

Exercice 1 : Dessins dans le plan avec Turtle

Question 1 :

La fonction est relativement simple.

Nous utilisons la fonction *randint()* de la bibliothèque random pour choisir les coordonnées (x,y) avec x, y des entiers choisis aléatoirement.

La fonction *randint()* permet de choisir un intervalle pour les valeurs de sortie.

Ainsi, $\min_x \leq x \leq \max_x$ ET $\min_y \leq y \leq \max_y$

La fonction renvoie une donnée de type Point qui prend les valeurs x et y.

Elle renvoie Point(x,y).

Question 2 :

Cette fonction n'utilise que la bibliothèque turtle. Elle permet d'afficher un point aux coordonnées (x,y).

On s'assure que le curseur ne dessine pas avant et après le code par mesure de sécurité, pour ne pas dessiner des traits. On utilise donc la commande *penup()*.

Puis, on se déplace à la position voulue (x,y) avec *setpos()*.

Enfin, on fait un point d'épaisseur 5 avec la commande *dot()*.

Question 3 :

Cette fonction permet de dessiner un polygone à partir d'une liste de points.

Comme précédemment, on s'assure que le curseur ne dessine avant et après avec *penup()*.

On se déplace au premier point avec *setpos()*, et on commence à tracer avec *pendown()*.

Puis, on se déplace de point en point avec la fonction *goto()* (qui joue le même rôle que *setpos()*) insérée dans une boucle *for*.

Il ne reste plus qu'à tracer le dernier trait entre le dernier et le premier point.

Question 4 :

Cette fonction a pour but de trier les points d'une liste **I** en fonction de son angle θ entre le vecteur PQ (P : point pivot ; Q appartenant à **I**) et l'axe des abscisses passant par P.

Pour ce faire, nous créons tout d'abord une liste **I_angle**.

Nous allons remplir **I_angle** avec les angles θ associés à chaque point.

Ainsi, **I_angle** sera associée à **I**. C'est-à-dire que l'angle du n-ième point dans **I_angle** correspond au n-ième point dans **I** (cf. schéma 1).

On utilise une boucle *for* pour un i allant de 0 à la taille de **I**. À chaque tour de boucle, on rajoute une valeur dans **I_angle** avec la fonction *append*.

Concernant le calcul de l'angle, on utilise une relation trigonométrique :

En effet, lorsque le point Q est au-dessus de l'axe des abscisses de P, on calcule l'angle avec la relation du schéma (cf. schéma 2).

Sinon, lorsque le point Q est en-dessous de l'axe des abscisses de P, θ devient : $2\pi - \theta$, car *arcsin* ne donne que les angles compris entre 0 et π .

	Point 1	Point 2	Point 3	.	.	.	Point n
l_angle	θ_1	θ_2	θ_3	.	.	.	θ_n

Schéma 1 :

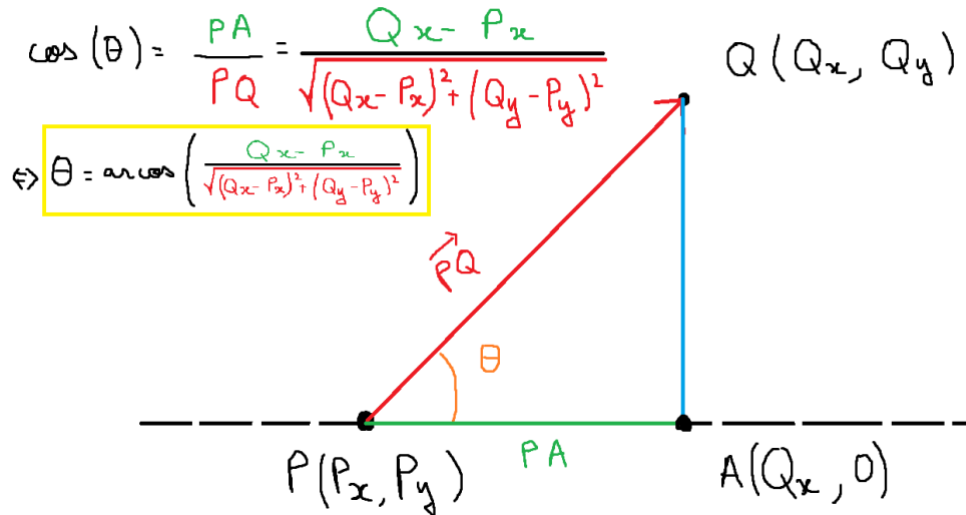


Schéma 2 :

Ensuite, pour pouvoir associer les deux listes, on utilise la fonction `zip()`. C'est-à-dire que la position du point dans la liste `l` et de son angle resteront associées même après tri croissant; qui est lui-même effectué par la fonction `sorted` (avec la clé : `lambda x : x[0]` car on ne veut trier que les angles). Après avoir trié, on dé-zippe avec `zip`. Cette partie correspond à la ligne de code suivante :

```
l_angle, l = zip(*sorted(zip(l_angle, l), key = lambda x: x[0]))
```

On retourne la liste `l` triée par angle croissant.

Python Turtle Graphics

Question 5 :

Voici ce que l'on obtient après avoir respecté les limites données, et en prenant comme point `P`, l'origine :

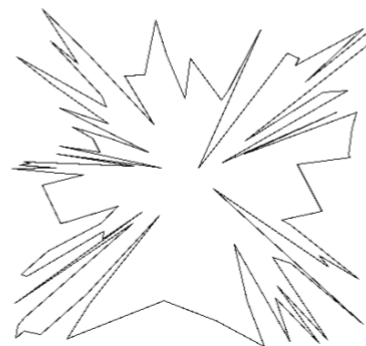


Schéma 3 :

Exercice 2 : Enveloppe convexe

Question 1 :

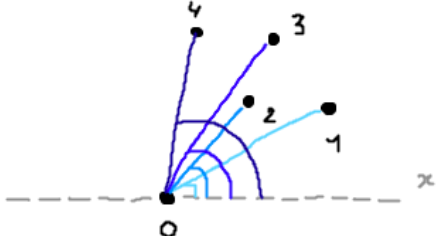
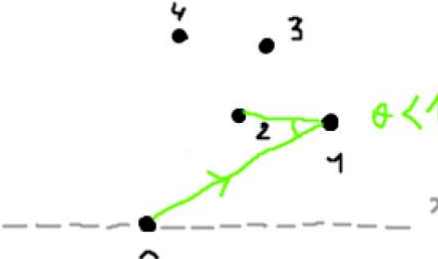
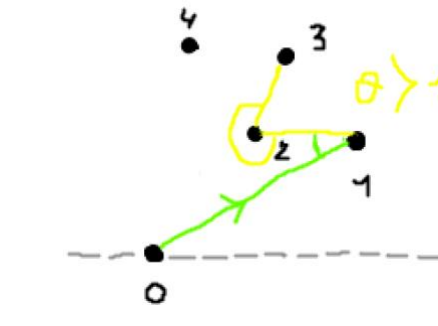
Cette question a été la plus difficile jusque-là. On commence par trouver dans notre nuage de points le point le plus bas (selon l'axe y). Pour ce faire, on procède par un tri avec la fonction `sorted` (on obtient **`I_trié`**). Après avoir récupéré ce point, on commence à créer la liste de sommets (**`I_sommets`**) de l'enveloppe convexe. Cette enveloppe convexe commence donc par ce point minimum.

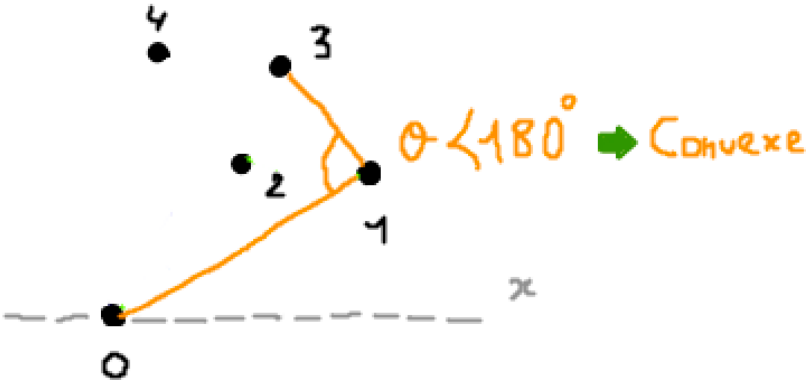
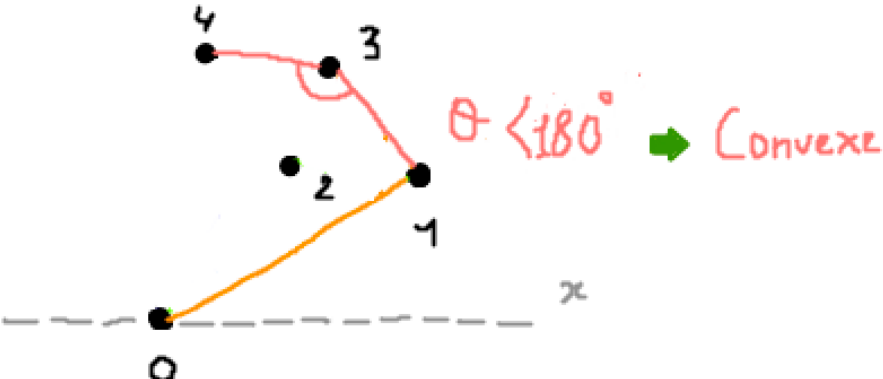
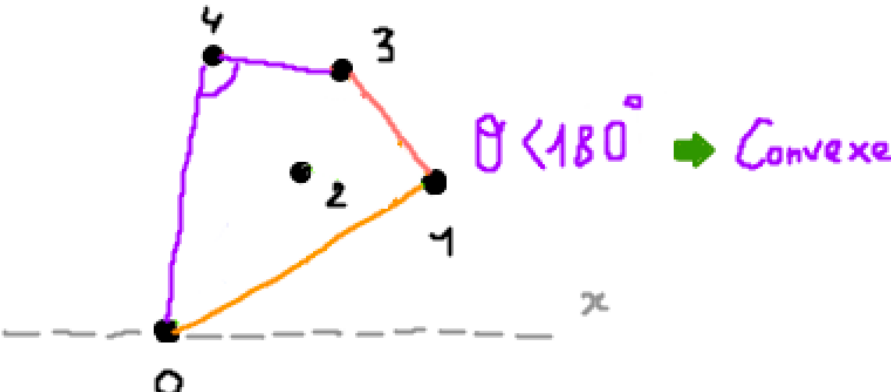
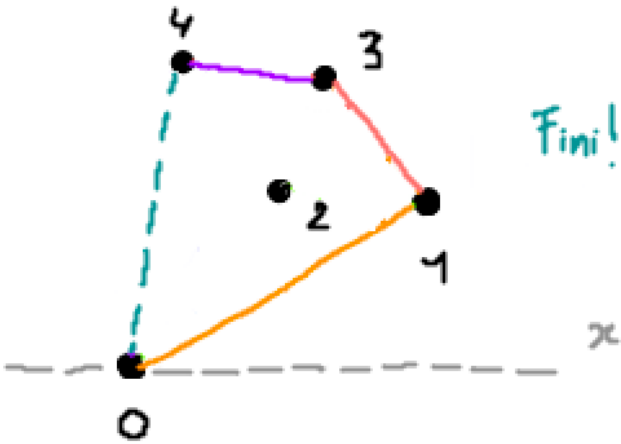
On trie ensuite **`I_trié`** par angle avec la fonction `trigosort()` avec **`p_min`** le point pivot. (Il ne faut pas oublier de supprimer d'abord le premier point minimum de **`I_trié`**)

On ajoute à **`I_sommets`** les deux premiers éléments de **`I_trié`**.

On entre alors dans une boucle qui va ajouter tous les points de **`I_trié`** un par un, en vérifiant à chaque fois l'angle créé entre les trois derniers points de **`I_sommets`**.

Si l'angle n'est pas convexe, on supprime de la liste **`I_sommets`** l'avant-dernier élément ajouté. On vérifie par la suite que le nouvel angle entre les trois derniers points de **`I_sommets`**. Cette étape se fait en boucle tant que l'angle considéré n'est pas convexe ET que la taille de **`I_sommets`** > 3.

	<p>On cherche le point dont l'ordonnée est la plus basse, il sera le point 0.</p> <p>Puis on trie les points par angle croissant par rapport à l'axe des abscisses du point 0.</p>
	<p>On commence à remplir le tableau des sommets avec :</p> <p>Sommets = [0 1 2]</p> <p>L'angle (0,1,2) est convexe donc on peut garder le point 1.</p>
	<p>On ajoute le point 3.</p> <p>L'angle formé par (1,2,3) est concave : le point du milieu (n°2) est donc supprimé de la liste des sommets. Il faut revenir en arrière et considérer l'angle formé par : (0, 1, 3).</p> <p>Sommets = [0 1 3]</p>

	<p>Après test de l'angle formé par (0, 1, 3), il est convexe, donc le point du milieu ($n^{\circ}1$) est définitivement gardé dans la liste des sommets.</p> <p>Sommets = [0 1 3]</p>
	<p>On ajoute le point 4. L'angle (1, 3, 4) est convexe donc on peut garder le point 3.</p> <p>Sommets = [0 1 3 4]</p>
	<p>On pourrait ajouter une dernière vérification : Regarder si l'angle (3, 4, 0) est convexe. On pourra garder le point 4.</p> <p>Sommets = [0 1 3 4]</p>
	<p>La liste des sommets étant complète, il ne reste plus qu'à tracer le polygone associé grâce à <i>drawpoly</i> !</p>

Cette schématisation peut être généralisée pour N points. Voici le résultat pour N=40 :

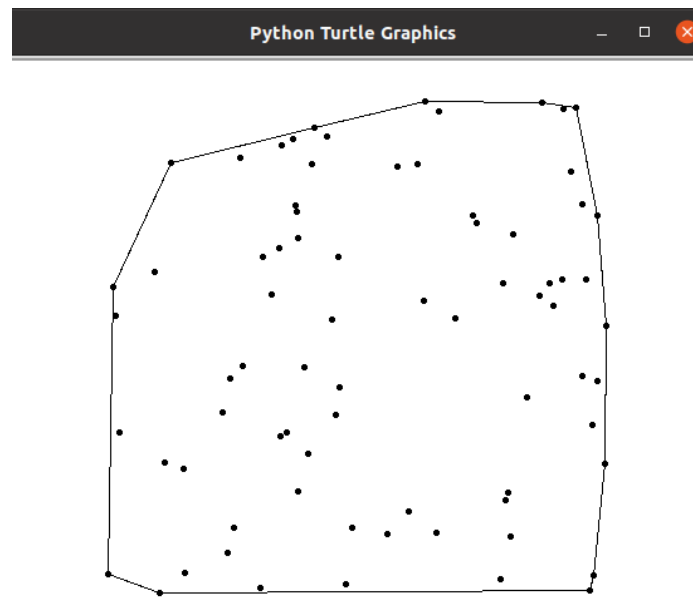


Schéma 4 :

Question 2 :

Calcul de la complexité de `convexhull(l)` :

On a N points dans le nuage de points **l**.

Dans le meilleur des cas, le polygone le plus petit est composé de trois sommets (triangle). Dans le pire des cas, on passe dans la boucle `for` (N-1) fois, vu que l'on ajoute les deux premiers points avant les boucles).

Ensuite, on passe dans la boucle `for` (N) fois.

Ainsi, dans les deux boucles, la complexité est de : $N + N-1 = 2N-1 = O(N)$.

Néanmoins, nous utilisons la fonction `sorted()`, qui est de complexité $O(N \ln(N))$ d'après l'énoncé.

En conclusion, la complexité de la fonction `convexhull(l)` est de : **$O(N \ln(N))$** .

Question 3 :

Pour créer `drawconvexhull(l)`, on se base sur la fonction `convexhull()` en rajoutant des commandes turtle. En effet, on commence, par sécurité à mettre la commande `penup()` au début, et à la fin de la fonction. Ensuite, le curseur se déplace grâce à `goto()` vers le tout premier sommet. À ce moment-là, on commence à tracer des traits avec `pendown()`.

Par la suite, on se déplace aux éléments de **l_sommets** voulus dans la boucle `for`; et si l'angle n'est pas convexe, on supprime les deux derniers traits effectués avec `undo()`.

Enfin, pour plus de visibilité, on a mis des temps de pause entre chaque dessin de trait avec `sleep()`.

Question 4 :

Dans le cas de mon ordinateur, pour obtenir un temps d'exécution de l'ordre d'une seconde, il faut prendre : $N = 250\,000$.

Exercice 3 : Diamètre en temps linéaire

Question 1 :

Soit un polygone P à n sommets qui ne contient aucun côté parallèle. On doit montrer qu'il a n paires de points antipodaux.
Si on tourne de 180° au total, on va forcément passer sur chaque point une fois, donc on va obtenir autant de paires que de points, soit : n .

Question 2 :

D'après l'animation de l'algorithme, la première paire de points antipodaux du polygone est celle composée des points d'ordonnées minimum et maximum.
La fonction *AntipodalPair()* doit renvoyer les indices respectifs de ces deux points extrêmes dans la liste I . Pour ce faire, on utilise une boucle *for*, qui parcourt toute la liste en comparant toutes les ordonnées pour en extraire les positions de y_{\min} et y_{\max} dans I .

Question 3 :

La fonction *NextAntipodalPair()* a pour entrées une paire de points antipodaux (i, j) , et la liste I .

-Première étape : Il faut trouver les vecteurs associés aux étriers.

-Deuxième étape : on cherche les angles (θ_i, θ_j) grâce à une fonction *angle* qui renvoie l'angle entre le vecteur de l'étrier et celui qui va du point i à $i-1$ (cf. schéma). Pour rappel, l'angle entre deux vecteurs est donné par la relation suivante :

$$\cos \theta = \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\| \cdot \|\vec{v}\|}$$

Si $\theta_i < \theta_j$: la fonction renvoie la prochaine paire de points antipodaux $(i-1, j)$.
Sinon, $\theta_i > \theta_j$ et la prochaine paire de points antipodaux est $(i, j-1)$.

La fonction est bien de complexité $O(1)$ car il n'y a pas de boucle *for* pour parcourir I .

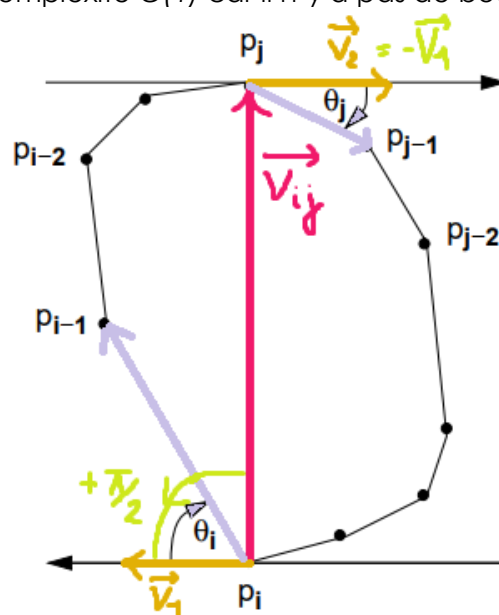


Schéma 5 :

Question 4 :

Pour la fonction *Diameter()*, on commence par créer une liste ***I_antipodaux*** que l'on remplit avec toutes les paires de points antipodaux. Pour sa création, on utilise une première fois la fonction *AntipodalPair()*, puis, $(n-1)$ -fois la fonction *NextAntipodalPair()* (avec n le nombre de sommets du polygone). On aura donc un total de n -paires de points antipodaux pour n sommets (vu à la question 1).

Par la suite, on crée la liste ***I_antipodaux_distances*** qui regroupe toutes les distances entre deux points d'une paire antipodale.

Enfin, on renvoie la valeur maximale de ***I_antipodaux_distances***.

Question 5 :

Pour la fonction *drawdiameter*, on se base sur la fonction *diameter* en rajoutant des commandes turtle. À chaque étape, on dessine ce qu'il se passe. Il ne faut pas oublier d'ajouter un moyen de retrouver la paire de points antipodaux dont la distance est la plus grande. Pour ce faire, on utilise une boucle *for*, qui parcourt toute la liste en comparant toutes les distances pour en extraire sa position dans la liste. En effet, sachant que les deux listes ***I_antipodaux*** et ***I_antipodaux_distances*** sont associées, on retrouve facilement la position de la paire à partir de la distance.

Question 6 :

Dans le cas de mon ordinateur, pour obtenir un temps d'exécution de l'ordre d'une seconde, il faut prendre : $N = 300\,000$.

Question 7 :

On prend donc : $N = 100$. Sachant que le nuage de points est quelconque, il faut reprendre depuis le début en appliquant *convexhull()*, de complexité $O(N \ln(N))$.

Ensuite, on utilise *AntipodalPair()* de complexité $O(N)$.

Puis, $(N-1)$ -fois *NextAntipodalPair()* de complexité $O(1) \Rightarrow$ ce qui donne $O(N-1) = O(N)$.

Enfin, on utilise une boucle *for* de complexité $O(N)$.

En conclusion, à partir d'un nuage de points quelconque, en utilisant *convexhull()* puis *diameter()*, la complexité totale est de : **$O(N \ln(N))$** .

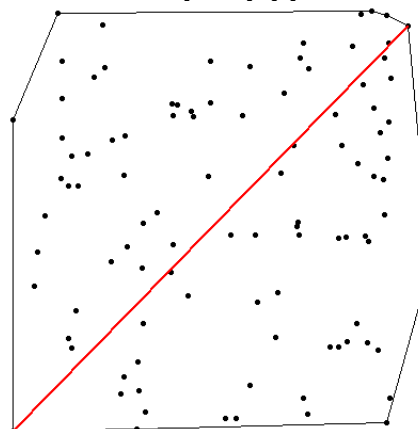


Schéma 6 :