

Algorithmique PEIP2

TP4 : Projet

Le but de ce mini projet est d'implémenter des opérations classiques de géométrie dans le plan.

Instructions pour le rendu

- Ce projet est à faire en **binôme** ; des échanges pourront se faire entre les groupes de TP pour résoudre les problèmes de parité et d'affinité.
- Les rendus devront se faire par mail à votre chargé de TP (en mettant `fpirot@liscn.fr` en copie), au plus tard le **dimanche 17 décembre 2023 à 23h59**.
- Un rendu complet doit contenir **deux fichiers**.
 - Un **fichier de code** `prenom1_nom1_prenom2_nom2.py`, qui doit produire une sortie graphique illustrant la résolution des exercices du projet sur un nuage de points généré aléatoirement. Il doit être **structuré** (les définitions de fonctions doivent être séparées des exécutions), **annoté** (décrire le rôle de chaque fonction avant sa définition), convenablement **commenté** (expliquer le comportement attendu de chaque instruction complexe, et préciser ce que représentent les variables internes), et **fonctionnel** (le programme doit pouvoir se lancer directement depuis un terminal avec `python3`, sans avoir à faire de manipulation dans son code).
 - Un **rapport** `prenom1_nom1_prenom2_nom2.pdf` qui décrit et justifie les choix d'implémentation pour chaque question. Un rapport de qualité doit suffire à comprendre le fonctionnement du programme, sans avoir à en lire le code.
- Les rendus en retard (même de quelques minutes) seront pénalisés : 2 points en moins par tranche de 12h entamée. Prévoyez suffisamment de marge pour faire votre envoi, et pour les utilisateurs de tablette gardez à l'esprit que les fichiers trop volumineux (>15Mo) ne peuvent pas être envoyés simplement en pièce jointe.

1 Dessins dans le plan avec le module Turtle

Nous représentons les points dans le plan à l'aide de la classe python `Point` définie comme suit (**inclure le bloc de code suivant au début de votre projet**).

```
class Point:
    def __init__(self, i, j):
        self.x = i
        self.y = j

    def __repr__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
```

Cette classe nous permet de :

- construire un point `p` de coordonnées (x, y) ;

```
p = Point(x, y)
```

- afficher les coordonnées d'un point p ;

```
print (p)
```

- tester l'égalité entre deux points p et q .

```
if p==q:  
    ...
```

Étant donné un point p , nous avons accès aux deux variables $p.x$ et $p.y$ qui représentent ses coordonnées.

Nous définissons en outre des variables globales qui délimitent les coordonnées des points que nous allons construire au sein de ce projet (**inclure le bloc de code suivant au début de votre projet**).

```
min_x = -200  
max_x = 200  
min_y = -200  
max_y = 200
```

1. Écrire une fonction `randpoint()` qui renvoie un point dont les coordonnées sont aléatoires et uniformes au sein du rectangle de coin inférieur gauche (min_x, min_y) et de coin supérieur droit (max_x, max_y) .

Afin de mieux visualiser le déroulement des algorithmes mis en place au cours de ce projet, nous allons utiliser un affichage graphique. Pour cela, nous utilisons les fonctions mises à disposition par le module `turtle` de Python, dont la documentation est disponible à cette adresse : <https://docs.python.org/3/library/turtle.html>. De manière informelle, ce module permet de déplacer un curseur au sein d'une fenêtre graphique, en traçant (après un appel à `turtle.pendown()`) ou non (après un appel à `turtle.penup()`) des formes sur son passage.

On souhaite accélérer au maximum l'affichage graphique de l'exécution des algorithmes. Pour cela, **inclure le bloc de code suivant au début de votre projet**.

```
turtle.speed(0)  
turtle.hideturtle()
```

De plus, on veut que la fenêtre graphique reste visible lors de son exécution depuis un terminal. Pour cela, **inclure le bloc de code suivant à la fin de votre projet**.

```
turtle.Screen().exitonclick()
```

2. Écrire une procédure `drawpoint(p)` qui dessine un point aux coordonnées du point p , à l'aide de la fonction `turtle.dot`. Attention à ne rien dessiner de plus que le point !
3. Écrire une procédure `drawpoly(l)` qui dessine le polygone donné par la liste l de ses sommets. Ce polygone peut s'auto-intersecter.
4. Écrire une procédure `trigosort(l,p)` qui prend en argument un nuage de points l et un point pivot p , et qui trie les éléments $q \in l$ par ordre croissant selon l'angle (compris dans l'intervalle $[0, 2\pi[$) que forme le vecteur \vec{pq} avec l'axe des abscisses. On utilisera pour cela la fonction `sorted` de Python, avec la clé appropriée. On rappelle que cette fonction est de complexité $O(n \ln n)$ sur une entrée de taille n . La documentation de cette fonction est disponible à l'adresse suivante : <https://docs.python.org/fr/3/howto/sorting.html>.
5. Tester les fonctions précédentes en construisant un nombre $N=100$ de points aléatoires, puis en dessinant le nuage de points obtenu, ainsi que le polygone non-intersectant obtenu en les considérant dans l'ordre induit par le pivot $(0, 0)$.

2 Enveloppe convexe

L'enveloppe convexe d'un nuage (fini) de points X est le plus petit polygone convexe P tel que X est entièrement contenu à l'intérieur de P . Le but de cette partie est d'implémenter un algorithme qui construit l'enveloppe convexe d'un nuage de points, de façon efficace.

L'algorithme que nous allons utiliser à cette fin fonctionne en deux temps. Il commence par trouver un point pivot p_0 d'ordonnée minimale dans X . On trie ensuite les points $q \in X$ par ordre croissant selon l'angle que forme le vecteur $\overrightarrow{p_0q}$ avec l'axe des abscisses. Enfin, on construit l'enveloppe convexe P en partant de la liste $[p_0]$, et en ajoutant les points de X un par un selon l'ordre établi, en retirant les derniers ajoutés si la ligne brisée formée par P devient non convexe. Une animation de cet algorithme est disponible à l'adresse suivante : <https://www.lri.fr/~fpirot/teaching/peip2/convex.gif>.

1. Justifier que le point p_0 se trouve nécessairement dans l'enveloppe convexe de X .
2. Écrire une fonction `convexhull(l)` qui renvoie la liste des points de l'enveloppe convexe du nuage de points représenté par la liste de points l .
3. Quelle est la complexité d'un appel à `convexhull(l)`, en fonction du nombre n de points dans l ?
4. Écrire une fonction `drawconvexhull(l)` qui calcule l'enveloppe convexe de l , et dessine au passage les étapes du calcul de manière similaire à l'animation citée plus haut. On pourra utiliser la fonction `turtle.undo()` qui efface le dernier tracé effectué.
5. Tester la fonction `drawconvexhull` sur le nuage de points aléatoire de la section précédente. Tester la fonction `convexhull` sur le plus grand nombre possible de points de sorte que le temps d'exécution soit de l'ordre de 1 seconde. Quelle est la taille obtenue ?

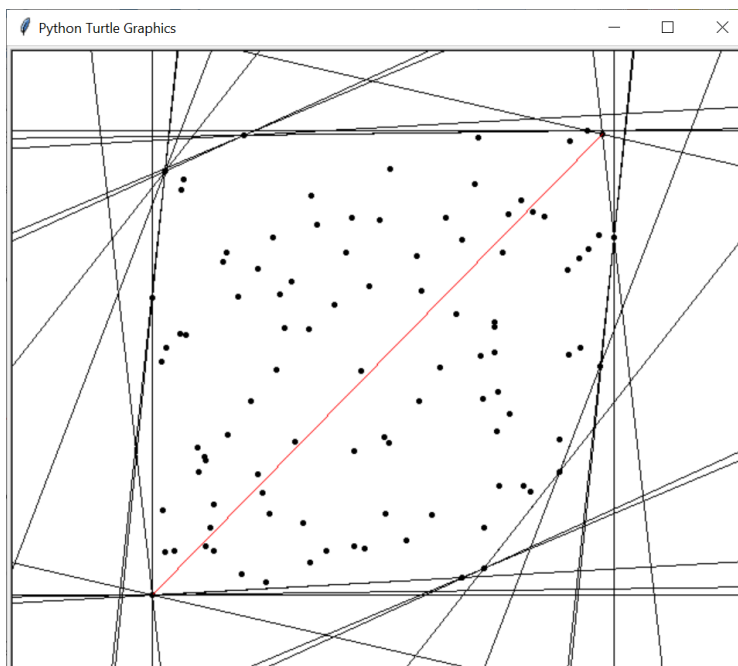
3 Diamètre en temps linéaire

Le but de cette section est d'implémenter un algorithme de complexité linéaire permettant de calculer le diamètre d'un polygone convexe.

Soit P un polygone convexe. Deux sommets $x, y \in P$ sont *antipodaux* s'il existe deux droites parallèles tangentes au polygone P en x et en y respectivement. Il est clair que le point x_0 d'ordonnée minimale et le point y_0 d'ordonnée maximale dans P forment une paire de points antipodaux. En faisant tourner P juste assez pour qu'un de ces deux points change, on trouve une nouvelle paire de points antipodaux. En répétant cette opération jusqu'à avoir fait une rotation de 180° de P , on trouve toutes les paires possibles de points antipodaux dans P . Une animation de cet algorithme est disponible à l'adresse suivante : <https://www.lri.fr/~fpirot/teaching/peip2/diameter.gif>

1. En utilisant le principe décrit ci-dessus, montrer qu'un polygone P à n sommets contient exactement n paires de points antipodaux, en supposant qu'il ne contient pas de côtés parallèles.
2. Écrire une fonction `AntipodalPair(t)` de complexité linéaire qui retourne les indices dans t de la paire (x_0, y_0) de points antipodaux décrits ci-dessus, lorsque t est un tableau contenant les points de P (dans le bon ordre).
3. Écrire une fonction `NextAntipodalPair(t, i, j)` qui renvoie les indices dans t de la paire de points antipodaux suivante selon la procédure décrite ci-dessus, en partant d'une paire de points antipodaux d'indices i et j . Cette fonction devra être de complexité $O(1)$.
4. Le diamètre d'un polygone convexe est la distance maximale réalisée par une paire de points antipodaux. En utilisant cette propriété, écrire une fonction `diameter(t)` de complexité linéaire calculant le diamètre d'un polygone convexe dont les points sont contenus (dans le bon ordre) dans le tableau t .

5. Écrire une fonction `drawdiameter` (1) de complexité linéaire calculant le diamètre d'un polygone convexe 1, et qui dessine les étapes du calcul de manière similaire à l'animation citée plus haut, et trace en rouge le diamètre du polygone en fin de calcul.
6. Tester la fonction `diameter` sur le plus grand polygone convexe possible de sorte que le temps d'exécution soit de l'ordre de 1 seconde. Quelle est la taille obtenue ?
7. Appliquer la fonction `drawdiameter` au polygone convexe obtenu par l'enveloppe convexe du nuage de point de la section précédente. Quelle est la complexité du calcul du diamètre d'un nuage de points quelconque ?



Un exemple de la sortie graphique attendue à la fin de votre programme.