0], poles_minus[i, 0]], [p[1], poles_minus[i, 1]], 'g--', alpha=0.5)
89
90
91  ax.set_title("Étape 2 : Calcul des pôles p⁺ et p⁻ pour chaque point")
92  ax.legend(loc='upper right')
93  ax.set_aspect('equal')
94  ax.set_xlim(-10, 10)
95  ax.set_ylim(-10, 10)
96
97  plt.grid(True)
98  plt.show()
99
100 # Étape 3 : Triangulation de Delaunay sur S ∪ P
101 union_points = np.vstack([points, poles_plus, poles_minus])
102 delaunay_all = Delaunay(union_points)
103
104 fig, ax = plt.subplots(figsize=(8, 8))
105 ax.triplot(union_points[:, 0], union_points[:, 1], delaunay_all.simplices, color='gray',
```

Annexe – Code – Graphes algorithme de Crust

```
    linewidth=0.8)
106  ax.plot(points[:, 0], points[:, 1], 'ko', label='Points d\'échantillonnage')
107  ax.plot(poles_plus[:, 0], poles_plus[:, 1], 'b^', label='Pôles p⁺')
108  ax.plot(poles_minus[:, 0], poles_minus[:, 1], 'gv', label='Pôles p⁻')
109
110  ax.set_title("Étape 3 : Triangulation de Delaunay sur Points ∪ Pôles")
111  ax.legend(loc='upper right')
112  ax.set_aspect('equal')
113  ax.set_xlim(-10, 10)
114  ax.set_ylim(-10, 10)
115
116  plt.grid(True)
117  plt.show()
118
119  # Étape 4
120
121  # Nombre de points originaux
122  n_original = len(points)
123
124  # Obtenir les triangles de la triangulation complète
125  simplices = delaunay_all.simplices
126
127  # Filtrer les arêtes où les deux sommets sont dans l'ensemble initial (indices < n_original)
128  crust_edges = set()
129  for triangle in simplices:
130      for i, j in combinations(triangle, 2):
131          if i < n_original and j < n_original:
132              crust_edges.add(tuple(sorted((i, j))))
133
134  # Tracer le résultat final : les arêtes Crust
135  fig, ax = plt.subplots(figsize=(8, 8))
136
137  # Tracer les arêtes filtrées
138  for i, j in crust_edges:
139      p1, p2 = points[i], points[j]
```

Annexe – Code – Graphes algorithme de Crust

```
140      ax.plot([p1[0], p2[0]], [p1[1], p2[1]], 'm-', linewidth=2)
141
142  # Tracer tous les points et pôles pour référence
143  ax.plot(points[:, 0], points[:, 1], 'ko', label='Points d\'échantillonnage')
144
145  ax.set_title("Étape 4 : Crust – Arêtes filtrées de Delaunay(S ∪ P)")
146  ax.legend(loc='upper right')
147  ax.set_aspect('equal')
148  ax.set_xlim(-10, 10)
149  ax.set_ylim(-10, 10)
150
151  plt.grid(True)
152  plt.show()
```

Annexe – Code – Graphes Voronoï Delaunay

```
 1  import matplotlib.pyplot as plt
 2  import numpy as np
 3  from scipy.spatial import Voronoi, Delaunay
 4
 5
 6  points = np.random.rand(10, 2)
 7  vor = Voronoi(points)
 8  delaunay = Delaunay(points)
 9
10  def plot_voronoi_with_infinite_ridges(ax, vor, color='gray'):
11      center = vor.points.mean(axis=0)
12      radius = np.ptp(vor.points, axis=0).max() * 2
13
14      # Arêtes finies
15      for simplex in vor.ridge_vertices:
16          simplex = np.asarray(simplex)
17          if np.all(simplex >= 0):
18              ax.plot(vor.vertices[simplex, 0], vor.vertices[simplex, 1], color=color)
19
20      # Arêtes infinies
21      for pointidx, simplex in zip(vor.ridge_points, vor.ridge_vertices):
22          simplex = np.asarray(simplex)
23          if np.any(simplex < 0):
24              i = simplex[simplex >= 0][0]
25              t = vor.points[pointidx[1]] - vor.points[pointidx[0]]
26              t = t / np.linalg.norm(t)
27              n = np.array([-t[1], t[0]])
28              midpoint = vor.points[pointidx].mean(axis=0)
29              direction = np.sign(np.dot(midpoint - center, n)) * n
30              far_point = vor.vertices[i] + direction * radius
31              ax.plot([vor.vertices[i, 0], far_point[0]],
32                      [vor.vertices[i, 1], far_point[1]], color=color)
33
34
35  fig, axs = plt.subplots(1, 2, figsize=(12, 6))
```

Annexe – Code – Graphes Voronoï Delaunay

```
36
37  # Voronoi seul
38  axs[0].set_title("Diagramme de Voronoi")
39  axs[0].scatter(points[:, 0], points[:, 1], s=100, color='black')
40  plot_voronoi_with_infinite_ridges(axs[0], vor, color='gray')
41  axs[0].set_xlim(0, 1)
42  axs[0].set_ylim(0, 1)
43  axs[0].set_aspect('equal')
44
45  # Delaunay + Voronoi
46  axs[1].set_title("Triangulation de Delaunay")
47  plot_voronoi_with_infinite_ridges(axs[1], vor, color='gray')
48  axs[1].triplot(points[:, 0], points[:, 1], delaunay.simplices, color='black')
49  axs[1].scatter(points[:, 0], points[:, 1], s=100, color='black')
50  axs[1].set_xlim(0, 1)
51  axs[1].set_ylim(0, 1)
52  axs[1].set_aspect('equal')
53
54  plt.tight_layout()
55  plt.show()
```