Grenoble INP - Ensimag

Projet d'algorithmique de polygones

Autors:
Mehdi El Idrissi El Fatmi
Saad El Hmaidi

Grenoble, April 7, 2024.

Contents

1 Introduction

1.1 Problématique et cahier de charge

Notre programme doit prendre en entrée un fichier *.txt* ou *.poly* contenant un nombre N de polygones, chacun représenté par une liste de points. On notera $\bar{n}$ le nombre moyen de points dans un polygone : $\bar{n} = \frac{\sum n_i}{N}$

Notre objectif est de mettre en place un programme qui à partir de ce fichier donne en sortie la liste des inclusions des polygones : Ainsi pour réussir le projet :

- Le programme doit être robuste et fonctionner sur tout type de polygones et tout type de cas spéciaux
- Le programme doit avoir un temps d'exécution optimal
- Le programme doit être lisible et réutilisable

1.2 Méthode du rayon

Il s'agit du bloc fonctionnel le plus important du projet, un algorithme pour vérifier si un point est à l'intérieur d'un polygone avec une complexité linéaire $O(n)$ avec n le nombre de points du polygone. → Pour trouver si un point p est à l'intérieur d'un polygone P il suffit d'envoyer un rayon dans n'importe quelle direction et de compter le nombre de segments du polygone coupés par le rayon :

- Si le nombre est pair : p est à l'extérieur de P
- Si le nombre est impair : p est à l'intérieur de P

Pour notre cas nous allons envoyer un rayon horizontal à droite, il y a deux cas à distinguer:

- Le cas où le rayon traverse les segments au milieu (càd ne touche pas les extrémités du segments mais traverse entre les deux)
  → Dans ce cas on compte le segment
- Le cas ou le rayon traverse un segment sur l'un de ses extrémités :
  → Dans ce cas on compte seulement les segments en haut du point d'intersection.
- Si le segment est horizontal, ne rien faire

2 Suivi linéaire de notre projet

Ce projet se divise en plusieurs couches fonctionnelles, dont l'implémentation est indépendante, c'est à dire que l'une peut être changée ou optimisée sans influencer l'autre.
Ainsi pour avoir un code modulaire et réutilisable on va découper ce projet en des couches fonctionnelles indépendantes :

- **Couches de prétraitement des polygones:** Tout ce qui concerne le traitement des données avant l'exécution des algorithmes d'inclusions et de recherche.
- **Couche de recherche de polygones**: étant donné un polygone P_i cette couche s'occupe d'en chercher le conteneur parmi les autres polygones en réduisant le domaine de recherche, elle est implémentée pour éviter du brut-force sur tous les polygones.
- **Couche point dans polygone:** Il s'agit de la couche la plus fondamentale et s'occupe de vérifier si un point p appartient à un polygone P_i

2.1 Algorithme du rayon V1

On dispose d'un point p, et d'un polygone P, pour compter le nombre de segments coupés par le rayon horizontal envoyé de p, pour l'instant nous n'avons d'autre choix que de faire une boucle sur les segments du polygone et vérifier à chaque fois s'il y'a une intersection avec le rayon.

Algorithme d'intersection dans le cas ou le rayon coupe le segment entre ses extrémités : Soit p un point de coordonnées x,y : $p = x\vec{e_x} + y\vec{e_y}$, et un segment du polygone d'extrémités p_1, p_2 :

- Les points du segment s'écrivent sous la forme : $p_\lambda = p_1 + \lambda(p_2 - p_1)$ avec $0 \leq \lambda \leq 1$
- Les points du rayon sortant de p à droite s'écrivent sous la forme : $p_t = p + t\vec{e_x}$ avec $t > 0$

Quand il y a intersection on a :

$$p_t = p_\lambda \implies \begin{bmatrix} \lambda = \frac{y_2 - y1}{y - y_1} \\ \\ t = \frac{(y_2 - y1)(x_2 - x_1)}{y - y_1} + x_1 - x \end{bmatrix}$$

L'intersection est valable si et seulement si $0 \leq \lambda \leq 1$ et $t > 0$:

$$\begin{bmatrix} min(y_1, y_2) \leq y \leq max(y_1, y_2) \\ \\ (x - x_1)(y - y_1)sign(y_2 - y_1) \geq (y_2 - y_1)(x_2 - x_1) \end{bmatrix}$$

Remarques :

- On remarque qu'on a pas directement comparé λ à 0 et à 1, car cela s'avère coûteux de diviser sur un flottant si $y - y_1$ atteint de très faible valeurs (polygones proches).

- On remarque la même chose, pour la deuxième inégalité, on a pas comparé t directement à 0.

Algorithme 1 point_dans_polygone (p, polygones, i)
p : Point, polygones : liste ou pointeur, i : int

```
compteur = 0
pour chaque segment dans polygones[i] faire        ▷ O(n) avec n nombre de points du polygone i
    y_min = min(y1, y2)
    y_max = max(y1, y2)
    si y_min ≠ y_max alors
        si y1 ≤ y ≤ y2 et (x − x1)(y − y1) ≥ (y2 − y1)(x2 − x1) alors
            incrémenter cpt
        sinon si y2 ≤ y ≤ y1 et (x − x1)(y − y1) ≤ (y2 − y1)(x2 − x1) alors
            incrémenter cpt
        sinon si y = y1 et x ≤ x1 et y2 ≤ y alors
            incrémenter cpt
        sinon si y = y2 et x ≤ x2 et y1 ≤ y alors
            incrémenter cpt
        fin si
    fin si
fin pour
si compteur impair alors
    retourne Vrai
fin si
```

Notes :

- l'algorithme point_dans_polygone a une complexité linéaire $O(n)$
- Si on avait mis le polygone P_i comme paramètre, par exemple : point_dans_polygone(p, P_i), on aurait copié le polygone P_i avec tous ses points, dont le nombre peut arriver à des milliers de points, ceci dit avec notre approche, on a mit la liste des polygones comme paramètre, qui est juste un pointeur et sa copie est beaucoup moins coûteuse.

2.2 Approche Naïve

Remarque Importante: Dans le Readme il était mntionné que les polygones ne s'intersectent pas entre eux, de cela nous avions compris qu'il n'avait pas d'intersection mais que peut être il pouvait avoir des points en commun :
Ainsi pour voir si un polygone P_i est inclus dans un autre polygone Pj il faudra faire notre algorithme point_dans_polygone sur tous les points du polygone
→ Ce cas est aussi intéressant à traiter car dans le cas de polygones infiniment proches l'algorithme peut se tromper s'il ne fait pas de boucle sur tous les points, voici des cas ou il se trompera :

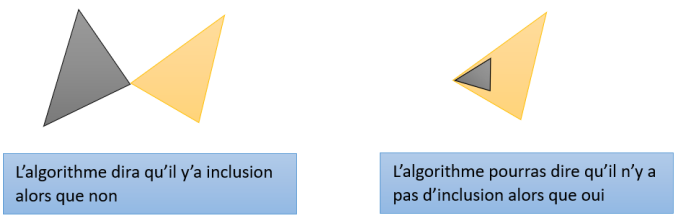

Figure 1: Cas spéciaux que notre algorithme traite bien

→ **Complexité moyenne** pour vérifier si P_i inclus dans P_j : $O(\bar{n}^2)$ avec $\bar{n}$ le nombre de points des polygones

→ **Complexité moyenne** pour trouver les polygones contenant P_i : $O(Nn^2)$ avec N le nombre de polygones de la liste

Ainsi pour trouver les polygones inclus dans P_i il nous faudra boucler sur la liste de polygones toute entière, ce qui est inacceptable ! Nous avons alors pensé à l'algorithme suivant qui va pouvoir amortir le N en N_a :

Algorithme 2 trouver_toutes_inclusions_polygone_i (polygones, i)
polygones : liste ou pointeur, i : int

```
//On utilise une liste d'indices pour réduire la compléxité et éviter copier des polygones
indices_polygones_possibles = [1, ...N]          ▷ Initialiser la liste des indices possibles
indices_polygones_possibles_suiv = []
pour chaque Point p dans polygones[i] faire                                   ▷ O(n)
    pour chaque j dans indices_polygones_possibles faire          ▷ Amortissement O(N_a)
        si point_dans_polygone(p, polygones, i) est Vrai alors                ▷ O(n)
            Ajouter j à index_polygones_possibles_suiv
        fin si
    fin pour
    //Réduire la liste en la mettant à jour
    indices_polygones_possibles ← indices_polygones_possibles_suiv   ▷ Mettre à jour la liste
fin pour
retourne indices_polygones_possibles → Complexité de l'algorithme : O(N_a n^2)
```

Donc pour trouver le polygone d'inclusion de P_j, il faut faire une boucle et prendre le polygone avec l'aire minimale :

Si on considère que notre algorithme crée des polygones l'un après l'autre avec une probabilité d'inclusion p_{inc}

- Dans le meilleur des cas $p_{inc} = 0$:
  On aura seulement un polygone contenant P_j → Complexité moyenne $O(1)$
- Dans le pire des cas $p_{inc} = 1$
  Tous les polygones se contiendront les un les autres, Complexité moyenne $O(\frac{\sum N-k}{N}) = O(\frac{N+1}{2})$

Donc la complexité finale pour trouver l'indice d'inclusion d'un polygone P_i est de l'ordre de $O(\frac{N_a(N+1)n^2}{2})$

2.3 Ordonner les polygones par Surfaces Croissantes

L'algorithme précédent recherchait des polygones selon l'ordre prédéterminé de la liste polygones, ce qui n'a pas de sens, car nous devons réduire au maximum le domaine de recherche.

Ordonner les polygones selon la surface permet alors de ne prendre en considération que les polygones qui sont plus grands que notre polygone cible, par exemple :
→ Étant donné un polygone P_i, ayant un indice i dans la liste ordonnée des polygones sorted_polygones, il suffit de voir les polygones de $i+1$ jusqu'à N.
→ On peut réduire la recherche encore plus en créant aussi une liste des limites de quadrants de chaque polygone, elle sera quadri-dimensionnelle et permettra de voir si un polygone est inclus dans le quadrant d'un autre polygone avec une complexité constante.

3 Touche Finale

Notre programme n'ayant pas réalisé le classement attendu, surtout dans le Test-0, nous avons fait l'hypothèse que le prof utilisait des polygones de grands nombres de points, (de l'ordre des centaines de points), car notre algorithme était optimisé pour des polygones normaux, de 4 ou 5 points.

Ainsi pour pouvoir pallier à la **lenteur du programme pour les gros polygones**, nous avons repensé l'algorithme point_dans_polygone.

Comment pourra-on alors réduire la complexité de cette algorithme ?
Nous savons par notre incorporation du cas spécial où les polygones ont des points en commun, il faut vérifier que tous les points d'un polygone P_i sont à l'intérieur d'un polygone P_j, ainsi il faut faire tourner l'algorithme point_dans_polygone n fois, *alors qu'on imagine que les autres binomes n'ont pas eu besoin de le tourner n fois car les tests n'ont pas montré ce cas.*
On travaille sur un algorithme intrinsèquement plus gros n fois (on ne savait pas cela , on croyait que tout le monde avait incorporé ce cas spécial et qu'il figurait dans les tests).

La question qui s'impose est si l'on peut faire une meilleur complexité que $O(n)$ dans l'algorithme point_dans_polygone :

- Autrement dit est ce qu'on peut éviter de regarder l'intersection du rayon avec **tous les segments** et voir juste ceux qui nous intéressent.

- **Idée !! :** Durant le pré-traitement on s'occupe de **partitionner chaque polygone en des Morceaux selon l'axe y**, par exemple pour le polygone dans la figure 1 :

$$Polygone \underset{partion}{\rightarrow} \{[Y_1, Y_2], [Y_2, Y_3], ...[Y_{N-1}, Y_N]\}$$

  - Ensuite on stocke dans chaque morceau $[Y_k, Y_{k+1}]$ la liste des segments qui le coupent par exemple dans la figure:

$$[Y_4, Y_5] \underset{segments}{\rightarrow} \{3, 6, 10\}$$

  - Dans le code on stocke $[Y_4, Y_5] \underset{segments}{\rightarrow} \{3, 6, 10\}$ par une liste bi-dimensionnelle, et on identifie $[Y_4, Y_5]$ à Y_4 la coordonnée la plus petite.

$$\rightarrow [Y_4, [Seg_3, Seg_6, Seg_{10}]]$$

  - On ne va pas compter les segments vers le bas dans les Morceaux, dans notre algorithme point_dans_polygone on ne compte que les segments vers le haut.

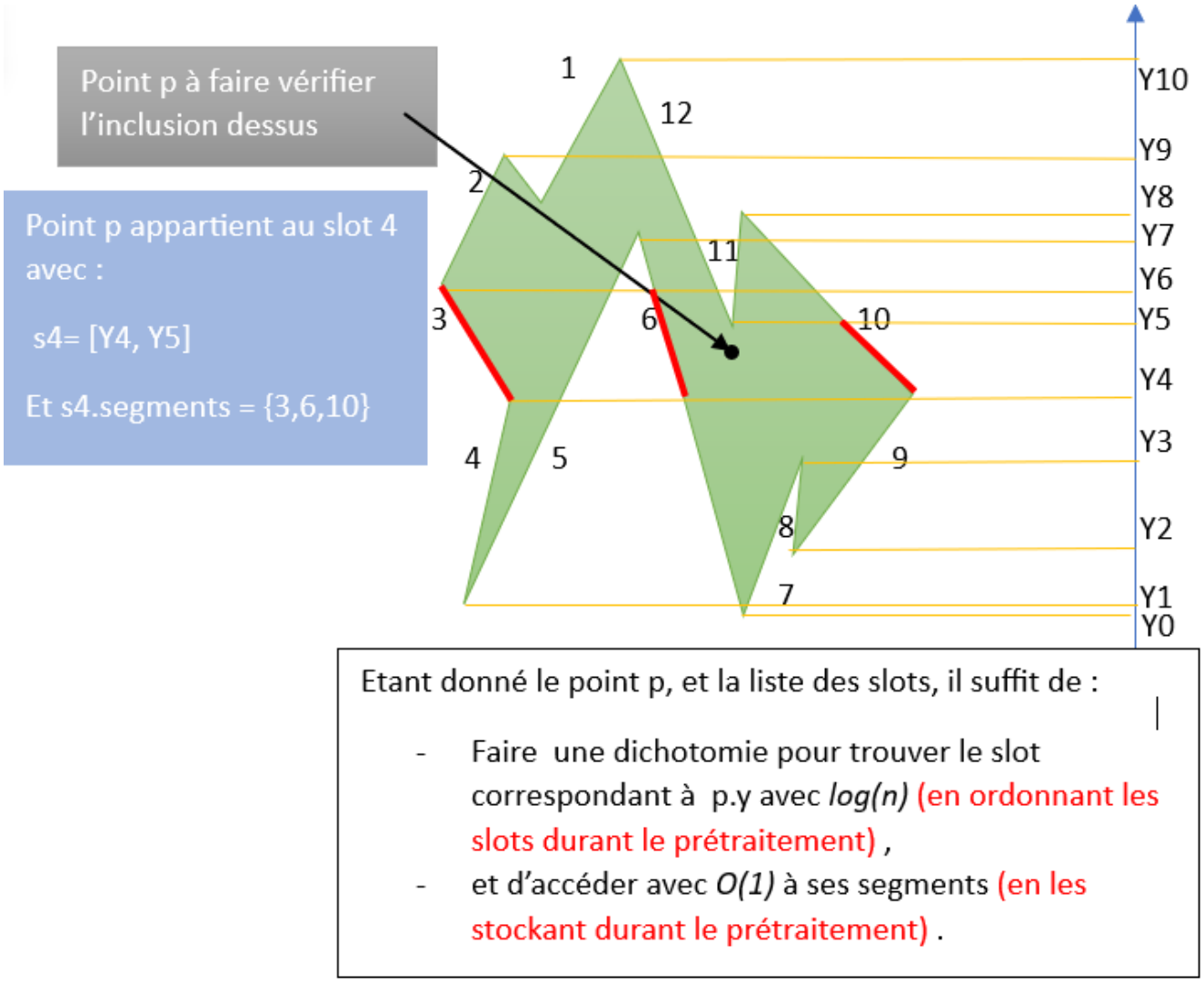

Figure 2: Représentation du partitionnement du partitionnement

Illustration avec terminology :

3.1 Pseudo-code pour le pré-traitement des points d'un polygone

Algorithme 3 Pretraitement(Points)
Points : liste ou pointeur

```
//Trouver les Morceaux Y_k
tab_Y_s_passé = {}                                   ▷ Dictionnaire vide
Morceaux = []                                        ▷ Sera une liste multidimensionnelle
pour chaque p de Points faire                        ▷ O(n)
    si p n'existe pas dans tab_Y_s_passé alors       ▷ O(1) car tab_y_s_passé est un dictionnaire
        ajouter [p.y, []] à Morceaux                 ▷ La liste vide sera pour stocker les segments
    fin si
fin pour
//Ordonner les morceaux et créer un dictionnaire pour trouver les indices de la liste à partir des éléments Y_k
morceau_to_indice={}
//triFusionReversible est un tri amélioré pour garder un dictionnaire capable de trouver les indices à partir des valeurs
triFusionReversible(Morceaux,morceau_to_indice)      ▷ O(nlog(n))
//Remplir les morceaux des segments correspondant
pour chaque segment non horizontal Seg de Points faire   ▷ O(n)
    trouver y_min et y_max les coordonnées verticales limites du segment
    i_min=morceau_to_indice[y_min]                   ▷ Indice minimum dans la liste des indices
    i_max=morceau_to_indice[y_max]
    pour chaque i entre i_min et i_max faire         ▷ Complexité O(n?) avec O(n) pire complexité
        ajouter Seg à Morceaux[i][1]
    fin pour
fin pour
retourne Morceaux → Complexité de l'algorithme : O(n + nlog(n) + nn?)
```

Algorithme 4 point_dans_polygone_v2 (p, polygones, i)
p : Point, polygones : liste ou pointeur, i : int
On utilisera Morceaux_all qui est une variable et point_dans_polygone peut y accéder

```
compteur = 0
indice_morceau = dichotomyMod(p.y,Morceaux_all[i])   ▷ O(log(n))
pour chaque segment dans Morceaux_all[i][indice_morceau] faire   ▷ O(n_amortie)
    y_min = min(y_1, y_2)
    y_max = max(y_1, y_2)
    si y_min ≠ y_max alors
        si y_1 ≤ y ≤ y_2 et (x − x_1)(y − y_1) ≥ (y_2 − y_1)(x_2 − x_1) alors
            incrémenter cpt
        sinon si y_2 ≤ y ≤ y_1 et (x − x_1)(y − y_1) ≤ (y_2 − y_1)(x_2 − x_1) alors
            incrémenter cpt
        sinon si y = y_1 et x ≤ x_1 alors
            incrémenter cpt
        sinon si y = y_2 et x ≤ x_2 alors
            incrémenter cpt
        fin si
    fin si
fin pour
si compteur impair alors
    retourne Vrai → Complexité de l'algorithme : O(n_a log(n)) ou O(log(n))
fin si
```

3.2 Structure du code et calcul de compléxité pour notre dernière version :

4 Tests et Génération des polygones

4.1 Algorithme

On a mis en place un algorithme capable de créer des polygones aléatoirement:
Pour créer ces polygones et être sûr de ne pas se tromper dans les inclusions et les intersections on a ajouté une classe héréditaire de Polygon appelée OriPolygon où on ajoute :

- un attribut origine : Point
- un attribut rayon_minimal représentant la distance minimale par rapport à l'origine
- un attribut rayon_maximal représentant la distance maximale par rapport à l'origine

Cela nous permettra de créer

- des carrés,
- des triangles,
- des cercles,
- **des polygones de forme aléatoire avec une probabilité de rayon uniforme selon l'algorithme suivant**
  A partir d'un point origine, et d'une référence d'angle parcourir les angles de 0 à 2π
  - mettre un point dans cet angle avec une distance r de l'origine, avec r suivant une loi uniforme entre deux limites. → Par exemple si on veut inclure un polygone aléatoire dans un cercle de rayon a, on peut envisager que r suive une loi uniforme entre 0.5a et 0.9a. Évitant ainsi toute intersection.

Après chaque polygone créé, on a deux cas :

- Soit on crée un autre polygone à l'intérieur de celui-ci (càd à l'intérieur du dernier polygone créé)
- Soit on avance avec un pas aléatoire selon une probabilité uniforme dans l'espace.
- ces deux évènements sont contrôlés par une probabilité dite d'inclusion p_inc

4.2 Tests

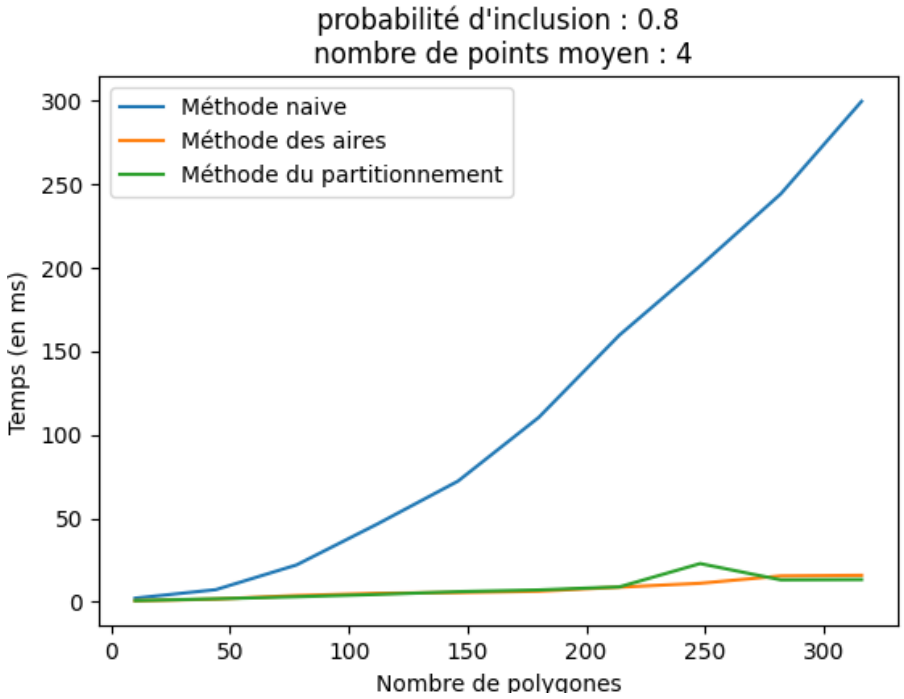

Figure 3: Variation du temps d'exécution en fonction du nombre de polygones pour les 3 méthodes

On voit bien que la méthode 1 naïve diverge vers 50 polygones et varie de façon exponentielle.

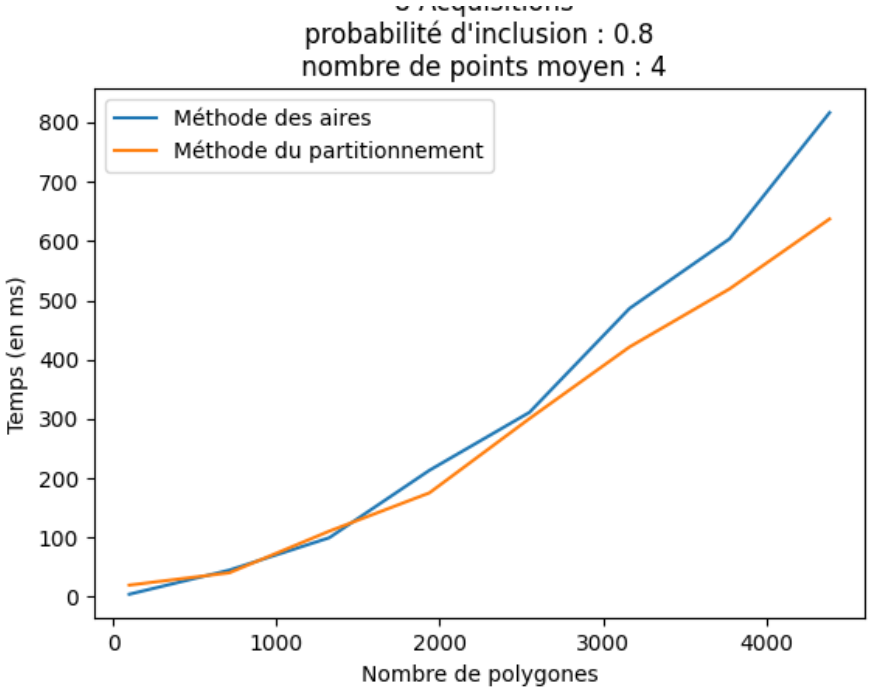

Figure 4: Variation du temps d'exécution en fonction du nombre de polygones

On voit que l'algorithme de partitionnement n'a d'effet qu'au delà des 3000 polygones, sans oublier que le nombre de points ici est faible (4 points en moyenne par polygone), alors que normalement la différence entre les deux algorithmes doit se marquer avec un grand nombre moyen de point. On voit aussi que les 2 méthodes varient de façon linéaire apparente.

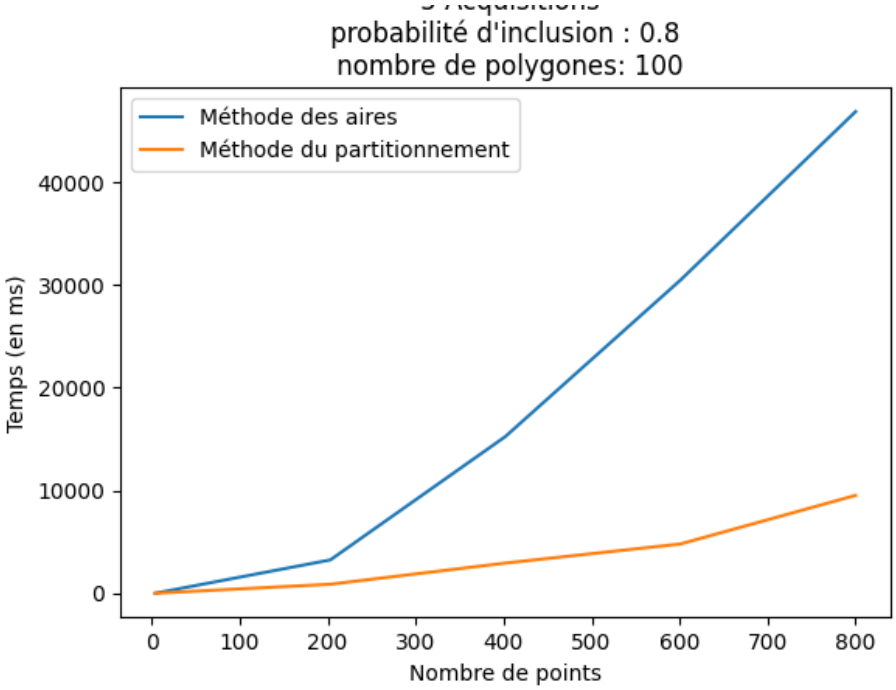

Figure 5: Variation du temps d'exécution en fonction du nombre de points

On varie ici le nombre moyen de points pour voir la différence entre les deux algorithmes, on voit bien alors pour 100 polygones que l'algorithme des surfaces et quadrants seul commence à diverger vers 200 points et que l'algorithme des morceaux, ou du partitionnement est bien meilleur.

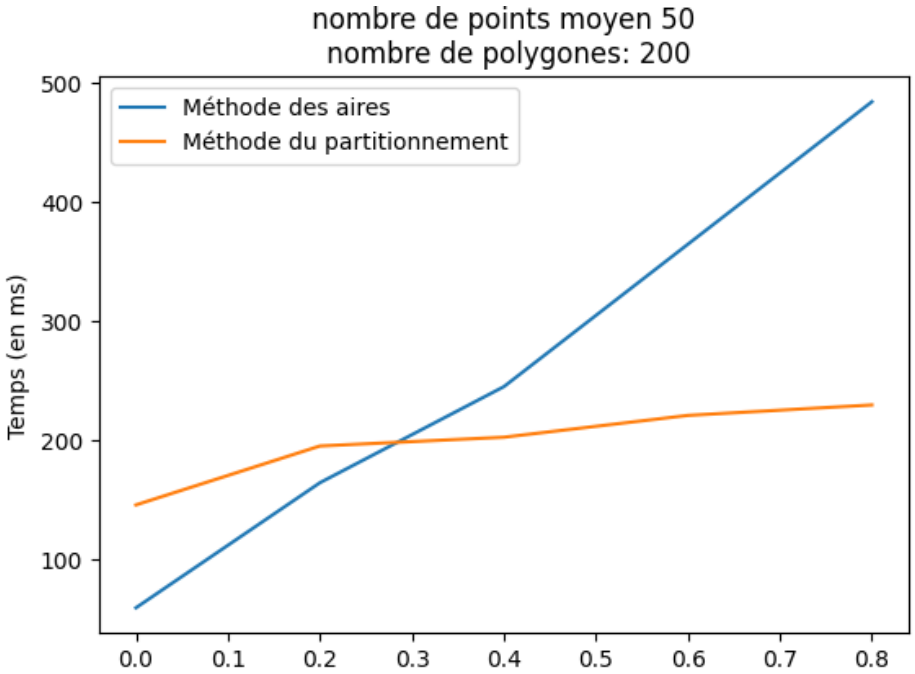

Figure 6: Variation du temps d'exécution en fonction de la probabilité d'inclusion

Enfin on varie la probabilité d'inclusion, et on voit que l'algorithme des morceaux est stable comparé à l'algorithme des aires qui varie linéairement avec la probabilité d'inclusion, et ainsi varie linéairement avec le nombre moyen d'inclusion.

5 Conclusion

Ce projet nous a permis de passer en revue plusieurs structures de données dont les tableau triés, les tables de hachage, les tableaux dynamiques, ou bien encore les arbres binaire de recherche même si à la fin on a décidé de ne pas les utiliser, on a vu à l'encontre d'un cahier de charge de temps combien le choix de la structure où stocker l'information est importante, et enfin on a vu le fait de pré-traiter les données, de les filtrer représente une astuce inestimable pour gagner de la complexité.

On a aussi pu avoir un bon classement (parmi les premiers au Test-0 avec 2040 ms) alors que notre algorithme de par sa robustesse traite des cas avec un ordre de grandeur n fois plus grand que ceux traités par les tests dans git.