Rapport – Projet INF421

Point clouds : subsampling and neighborhood

par

BENDAHI Abderrahim, FRADIN Adrien

Table des matières

Notation

Dans l'ensemble du sujet, nous adopterons les notations suivantes :

- d (un entier naturel) : la dimension de l'espace. Dans tous nos tests nous prenons $d = 2$, bien que nos codes en Python ne soient pas restreint quant à la dimension de l'espace.
- $d\colon \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}_+$ (une fonction) : une distance sur $(\mathbb{R}^d, +, \times)$. Dans tous nos exemples $d(\cdot,\cdot)$ est la distance euclidienne usuelle.
- P (un ensemble) : ensemble de point de $\mathbb{R}^d$, c'est-à-dire $P \subset \mathbb{R}^d$.
- n (un entier naturel) : le cardinal de P i.e. $n = |P|$.
- S (un ensemble) : une partie non vide de P ($S \subseteq P$), nous l'appelons ensemble des *centres*. C'est cet ensemble de centre
- k (un entier naturel) : le cardinal de S i.e. $k = |S|$.
- p_1, …, p_k (points de $\mathbb{R}^d$) : la succession (ordonnée) des centres de P et $S = \{p_1, \ldots, p_k\}$.
- $\mathscr{M}(P)$ (une matrice $n \times n$) : la matrice (symétrique) des distances mutuelles entre toute paire de points de P.
- $G = (V, E)$: la notation usuelle pour un graphe avec V l'ensemble des sommets et E l'ensemble des arêtes. Les graphes considérés ici seront non orientés, sans boucles ni multiarêtes.

Introduction : problèmes & objectifs

Les problèmes

Problème (P1) : Étant donné $P \subset \mathbb{R}^d$ de cardinal $n \in \mathbb{N}^*$ et $p_1 \in P$, nous souhaitons calculer l'ensemble des $k \in \mathbb{N}^*$ premiers centres de P c'est-à-dire, des points p_1, …, p_k de P tels que pour tout $i \in [\![2\,;n]\!]$:

$$\begin{cases} p_i \in P \backslash \{p_1, \ldots, p_{i-1}\} \\ \mathrm{d}(p_i, \{p_1, \ldots, p_{i-1}\}) = \min\limits_{p \in P\backslash\{p_1,\ldots,p_{i-1}\}} (\mathrm{d}(p, \{p_1, \ldots, p_{i-1}\})) \end{cases}$$

Problème (P2) : Étant donné $P \subset \mathbb{R}^d$ de cardinal $n \in \mathbb{N}^*$ nous souhaitons calculer le *graphe des voisins* de P c'est-à-dire, le graphe $G = (P, E)$ tel que, pour toute arête $e = \{u, v\} \in E$ où u, $v \in P$ et tout point $p \in P \backslash \{u, v\}$:

$$\mathrm{d}(u, v) \leqslant \max\left(\{\mathrm{d}(u, p), \mathrm{d}(v, p)\}\right)$$

Les objectifs

Dans ce projet, nous allons décrire 4 première fonctions (tâches 1 à 4), de plus en plus raffinées, afin de résoudre le problème (**P1**). Puis, dans un second, temps, nous décrirons 3 fonctions (tâches 5 à 7) afin de calculer le graphe des voisins.

I Subsampling

1) Basics

Algorithme : Le premier code présente une approche naïve pour déterminer l'ensemble S. Il consiste, une fois les i premiers p_1, …, p_i centres trouvés, à parcourir tous les points de $P \backslash \{p_1, \dots, p_i\}$, à déterminer leurs distances respectives à $\{p_1, \dots, p_i\}$, et finalement à choisir le point $p := p_{i+1}$ qui maximise la distance à $\{p_1, \dots, p_i\}$.

Voici le pseudo-code correspondant :

Algorithme 1 : Algorithme naïf pour la tâche 1.

Données : L'ensemble $P \subset \mathbb{R}^d$, $p_1 \in P$ et $k \in [\![1\,;n]\!]$.
Résultat : L'ensemble S des k premiers centres de P.

1 **Initialisation :**
2 $\quad S \leftarrow [p_1]$
3 $\quad \mathscr{M}(P) \leftarrow$ matrice des distances mutuelles entre chaque paire de points de P

4 **pour** $i \leftarrow 2$ **à** k **faire**
$\quad$ `// On cherche le point hors de` S `qui maximise la distance à ce dernier.`
5 $\quad p_i \leftarrow$ le point de $P \backslash \{p_1, \dots, p_{i-1}\}$ qui maximise la distance à $\{p_1, \dots, p_{i-1}\}$
$\quad$ `// On met` p_i `dans` S`.`
6 $\quad S \leftarrow [p_1, \dots, p_i]$
7 **retourner** S

Complexité temporelle : La complexité temporelle de l'algorithme est $\mathrm{O}(n \cdot k^2)$ [1], où, on rappel, n est le nombre de points dans P. Cela est dû à la recherche itérative pour chaque point non inclus dans S à chaque itération principale.

Afin d'effectuer la recherche du point p_i (ligne 5) efficacement, nous utilisons une file dans laquelle nous mémorisons les points de $P \backslash \{p_1, \dots, p_{i-1}\}$ et en défilant successivement les points, nous maintenons (hors de la file) le point à distance maximale de $\{p_1, \dots, p_{i-1}\}$.

2) More efficient

Algorithme : On améliore l'approche précédente en ne calculant plus cette matrice $\mathscr{M}(P)$ des distances mais en enregistrant seulement, pour chaque point $p \in P$ la distance de p à $S = \{p_1, \dots, p_i\}$ au fur et à mesure de l'ajout de nouveaux centres. On met à jour ces distances après chaque ajout d'un nouveau centre.

Voici le pseudo-code correspondant :

1. Ce coût est le coup de la boucle `for` ligne 4 si on omet le calcul de la matrice des distances $\mathscr{M}(P)$ qui ajoute un $\mathrm{O}(n^2)$ dans nos calculs de complexité.

Algorithme 2 : Algorithme plus efficace (en mémorisant les distances à S) pour la tâche 2.

Données : L'ensemble $P \subset \mathbb{R}^d$, $p_1 \in P$ et $k \in [\![1\,;n]\!]$.
Résultat : L'ensemble S des k premiers centres de P.

1 **Initialisation :**
2 $\quad S \leftarrow [p_1]$
3 $\quad \mathscr{L}(P) \leftarrow$ liste des distances entre chaque de points de P et $S = \{p_1\}$

4 **pour** $i \leftarrow 2$ **à** k **faire**
$\quad$ `// On cherche le point hors de` S `qui maximise la distance à ce dernier.`
5 $\quad p_i \leftarrow$ le point de $P\backslash\{p_1, \dots, p_{i-1}\}$ qui maximise la distance à $\{p_1, \dots, p_{i-1}\}$
$\quad$ `// Mettre à jour` $\mathscr{L}(P)$.
6 $\quad$ **pour** $j \leftarrow 1$ **à** n **faire**
7 $\quad\quad \mathscr{L}(P)[j] \leftarrow \min\left(\mathscr{L}(P)[j], \mathrm{d}(p_i, P[j])\right)$
$\quad$ `// On met` p_i `dans` S.
8 $\quad S \leftarrow [p_1, \dots, p_i]$
9 **retourner** S

Complexité temporelle : La complexité temporelle de cet algorithme est maintenant en $\mathrm{O}(n \cdot k)$, où n est le nombre de points dans P. Cette amélioration est obtenue grâce à la mémorisation non plus de la matrie $\mathscr{M}(P)$ mais la liste $\mathscr{L}(P)$ des distances de chaque $p \in P$ à S.

Là encore, nous utilisons, comme à la tâche 1 une file afin de rechercher efficacement le point p_i (ligne 5) en maintenant l'ensemble $P\backslash\{p_1, \dots, p_{i-1}\}$ à jour.

3) Metric case : small k

Algorithme : Le troisième algorithme améliore la recherche du point de $P\backslash S$ à distance maximale de S en regroupant les points de $P\backslash S$ par *région*. Étant donné $i \in [\![1\,;k]\!]$, on définit la région du centre p_i par :

$$R_i := \{p \in P\backslash S \,/\, \forall q \in S\backslash\{p_i\}\,, \mathrm{d}(p, p_i) < \mathrm{d}(q, p_i)\}$$

et nous définissons le rayon de la région R_i par :

$$r_i := \max\left(\{\mathrm{d}(p_i, p) \,/\, p \in R_i\}\right)$$

et pour tout $p \in R_i$, p_i est le *parent* de p.

Évidemment, le calcul des centres de P se fait au fur et à mesure si bien que les régions calculées à une certaine itération i (de la boucle `for` principale) sont susceptibles d'être modifiées (i.e. leur rayon peut décroître avec le temps). Nous observons que :

- à une certaine itération i, le point $p \in P\backslash S$ le plus éloigné de son parent est le prochain centre i.e. $p_i := p$.
- le rayon r_i de la nouvelle région associée à p vérifie :

  $$r_i < r$$

  où r est le rayon de la région R du parent de p. En effet, si tel n'était pas le cas alors la région R_i associée à p contiendrait au moins un point $q \in P\backslash S$ tel que $\mathrm{d}(p, q) \geqslant r$ ce qui signifie que la rayon r_k de la région R_k associée à q satisfait :

  $$r_k > \mathrm{d}(p, q) \geqslant r$$

  car le point q doit etre plus proche de p que de son parent. Ceci contredit le fait que p doit être le prochain centre.
- si une région R_j (de centre p_j) de rayon r_j vérifie :

  $$\mathrm{d}(p, p_j) \geqslant 2r_j$$

  alors aucun point de la région R_j ne se retrouvera dans la région R_i associée à p. En effet, si tel n'était pas le cas, en notant $q \in R_j$, nous aurions :

  $$\begin{aligned} r_j &> \mathrm{d}(p, q) \\ &\geqslant \mathrm{d}(p, p_j) - \mathrm{d}(p_j, q) \\ &\geqslant \mathrm{d}(p, p_j) - r_j \\ &\geqslant 2r_j - r_j \\ &= r_j \end{aligned}$$

ce qui ne se peut.

Ces observations permettent d'éviter de mettre à jour toutes les régions présentes lors de la création d'une nouvelle région, seulement les régions « *proches* » de la nouvelle région.

Voici le pseudo-code correspondant :

Algorithme 3 : Algorithme plus efficace, pour la tâche 3.

Données : L'ensemble $P \subset \mathbb{R}^d$, $p_1 \in P$ et $k \in [\![1\,;n]\!]$.
Résultat : L'ensemble S des k premiers centres de P.

```
1  Région (objet) :
2      c ← centre (c'est l'indice du point dans la liste P)
3      r ← rayon
4      ℓ ← liste des (indices des) points de la région
5      p ← point au bord (un indice)

6  Initialisation :
7      S ← [p_1]
8      𝒟(P) ← liste des distances entre chaque paire de points de P de S = {p_1}
9      ℛ ← file vide des régions

   /* Met à jour la région old dans le cas où la région new viendrait lui piquer des points.  */
10 Fonction Split(old, new) :
11     pour p dans old.ℓ faire
           // Si p est « avalé » par la nouvelle région.
12         si d(p, new.c) < d(p, old.c) alors
13             Retirer p de old et le mettre dans new
   /* Créer une nouvelle région de centre c.                                                  */
14 Fonction Create(c) :
15     𝒞 ← région vide de centre c
16     pour region dans ℛ faire
           // Si la nouvelle région peut piquer des points dans la région region.
17         si d(c, region.c) < 2region.r alors
18             Split(region, 𝒞)
19     Mettre 𝒞 dans la file ℛ

20 Mettre dans ℛ la région initiale (centre p_1, contenant P\{p_1} avec le bon rayon, etc.)
21 pour i ← 2 à k faire
       // On cherche la région de rayon maximal.
22     region ← la région de rayon maximal
       // On met le point au bord de region dans S.
23     S ← [p_1, ..., region.p]
       // On crée la nouvelle région.
24     Create(region.p)
25 retourner S
```

4) Metric case (hard)

Algorithme : Pour la tâche 4 qui est un prolongement de la tâche 3, nous avons identifié plusieurs portions du code à améliorer / optimiser :

- **(A)** la recherche de la région de rayon maximal
- **(B)** la recherche et la modification des régions à modifier qui comprends :
  - **(B.1)** la recherche effective des régions (boucle `for` ligne 16)
  - **(B.2)** le *splitting* de ces régions
- **(C)** la mise à jour des régions modifiées (dans la structure de donnée utilisée pour stocker ces régions) et l'ajout de la nouvelle région.

Pour la quatrième tâche, nous avons repris la même structure que pour la tâche 3 en modifiant tout d'abord la

structure de donnée contenant les régions. Pour la tâche 3, avec l'utilisation de file, nous avions comme complexités (dans le pire cas) :

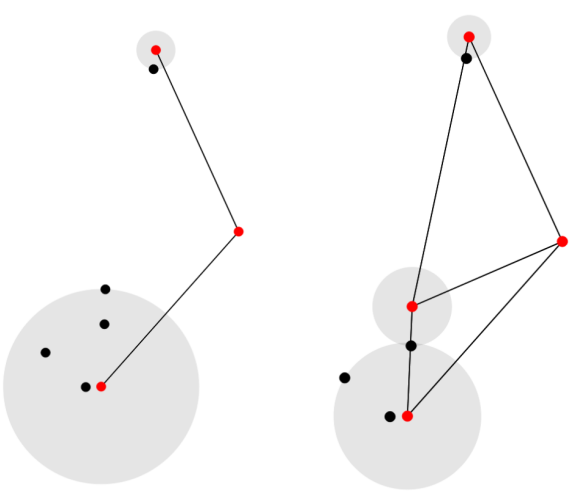
- **(A)** : $O(|\mathscr{R}|) = O(k)$ pour parcourir les régions dans la file ($\mathscr{R}$ est la file des régions).
- **(B)** :
  - **(B.1)** : $O(\mathscr{R}) = O(k)$.
  - **(B.2)** : $O(|R_i|)$ pour une (ancienne) région $R_i \in \mathscr{R}$ particulière.
- **(C)** : $O(1)$, les anciennes régions sont mises à jour en parallèle de **(B)** et la nouvelle région est simplement enfiler (coût : $O(1)$).

Nous avons utilisé d'un *tas-max* afin de représenter une file de priorité avec comme priorité le rayon des régions. L'utilisation du tas-max permet d'améliorer la complexité du problème **(A)** mais entraîne aussi des coûts supplémentaire pour le maintient des invariants du tas, voici les complexités actuelles :

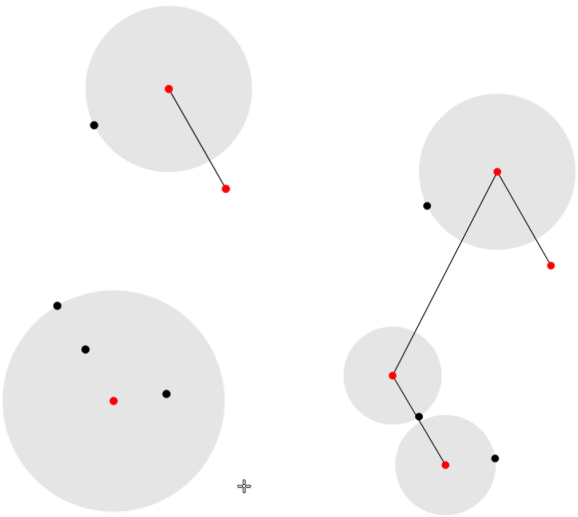
- **(A)** : $O(1)$, il s'agit de regarder (sans extraire !) la région à la racine ud tas.
- **(B)** : ces complexités ne changent pas par rapport à la tâche 3.
- **(C)** : $O(|\mathscr{R}|) = O(k)$ [2] pour la mise à jour des anciennes régions (qui se fait petit-à-petit en parallèle de **(B)**) et la nouvelle région est rajouter dans le tas avec un coup $O(\log(k))$.

Pour ce qui est du *graphe des amis*, nous n'avons malheureusement pas trouvé de contrainte suffisante qui garantisse que si $p \in P \backslash S$ est le prochain centre et $s \in S$ son parent alors, les voisins de p dans le graphe des amis sont parmi les voisins de s.

Nous avons testé deux contraintes, l'idée de ces contraintes est dire que deux centres s et s' sont voisins dans le graphe des amis s'il n'existe pas d'autre centre suffisamment proche qui empêcherait la région créée par p d'empiéter sur la région de s'. Ci-dessous nous montrons ces deux contraintes et les contre-exemples associés :



(a) *Les deux régions grisées ne sont pas voisines et pourtant, le nouveau centre créée admet ces deux régions pour voisins.*



(b) *Même constat.*

Pour la figure (a) la contrainte est que deux centre s et s' sont voisins ssi pour tout $\widetilde{s} \in S$:

$$(C1): \quad \mathrm{d}(s, s') < r + r' + \max(\{\mathrm{d}(\widetilde{s}, s), \mathrm{d}(\widetilde{s}, s')\})$$

où r (resp. r') est le rayon de la région de s (resp. s')

Pour la figure (b), s et s' sont voisins ssi pour tout $\widetilde{s} \in S$:

$$(C2): \quad \mathrm{d}(s, s') < r + r' + \min(\{\mathrm{d}(\widetilde{s}, s), \mathrm{d}(\widetilde{s}, s')\})$$

où r (resp. r') est le rayon de la région de s (resp. s')

Notons que le graphe induit par la contrainte de voisinage $(C1)$ fourni, lorsque $k = n$ la graphe des voisins (calculé dans la section suivante).

2. Notons que, si nous mettons à jour le tas en partant des nœuds les plus profonds puis en remontant, comme les régions ont des rayons décroissants, les nœuds mis à jour descendrons. Ainsi, cette opération de mise à jour ne nous coût qu'au pire un $O(k)$ au lieu d'un $O(k \log(k))$ si nous mettions à jour les régions du tas dans un ordre quelconque.

II Neighborhood

1) Basic

Algorithme : Le premier code présente une approche naïve pour déterminer le graphe des voisins, l'idée est de parcourir tous les couples de points et, pour chaque couple $(u, v) \in P^2$, de parcourir tous les points $p \in P$ pour déterminer si p est bloquant pour (u, v) i.e. si :

$$\max\left(\{\mathrm{d}(u,p), \mathrm{d}(v,p)\}\right) < \mathrm{d}(u,v)$$

Voici un pseudo-code de l'algorithme :

Algorithme 4 : Algorithme naïf pour la tâche 5.

Données : L'ensemble $P \subset \mathbb{R}^d$,.
Résultat : Le graphe des voisins G, de P.

Initialisation :
  $G = (P, E) \leftarrow$ graphe vide
  $\mathscr{M}(P) \leftarrow$ matrice des distances entre chaque paire de points de P

pour $i \leftarrow 1$ **à** n **faire**
  pour $j \leftarrow 1$ **à** $i - 1$ **faire**
    si *il n'existe pas de bloqueur* $p \in P \setminus \{P[i], P[j]\}$ **alors**
      `// Ajout d'une arête.`
      $E \leftarrow \{i, j\}$
retourner G

Complexité temporelle : La complexité temporelle de l'algorithme est en $\mathrm{O}(n^3)$, où n est le nombre de points dans P avec les trois boucles `for` imbriquées.

2) More efficient

Algorithme : Pour l'algorithme, nous implémentons celui fourni dans l'énoncé sous la forme d'une procédure `find_neighbors` qui prend en entrée un point $p \in \mathbb{R}^d$, un ensemble de points P et la matrice $\mathscr{M}(P)$ des distances et renvoie les voisins (dans le graphe des voisins) de p parmi les points de P (sous forme d'une liste).

Cette procédure sera ré-utilisée pour la tâche 7.

La tâche 6 se contente d'une boucle `for` sur P en appelant la fonction `find_neighbors` citée plus haut afin de construire sommet après sommet le graphe des voisins G de P.

Complexité temporelle : La fonction `find_neighbors` a une complexité de $\mathrm{O}(n)$, où n est le nombre de points dans l'ensemble donné P. Ainsi, la complexité totale de la tâche 6 est un $\mathrm{O}(n^2)$.

3) Incremental

Algorithme : Pour cet algorithme, nous avons besoin de la matrice des distances mutuelles entre chaque point de P, où $P \subset \mathbb{R}^d$. Pour un nouveau point $p \in \mathbb{R}^d \setminus P$, nous mettons à jour cette matrice ainsi que le graphe des voisins G (de P) qui est fourni en entrée.

L'algorithme procède en deux temps :
- vérifier pour quelle arêtes $e \in E$ dans $G = (P, E)$ le point p est-il bloqueur et si tel est le cas, on retirer cet arête e de G.
- ajouter les voisins de p grâce à la fonction `find_neighbors`.

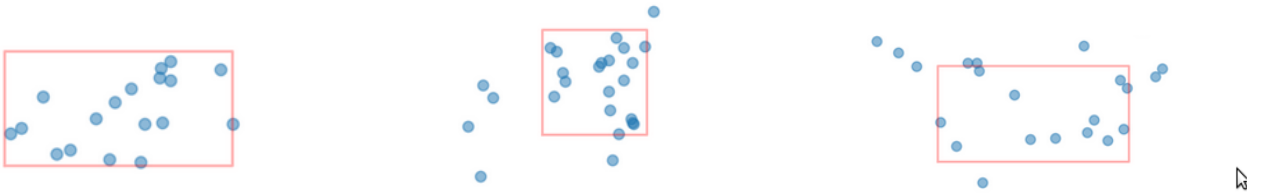
Complexité temporelle : La complexité temporelle de l'algorithme est en $O(n)$ où n est le nombre de points de P, car le nombre de voisins de chaque point i est supposé en $O(1)$ par l'énoncé, donc la boucle principale a une complexité linéaire, et il en est de même pour la fonction *find_neighbors*.

III Datasets

1) Datasets generation

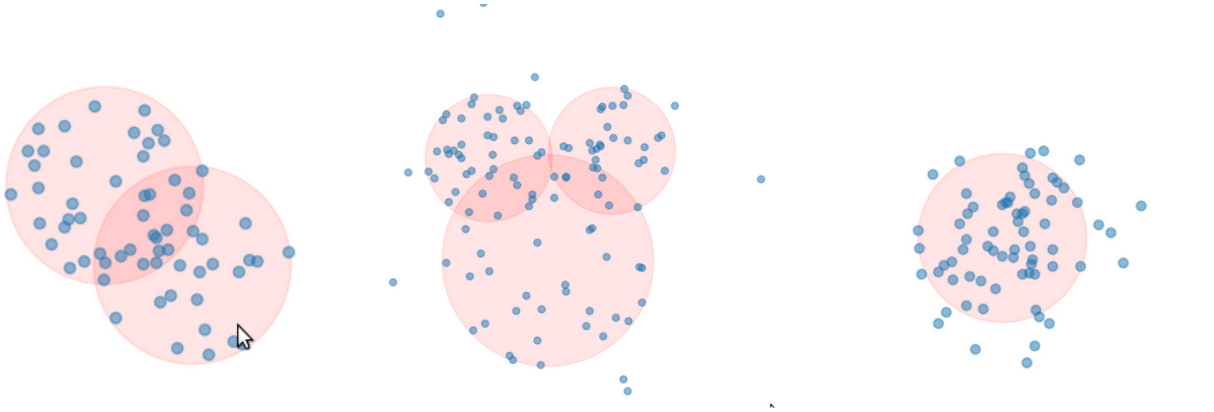
Nos jeux de données sont générés aléatoirement et nous en avons de trois catégories différentes avec présence de bruit (une partie des points sont disposés uniformément dans une certain pavé, les autres points restes inchangés) et / ou des perturbations (certains points sont déplacés d'un vecteur $\vec{e} \sim \mathcal{N}(0,1)$ de loi normale et dilatés d'un certain facteur d'échelle $r \in \mathbb{R}_+^*$ – souvent, $0 < r < 1$) :

– génération uniforme de points dans un pavé de $\mathbb{R}^d$:



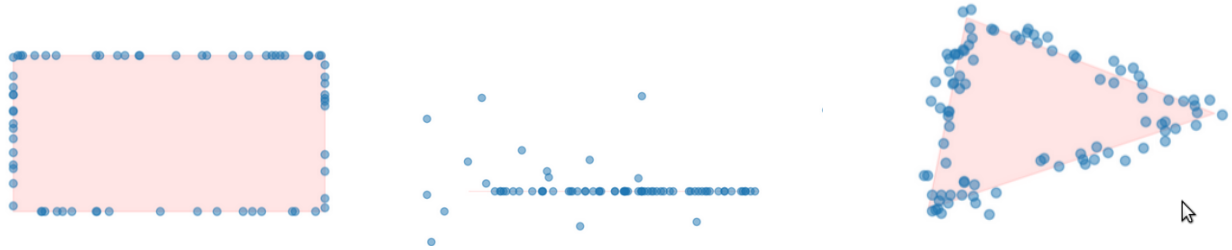
(a) *Zone rectangulaire.* (b) *Zone rectangulaire avec du bruit.* (c) *Zone rectangulaire des perturbations.*

– génération de clusters dans $\mathbb{R}^d$:



(a) 2 *clusters.* (b) 3 *clusters avec du bruit.* (c) 1 *cluster avec perturbations.*

– génération de points uniformes sur une ligne polygonale dans $\mathbb{R}^d$:



(a) *Ligne polygonale.* (b) *Ligne polygonale avec du bruit.* (c) *Ligne polygonale avec perturbations.*

2) Datasets manipulations

Nous avons deux fonctions afin de sauvegarder et de charger en mémoire un jeu de données. Le format d'un jeu de donnée est expliqué dans le notebook Jupyter : chaque fichier dispose d'un nom au format :

$$d\_n\_[\text{nom}].\text{txt}$$

où d est la dimension de l'espace dans lequel se trouvent les points du jeu de donnée, n est le nombre de points du jeu de donnée et [nom] est le nom du jeu de donnée. Par ailleurs, nous convenons aussi de rajouter à la dernière ligne du jeu de donnée le point p_1 qui est fréquemment utilisé dans la partie 1 (nous recopions ses coordonnées à la ligne $n+1$ du fichier).

Nous mettons à disposition du lecteur une application web écrite en JavaScript et située le dossier `web`, qui permet de jouer avec les différents algorithmes (tâches 1, 2, 3, 5 et 6) et de charger / enregistrer des jeux de données avec le format prescrit ci-dessus. Se référer au fichier `README.md` à la racine du projet.

IV Benchmark

Pour comparer les performances des différents algorithmes nous les avons exécutés sur plusieurs ensembles de données. Les résultats obtenus sont regroupés dans le tableau suivant (nous avons compté le temps, par soucis de rapidité, sur une seule exécution à chaque test) :

function	n	dataset name	time in sec.
task_5	18	2_18_0.txt	0.005069
task_5	16	2_16_1.txt	0.000922
task_5	18	2_18_2.txt	0.002159
task_5	45	2_45_3.txt	0.008744
task_5	35	2_35_4.txt	0.007116
task_5	33	2_33_5.txt	0.007091
task_5	141	2_141_6.txt	0.124069
task_5	195	2_195_7.txt	0.193670
task_5	100	2_100_8.txt	0.080921
task_5	344	2_344_9.txt	0.443889
task_5	433	2_433_10.txt	0.541500
task_5	467	2_467_11.txt	0.628888
task_5	591	2_591_12.txt	1.093411
task_5	927	2_927_13.txt	2.700560
task_5	751	2_751_14.txt	1.915688
task_5	2161	2_2161_15.txt	Timeout
task_5	2791	2_2791_16.txt	Timeout
task_5	2685	2_2685_17.txt	Timeout
task_5	6359	2_6359_18.txt	Timeout
task_5	5849	2_5849_19.txt	Timeout
task_5	9286	2_9286_20.txt	Timeout

function	n	dataset name	time in sec.
task_6	18	2_18_0.txt	0.002216
task_6	16	2_16_1.txt	0.001901
task_6	18	2_18_2.txt	0.002382
task_6	45	2_45_3.txt	0.008419
task_6	35	2_35_4.txt	0.007681
task_6	33	2_33_5.txt	0.006723
task_6	141	2_141_6.txt	0.0541
task_6	195	2_195_7.txt	0.156235
task_6	100	2_100_8.txt	0.051780
task_6	344	2_344_9.txt	0.251869
task_6	433	2_433_10.txt	0.454371
task_6	467	2_467_11.txt	0.490659
task_6	591	2_591_12.txt	0.836856
task_6	927	2_927_13.txt	2.017283
task_6	751	2_751_14.txt	1.274463
task_6	2161	2_2161_15.txt	Timeout
task_6	2791	2_2791_16.txt	Timeout
task_6	2685	2_2685_17.txt	Timeout
task_6	6359	2_6359_18.txt	Timeout
task_6	5849	2_5849_19.txt	Timeout
task_6	9286	2_9286_20.txt	Timeout

function	n	dataset name	time (1 iteration) in sec.
task_7_benchmark	18	2_18_0.txt	0.001865
task_7_benchmark	16	2_16_1.txt	0.001484
task_7_benchmark	18	2_18_2.txt	0.001715
task_7_benchmark	45	2_45_3.txt	0.006493
task_7_benchmark	35	2_35_4.txt	0.005355
task_7_benchmark	33	2_33_5.txt	0.004408
task_7_benchmark	141	2_141_6.txt	0.041992
task_7_benchmark	195	2_195_7.txt	0.132100
task_7_benchmark	100	2_100_8.txt	0.039518
task_7_benchmark	344	2_344_9.txt	0.284741
task_7_benchmark	433	2_433_10.txt	0.395388
task_7_benchmark	467	2_467_11.txt	0.417329
task_7_benchmark	591	2_591_12.txt	0.658521
task_7_benchmark	927	2_927_13.txt	1.569363
task_7_benchmark	751	2_751_14.txt	1.014913
task_7_benchmark	2161	2_2161_15.txt	9.384394
task_7_benchmark	2791	2_2791_16.txt	Timeout
task_7_benchmark	2685	2_2685_17.txt	Timeout
task_7_benchmark	6359	2_6359_18.txt	Timeout
task_7_benchmark	5849	2_5849_19.txt	Timeout
task_7_benchmark	9286	2_9286_20.txt	Timeout

Pour les tâches 1, 2, 3 et 4 les tableaux sont trop longs. Les résultats sont accessibles depuis le fichier à la racine du projet : `benchmark_0_without_results.txt`.