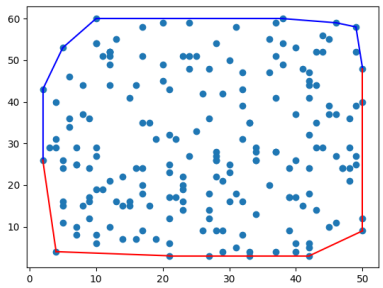
Projet 1 : **Étude d'un ensemble forestier.**

Dans ce projet nous nous proposons de mettre en oeuvre trois algorithmes géométriques (dont deux classiques) portant sur un ensemble de points du plan.

On suppose donné un ensemble d'arbre (nos points) sur lequel on souhaite faire trois choses :

- Déterminer la plus grande lignée d'arbres (pour y faire longer un canal par exemple),
- Déterminer la plus petite clôture entourant cet ensemble d'arbre (et calculer sa longueur),
- Déterminer (de manière efficace) le diamètre de cet ensemble (déterminer la plus grande distance séparant deux arbres).

Le deuxième algorithme est ce qu'on appelle la recherche de l'enveloppe convexe d'un ensemble de points : c'est le plus petit polygone convexe contenant les points (un polygone est dit convexe s'il contient **tout** segment reliant deux points situés à l'intérieur du polygone).



On aura besoin pour cela de trier les points par abscisses croissantes suivant l'ordre appelé ordre lexicographique :

$$(x, y) \leqslant (x', y') \Leftrightarrow x < x' \text{ ou } (x = x' \text{ et } y \leqslant y')$$

Pour cela, on pourra utiliser l'algorithme de tri utilisant un arbre binaire minimal (ce qui enrichira le projet).

Le calcul du diamètre se fera en se limitant aux points de l'enveloppe convexe d'une façon optimale.
On pourrait imaginer tester la distance séparant tout couple de points mais, s'il y avait n points, on aurait de l'ordre de n^2 calculs de distances à faire. Cela peut être envisageable si $n \leqslant 1000$ mais ne devient plus raisonnable si $n \geqslant 10000$.
Notre algorithme permettra d'effectuer seulement de l'ordre de n calculs pour calculer ce diamètre.

Exercice 1 :
1) Créer une fonction `Genere_Points` prenant en entrée `nb_Points, x, y, X, Y` où `nb_Points` est le nombre de points (nombre d'arbres) au plus que l'on veut, `x, y` sont respectivement la plus petite valeur possible des abscisses et la plus petite valeur possible des ordonnées tandis que `X, Y` sont respectivement la plus grande valeur possible des abscisses et la plus grande valeur possible des ordonnées.
On pourra supposer que `x, y, X, Y` sont des entiers.

Cette fonction doit générer une liste `Points` de `nb_Points` points choisis au hasard et donnés par leurs couples de coordonnées (a, b)
(où $a, b \in \mathbb{Z}$ et `x`$\leqslant a \leqslant$`X`, `y`$\leqslant b \leqslant$`Y`).
Avant de retourner la liste `Points`, ajouter ces quelques lignes à la fin de votre fonction (en indentant convenablement ces lignes) :

```
Points_bis = list(Points)
Points_bis.sort()
Points = [Points_bis[0]]
for k in range(1,nb_Points):
    if Points_bis[k] != Points_bis[k-1]:
        Points.append(Points_bis[k])
```

Expliquer le rôle de ces dernières instructions : quelle est alors la particularité de la liste de points donnée en sortie ?
2) Adapter la fonction précédente pour générer autrement les points, plutôt "sous forme d'un disque". On appellera cette nouvelle fonction `Genere_Points_Bis` qui prendra en arguments `nb_Points, R, n`. La variable `R` sera un réel strictement positif (le rayon du disque contenant les points) et `n` sera un entier naturel.
Principe : on créera chaque point en choisissant un entier naturel r au hasard entre 0 et `R`, ainsi qu'un entier m entre 0 et `n` : le point correspondant sera celui de coordonnées $(r\cos(m), r\sin(m))$.

Astuce

Une fois la liste `Points`$= [(x_0, y_0), (x_1, y_1), \ldots, (x_p, y_p)]$ créée, vous pouvez représenter les points graphiquement.
On peut utiliser `plt.scatter(X,Y)` (suivi de `plt.show()`) où `X` est la liste des abscisses (ici c'est $[x_0, x_1, \ldots, x_p]$) et `Y` la liste des ordonnées (ici c'est $[y_0, y_1, \ldots, y_p]$).
Pour récupérer la liste des abscisses et la liste des ordonnées à partir de la liste $[(x_0, y_0), (x_1, y_1), \ldots, (x_p, y_p)]$, on peut utiliser la commande `zip(*...)`. Voyez donc ce que sont a et b après les lignes d'instructions suivantes et généralisez.

```
a,b=zip(*[(1,2),(10,15)])
```

Notre second exercice a pour but d'atteindre notre premier objectif.
Une première idée pourrait être de coder ceci :

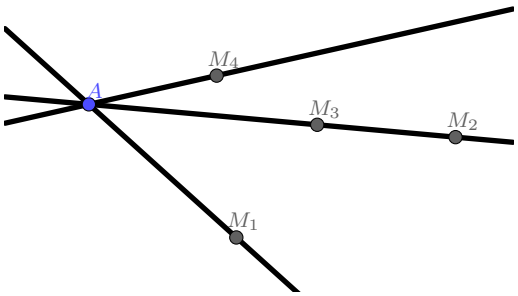
```
Pour chaque arbre A
   Pour chaque autre arbre B
      nbAlignement = 2
      Pour chaque arbre C
         Si A,B,C alignés,
            nbAlignement+=1
      maxAlignement = max(maxAlignement, nbAlignement)
```

Pseudo-code 1

Mais cela ne va pas s'avérer efficace (à peu près n^3 opérations pour n arbres (ou n points)).

L'idée va plutôt être la suivante : ordonner (efficacement) les points suivant leur position par rapport à un point de référence A.
Si A est un point donné, et si M_1 et M_2 sont deux autres points, on écrira $M_2 \succcurlyeq M_1$ ssi l'angle $\overline{(\overrightarrow{AM_1}, \overrightarrow{AM_2})}$ est positif (modulo 2π).
(et $M_2 \preccurlyeq M_1$ ssi l'angle $\overline{(\overrightarrow{AM_1}, \overrightarrow{AM_2})}$ est négatif)
M_1 et M_2 seront dits équivalents ssi l'angle orienté $\overline{(\overrightarrow{AM_1}, \overrightarrow{AM_2})}$ est nul.
On dit que la relation $\preccurlyeq$ est un *préordre* (total).

Si les points $M_1,\ M_2, \ldots, M_n$ sont ordonnés suivant cette relation de préordre $\preccurlyeq$, alors des points M_i appartenant à une même droite passant par A (points équivalents) seront côte à côte dans cette séquence.



Ici $M_1 \preccurlyeq M_2$, les points M_2 et M_3 sont équivalents et $M_3 \preccurlyeq M_4$.

Le pseudo-code serait alors :

```
Pour chaque arbre A
   Trier la liste des autres arbres B
      Trouver la longueur maximale des séquences de points équivalents pour le point A
Renvoyer la longueur maximale et une séquence de longueur maximale
```

Si la méthode de tri utilisée est efficace (ce sera notre cas : on utilisera la fonction `sorted` de Python), l'algorithme sera plus rapide que celui du pseudo-code 1, page 3.
On démontrera ultérieurement que l'angle $\overline{(\overrightarrow{AM_1}, \overrightarrow{AM_2})}$ est strictement positif ssi $\det(\overrightarrow{AM_1}, \overrightarrow{AM_2}) > 0$ (il est nul ssi $\det(\overrightarrow{AM_1}, \overrightarrow{AM_2}) = 0$ et strictement négatif ssi $\det(\overrightarrow{AM_1}, \overrightarrow{AM_2}) < 0$).

Ainsi, $M_2 \succcurlyeq M_1$ ssi $\det(\overrightarrow{AM_1}, \overrightarrow{AM_2}) \geqslant 0$ ce qui offre un moyen pratique de tester comment s'ordonnent deux points donnés M_1 et M_2 pour le préordre $\preccurlyeq$ relatif au point A.

Exercice 2 :

1) Écrire une fonction Python qui calcule le produit scalaire de deux vecteurs donnés en entrée. Écrire de même une fonction Python calculant la norme euclidienne d'un vecteur donné en entrée.

2) Écrire une fonction Python calculant le déterminant de deux vecteurs donnés en entrée.

3) Écrire une fonction Python prenant en entrée deux points $p = (x, y)$ et $q = (x', y')$ et donnant en sortie le couple des coordonnées du vecteur $\overrightarrow{pq}$.

4) Écrire une fonction Python `Tri_points` prenant en entrée deux arguments : une liste de points $[M_1, M_2, \ldots, M_n]$ (donnés par leurs couples de coordonnées) et un point A.
Cette fonction devra donner en sortie la liste des points M_i triée par ordre croissant suivant le préordre $\preccurlyeq$ relatif au point A (et décrit juste avant cet exercice 2).

Astuce

Pour trier une liste `L` d'objets selon une relation de préordre $\preccurlyeq$ personnalisée sur ces objets, on aura d'abord à utiliser la fonction `cmp_to_key` du module `functools` qu'on importera de cette manière :

```
from functools import cmp_to_key
```

On utilise ensuite la syntaxe `sorted(L,key=cmp_to_key(fonction_comparaison))` où `fonction_comparaison` est une fonction prenant en entrée deux arguments x et y (dans cet ordre). Cette fonction a pour but de définir la relation de préordre $\preccurlyeq$ existant sur x et y. Elle doit donner en sortie un nombre réel. Si ce nombre est positif cela signifie $x \succcurlyeq y$, s'il est négatif, cela signifie $x \preccurlyeq y$.

Exemple : Trions une liste de mots (sans accent, en minuscules) suivant le nombre de voyelles qu'ils comportent (on les trie par nombre de voyelles croissant).
D'abord, nous créons la fonction de comparaison entre `mot1` et `mot2`. Puisque nous souhaitons que `mot1`$\succcurlyeq$`mot2` si et seulement si `mot1` a plus de voyelles que `mot2`, notre fonction de comparaison va donner en sortie la différence du nombre de voyelle(s) de `mot1` et du nombre de voyelle(s) de `mot2`.
Il suffit alors d'utiliser la fonction `sorted` avec cette fonction de comparaison pour définir convenablement la variable de sortie.

```python
from functools import cmp_to_key

def Tri(Liste_mots):

    def nombre_voyelles(mot):
        voyelles = ["a","e","i","o","u","y"]
        nb_voyelles = 0
        for car in voyelles:
            nb_voyelles += mot.count(car)
        return nb_voyelles

    def fct_compare(mot1, mot2):
        return nombre_voyelles(mot1)-nombre_voyelles(mot2)

    Liste_mots_triee = sorted(Liste_mots, key = cmp_to_key(fct_compare))
    return Liste_mots_triee

# Exemple d'exécution:
L=["abreuvoir","charme","trop","prairie","frime","troupe"]
Tri(L)
['trop', 'charme', 'frime', 'troupe', 'prairie', 'abreuvoir']
```

5) En déduire une fonction Python prenant en entrée la liste `Points` des points et donnant en sortie le nombre maximum de points alignés ainsi qu'une liste de tels points (sous forme de couples de coordonnées). Cette fonction devra également représenter le nuage de points (cf **Astuce** de l'exercice 1) ainsi qu'un segment joignant une plus grande série de points alignés.

Indication :

Comme cela a déjà été expliqué, on triera les points suivant le préordre $\preccurlyeq$ relatif au point A et on détectera la plus grande séquence de points équivalents (pour tous les points A).

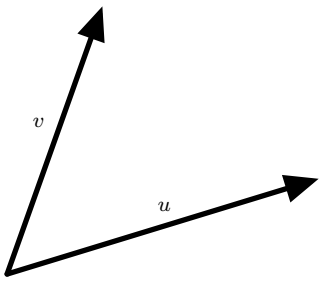
Exercice 3 : *Construction de la face "nord" et "sud" de l'enveloppe convexe (début)*

Les deux premières questions sont mathématiques (de thème géométrique).

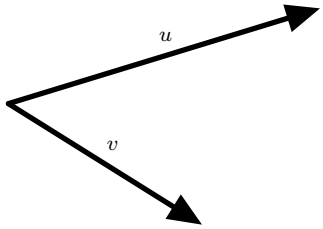
1) Soit $\overrightarrow{u}$, $\overrightarrow{v}$ deux vecteurs du plan donnés par leurs coordonnées $\overrightarrow{u} = (r_1\cos(a), r_1\sin(a))$ et $\overrightarrow{v} = (r_2\cos(b), r_2\sin(b))$ (où $r_1 \geqslant 0$, $r_2 \geqslant 0$).

Calculer leur déterminant $\det(\overrightarrow{u}, \overrightarrow{v})$ sous la forme $R\sin(\theta)$ où l'on identifiera $R \geqslant 0$ et θ en fonction de r_1, r_2, a et b.

2) En déduire que $\det(\overrightarrow{u}, \overrightarrow{v}) \geqslant 0$ si et seulement si l'angle orienté de $\overrightarrow{u}$ et $\overrightarrow{v}$ est positif ou nul (ou si $\overrightarrow{u} = \overrightarrow{0}$ ou $\overrightarrow{v} = \overrightarrow{0}$) et que $\det(\overrightarrow{u}, \overrightarrow{v}) \leqslant 0$ si et seulement si l'angle orienté de $\overrightarrow{u}$ et $\overrightarrow{v}$ est négatif ou nul (ou si $\overrightarrow{u} = \overrightarrow{0}$ ou $\overrightarrow{v} = \overrightarrow{0}$).



$\det(\overrightarrow{u}, \overrightarrow{v}) \geqslant 0$



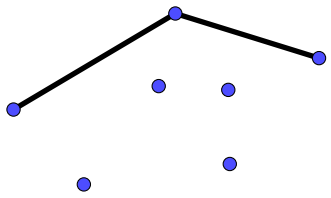
$\det(\overrightarrow{u}, \overrightarrow{v}) \leqslant 0$

3) Nous allons programmer dans cette question une fonction fondamentale : celle qui va déterminer le prochain point à insérer dans la liste des points d'une face de l'enveloppe convexe de notre nuage de points.

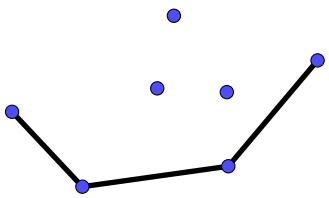
On construira en effet l'enveloppe convexe en deux temps : sa face "nord" puis sa face "sud".

On suppose que `Env_Convexe` est une liste de points de l'une des deux faces où les points sont supposés rangés par abscisses croissantes.

Dans le cas de la construction de la face nord, si M, N, P sont trois points consécutifs de cette face, l'angle orienté des deux vecteurs $\overrightarrow{MN}$ et $\overrightarrow{NP}$ doit être toujours négatif. Pour la face sud, l'angle orienté des vecteurs $\overrightarrow{MN}$ et $\overrightarrow{NP}$ doit être toujours positif (cf figure ci-dessous).

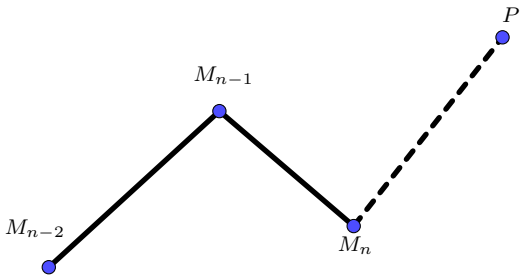


Face nord

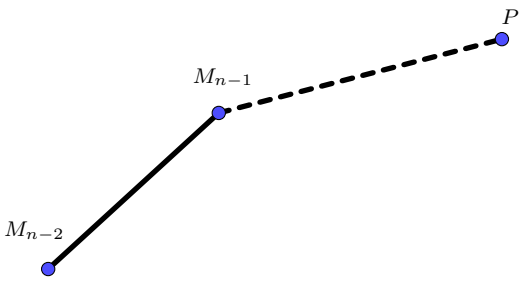


Face sud

Supposons maintenant que $M_0, M_1, \ldots, M_n$ sont les points (dans l'ordre) de la liste `Env_Convexe` en construction (avec $n \geqslant 1$). On se place dans le cas de la construction de la face nord. On suppose que P est le prochain point à insérer (d'abscisse supérieure à celle de N). Pour que P soit accepté dans la liste `Env_Convexe`, il faut et il suffit que l'angle orienté de $\overrightarrow{M_{n-1}M_n}$ et $\overrightarrow{M_nP}$ soit négatif. Si ce n'est pas le cas, on retire M_n (le dernier point de `Env_Convexe`), et on recommence le test avec les vecteurs $\overrightarrow{M_{n-2}M_{n-1}}$ et $\overrightarrow{M_{n-1}P}$, etc. jusqu'à ce que l'angle soit négatif ou qu'il y ait au plus un point dans `Env_Convexe` auquel cas on place P à la fin de la liste `Env_Convexe` (voir figure ci-dessous).



L'angle orienté de $\overrightarrow{M_{n-1}M_n}$ et $\overrightarrow{M_nP}$ est positif : on supprime M_n.



L'angle orienté de $\overrightarrow{M_{n-2}M_{n-1}}$ et $\overrightarrow{M_{n-1}P}$ est négatif : on ajoute P.

Écrire une fonction Python `ajouter_Point` prenant en entrée un entier `nord_sud` égal à 1 ou -1 et les coordonnées (x, y) d'un point P.
Cette fonction déclarera `Env_Convexe` comme variable globale et traduira le pseudo-code ci-dessous (cas de la face nord) :

```
Fonction ajouter_Point(nouveau_point)
Tant qu'il reste au moins deux points dans Env_Convexe
      Soit MN le vecteur joignant les deux derniers points
      Soit NP le vecteur joignant le dernier point au nouveau point
      Si l'angle orienté de MN et NP est strictement négatif (cas de la face nord)
         on peut ajouter le nouveau point P et on a terminé
      Sinon
         On retire le dernier point N de Env_Convexe
```

On reconnaît dans `Env_Convexe` une structure de Pile, c'est-à-dire une liste dans laquelle on fait des ajouts en fin de liste (`Env_Convexe.append(...)`) ou des retraits du dernier élément (`Env_Convexe.pop()`).
L'entier `nord_sud` sera égal à 1 si on construit la face nord et égal à -1 si on construit la face sud de l'enveloppe. On adaptera en fonction de la valeur de `nord_sud` le pseudo-code ci-dessus.
4) Créer une fonction `Trace_et_Longueur` prenant en entrée une liste $[M_0, M_1, \ldots, M_n]$ de points donnés par leurs couples de coordonnées (ainsi M_i sera en fait (x_i, y_i) où x_i, y_i sont l'abscisse et l'ordonnée de M_i) ainsi qu'une couleur ("black", "brown", "red", "blue", "green", etc.).
Cette fonction aura pour but de tracer les segments $[M_0, M_1]$, $[M_1, M_2]$, $\ldots$, $[M_{n-1}, M_n]$ dans la couleur indiquée en entrée, et devra donner en sortie la somme des longueurs de ces segments (ce qui correspond à la longueur de la ligne "brisée" reliant les $n+1$ points $M_0, M_1, \ldots, M_n$.).
Rappel : pour tracer par exemple en bleu le segment reliant les points $(2, 6)$ et $(4, 3)$, on écrira

```
1 import matplotlib.pyplot as plt
2
3 plt.plot((2,4),(6,3),color="blue")
```

Cette fonction pourra faire appel à la fonction de la question 3 de l'exercice 2 (calculant la norme d'un vecteur).

5) Dans cette question, on va construire de manière effective l'enveloppe convexe du nuage de point (la clôture du domaine forestier).
Pour cela, il faudra saisir aléatoirement les points, les ranger par abscisses croissantes, construire la face nord puis la face sud de l'enveloppe.
En effet, il faudra ranger les points par abscisses croissantes car, qu'il s'agisse de construire la face nord ou la face sud de l'enveloppe, nous regarderons toujours les points "de gauche à droite" et voir comment insérer le prochain point (cf fonction de la question 4).
Écrire une fonction `Construction_enveloppe` prenant en entrée une liste de points (couples de coordonnées) triée efficacement par abscisses croissantes (via une des fonctions de l'exercice 1 utilisant la méthode `sort()` ou avec le tri par arbre binaire vu en début d'année, ou avec le tri rapide.).
Cette fonction déclarera `Env_Convexe` comme variable globale.
Elle devra :

❶ afficher ces points sur un graphique,

❷ déterminer et tracer en bleu la face nord de l'enveloppe puis calculer sa longueur,

❸ déterminer et tracer en rouge la face sud de l'enveloppe puis calculer sa longueur,

❹ donner en sortie la longueur totale de l'enveloppe (somme des deux longueurs précédentes) ainsi que la liste des points de l'enveloppe **rangés dans le sens trigonométrique** : attention à la face nord !.

Pour effectuer les tâches ❷ et ❸, on pourra faire appel à la fonction de la question 5.
Pour chacune de ces deux tâches, on pourra initialiser la face de l'enveloppe comme une liste vide et, pour chaque point (points triés par abscisses croissantes), l'insérer dans la face de l'enveloppe par la fonction `ajouter_Point` (qui gèrera l'insertion de ce point, quitte à modifier la face de l'enveloppe en construction).

Remarques

Cet algorithme est l'un des nombreux qui existent permettant le calcul efficace de l'enveloppe convexe d'un ensemble fini de points du plan. Ce n'est pas le plus connu. Il est dû à A.M. Andrew qui a publié un article à ce sujet en 1979. En anglais, cet algorithme s'intitule "Andrew's monotone chain convex hull algorithm".

Exercice 4 :
Pour créer une animation, on importera les bibliothèques ci-dessous :

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
```

et on initialise une figure (fenêtre graphique où apparaîtra l'animation) comme ci-dessous :

```
fig = plt.figure()
```

Pour créer une animation, le principe est assez simple, on écrit les lignes de codes suivantes

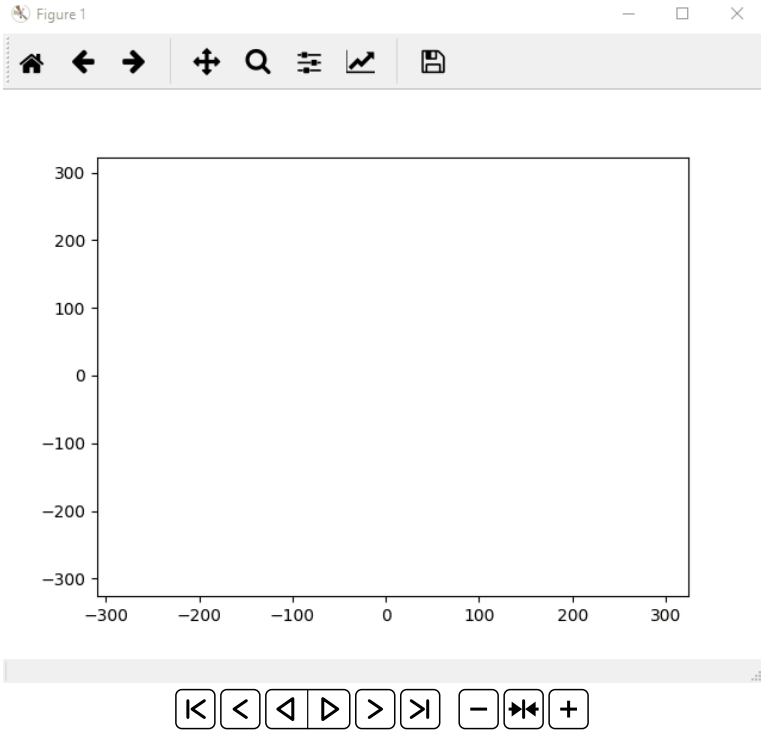
```
im_ani = animation.ArtistAnimation(fig, Diapos, interval=500, repeat_delay=1000,blit=True)
plt.show()
```

où `Diapos` est la liste de diapositives préalablement définie par l'utilisateur (chaque diapositive est une liste de graphiques créés avec `matplotlib.pyplot` devant figurer sur cette diapositive).
Ici `interval=100, repeat_delay=3000` signifie que l'intervalle de temps entre chaque diapositive doit être de 100 ms et que l'animation totale est répétée au bout de 3s = 3000ms à partir de la fin de celle-ci.
Si on ne veut pas répéter l'animation, on n'utilisera pas l'option `repeat_delay` et on précisera `repeat = False` dans les paramètres de `animation.ArtistAnimation`.
Voici un exemple minimal, mais complet, d'animation (il y a ici trois diapositives, chacune constituée de points (variables `a`) et/ou de segments de droites (variables `b`)).

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Initialisation de la figure et des diapositives
fig = plt.figure()
Diapos = []

# Définition de la diapo 1
a = plt.scatter([1,2],[7,4],c=["red","blue"])
b, = plt.plot([1,9],[2,3], color = "black")
Diapos.append([a,b])

# Définition de la diapo 2
a = plt.scatter([8,9],[2,5],c=["green","brown"])
b, = plt.plot([4,3],[2,1], color = "pink")
Diapos.append([a,b])

# Définition de la diapo 3
b, = plt.plot([1,2,3],[2,8,3],color="orange")
Diapos.append([a])

im_ani = animation.ArtistAnimation(fig, Diapos, interval=500, repeat_delay=1000,blit=True)
plt.show()
```

Essayer de retoucher les fonctions `Trace_et_Longueur` et `Construction_enveloppe` pour permettre une animation de la construction de la clôture (ces fonctions déclareront la variable `Diapos` comme variable globale : ce sera la liste des diapositives. Elles déclareront aussi globale une autre variable `image` qui va

stocker sous forme de liste les éléments à tracer dans chaque diapositive). Le résultat devra ressembler à ça (utilisé avec la génération des points sous forme de disque) :



Exercice 5 : Calcul et tracé du diamètre.

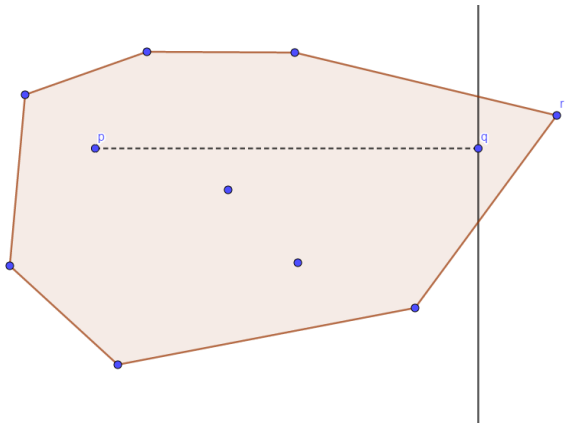
Préliminaire : un peu de théorie.
Le diamètre d'un nuage de points est la distance la plus importante entre deux points de ce nuage (c'est donc la distance entre deux points les plus éloignés de ce nuage).
Une première approche (raisonnable si l'on a affaire à moins de 1000 points) serait, si le nuage comporte n points, de tester les n^2 distances entre couples de points du nuage et de garder celle correspondant au maximum.
Cependant, dès que le nuage possède de l'ordre de 10000 points ou plus, cette méthode est trop lente.

On peut déjà remarquer que deux points les plus éloignées du nuage sont à chercher dans l'enveloppe convexe de ce nuage.
Autrement dit, la distance la plus importante entre deux points du nuage est égale à la distance maximale entre deux points de son enveloppe convexe.

Preuve

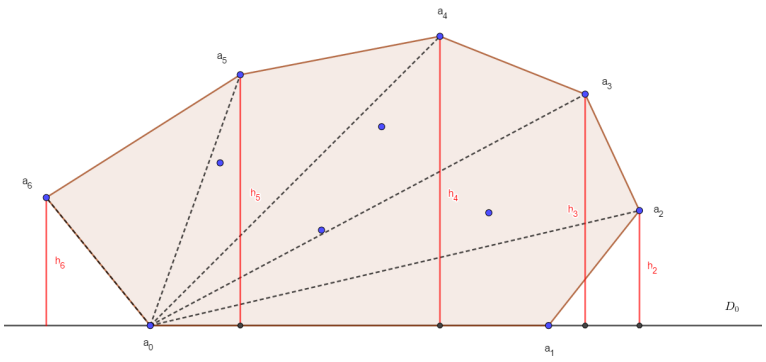
On rappelle que l'enveloppe convexe de l'ensemble de points est un polygone convexe. Le mot "convexe" signifie que, pour tout couple de points situés à l'intérieur du polygone, le segment qui joint ces deux points est encore dans le polygone.
Notons A l'ensemble des points du nuage et E l'ensemble des points de l'enveloppe convexe.
Donnons-nous deux points p et q de A tels que p ou q n'appartient pas à E. Par exemple, supposons $q \notin E$.
Nous allons montrer qu'il existe au moins un point r de E tel que la distance $d(p,r)$ de p à r est supérieure à la distance $d(p,q)$ de p à q.

On peut supposer sans perdre en généralité que le segment $[p,q]$ est horizontal et que p est à gauche de q (cf figure ci-dessous) :

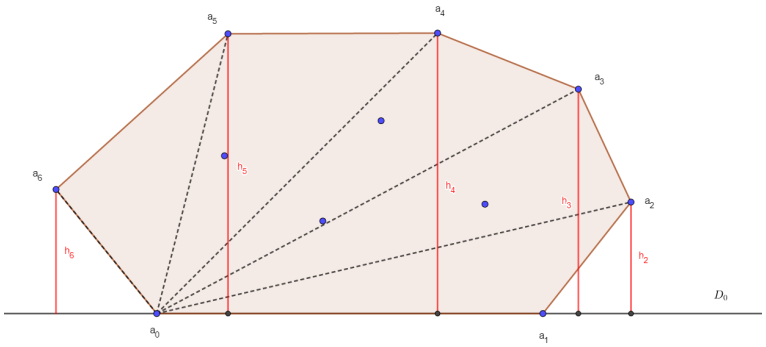


Puisque q n'appartient pas à E, il existe un sommet r de E qui est à droite ou sur la droite verticale passant par q (si tous les points de E étaient à gauche de cette droite verticale, le segment joignant p à q ne pourrait être contenu dans le polygone, ce qui contredit le fait qu'il est convexe.). On a alors facilement $d(p,q) < d(p,r)$.
Si $p \in E$, alors $(p,r) \in E^2$ et $d(p,q) < d(p,r)$. Si $p \notin E$, alors on trouverait de même un point $s \in E$ tel que $d(p,r) < d(s,r)$ et donc $d(p,q) < d(s,r)$ avec $(s,r) \in E^2$. Cela démontre le résultat (étant donnés deux points de A n'appartenant pas tous deux à E, on peut toujours trouver deux points de E séparés par une plus grande distance).

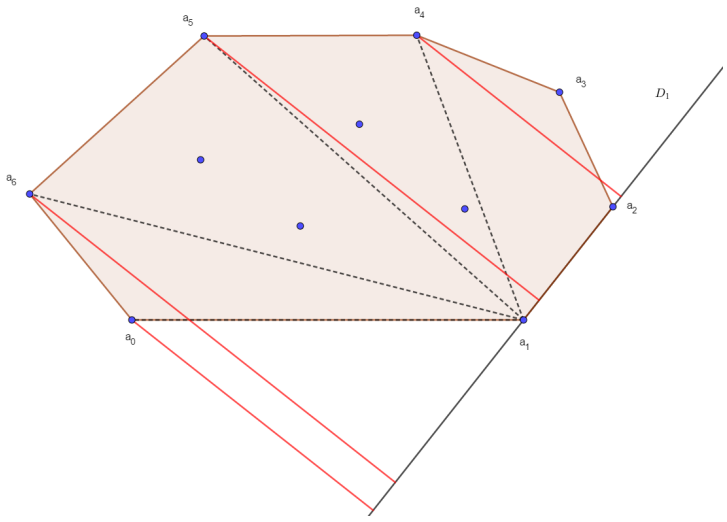
Il faut donc trouver un moyen efficace de calculer le diamètre de l'enveloppe convexe.
Notons $[a_0, a_1, \dots, a_k]$ les k points de l'enveloppe convexe rangés dans le sens trigonométrique. Notons M la distance maximale (en cours de calcul) entre deux points de cette enveloppe (M est initialisée à $-\infty$ où à 0).
On suppose qu'il n'y a pas plus de deux points alignés dans l'enveloppe convexe (réfléchir : si nécessaire, corriger un tout petit quelque chose dans la fonction `ajouter_Point` pour que ce soit bien le cas (question 4 de l'exercice 3)).
L'idée est la suivante : étant donnée la droite $D_0 = (a_0, a_1)$ de l'enveloppe convexe, on calcule les distance h_i de D_0 à a_i pour $i \in [\![2, n]\!]$. En raison du caractère convexe du polygone formant l'enveloppe convexe, ces distances $d_i = d(D_0, a_i)$ sont d'abord strictement croissantes puis décroissantes (voir dessin ci-dessous).



On a donc l'existence d'un i tel que $d(D_0, a_2) < d(D_0, a_3) < d(D_0, a_4) < \cdots < d(D_0, a_i)$ puis $d(D_0, a_k) < d(D_0, a_{k-1}) < \cdots < d(D_0, a_{i+1}) \leqslant d(D_0, a_i)$ (dans le dessin ci-dessus, $i = 4$).
Si $d(D_0, a_{i+1}) < d(D_0, a_i)$ (cas du dessin ci-dessus), on calcule le maximum des distances $d(a_0, a_i)$ et $d(a_1, a_i)$ (et on garde en mémoire le couple de points réalisant ce maximum), et on met à jour M comme étant égal à ce maximum (si M lui est inférieur).
Si $d(D_0, a_{i+1}) = d(D_0, a_i)$, on calcule le maximum des distances $d(a_0, a_i)$, $d(a_1, a_i)$, $d(a_0, a_{i+1})$ et $d(a_1, a_{i+1})$ (on garde en mémoire le couple de points réalisant ce maximum), et on met à jour M comme étant égal à ce maximum (si M lui est inférieur).
L'illustration ci-dessous correspond à un cas où $d(D_0, a_i) = d(D_0, a_{i+1})$ (avec $i = 4$), cela traduit que les droites $D_0 = (a_0, a_1)$ et (a_i, a_{i+1}) sont parallèles.



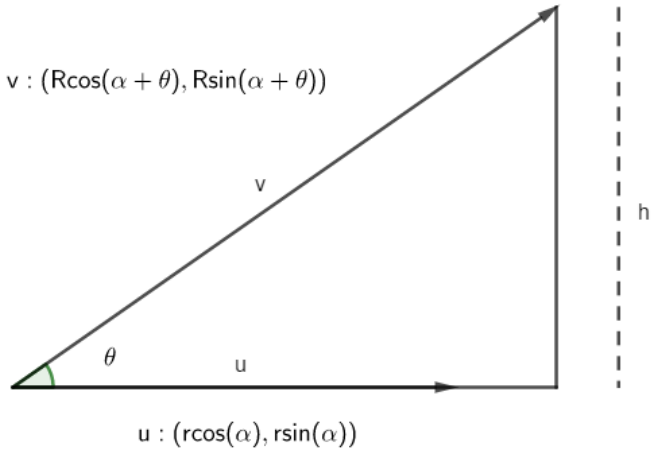
On recommence ensuite la procédure avec la prochaine droite $D_1 = (a_1, a_2)$ correspondant au prochain segment de l'enveloppe convexe (dans l'ordre trigonométrique) en calculant les distances $d(D_1, a_j)$ pour $j \in [\![i, n]\!] \cup \{0\}$ (**important :** on commence en effet par calculer $d(D_1, a_i)$ et non $d(D_1, a_3)$ car on peut montrer que les distances $d(D_1, a_j)$ pour $j \in [\![3, i-1]\!]$ sont plus petites que $d(D_1, a_i)$).



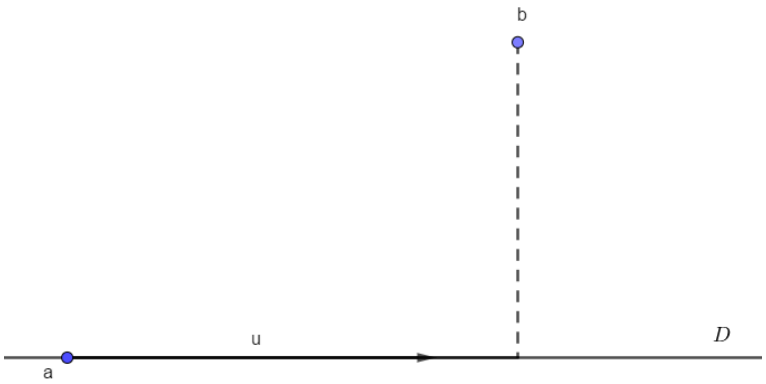
etc. jusqu'à $D_{k-1} = (a_{k-1}, a_k)$ et enfin $D_k = (a_k, a_0)$.
On peut montrer que l'on effectue alors de l'ordre de $4n$ opérations pour comparer les distances (au lieu de n^2 ce qui est un net progrès !).

1) Question mathématique :
L'objectif est d'avoir une formule commode pour calculer les distances $d(D_j, a_i)$ évoquées ci-dessus.

Soit $\overrightarrow{u}$ et $\overrightarrow{v}$ deux vecteurs (non nuls) de coordonnées polaires respectives $(r\cos(\alpha), r\sin(\alpha))$ et $(R\cos(\alpha+\theta), R\sin(\alpha+\theta))$, avec $\pi \geq \theta \geq 0$.



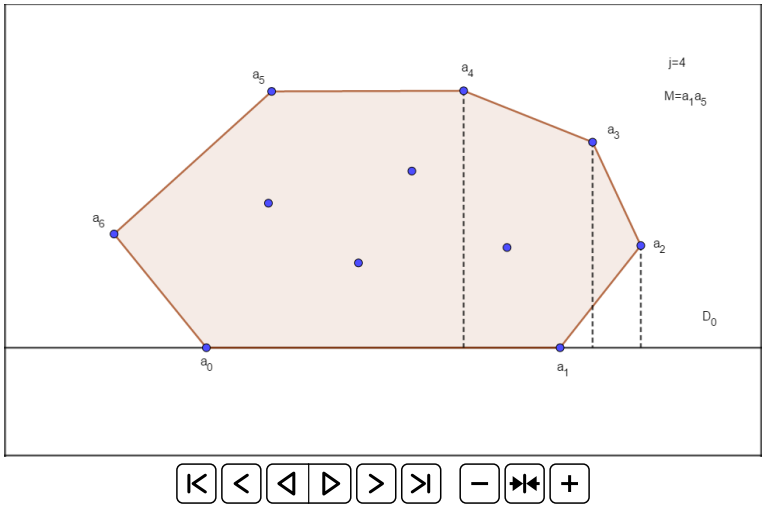
a) Rappeler la formule reliant h, R et $\sin\theta$ (question niveau collège).
b) Calculer $\det(\overrightarrow{u}, \overrightarrow{v})$ en fonction de r, R et $\sin\theta$. En déduire $h = \frac{\det(\overrightarrow{u},\overrightarrow{v})}{r}$.
c) Écrire alors une fonction Python `distance` prenant en entrée un vecteur $\overrightarrow{u}$ non nul (dirigeant une droite D), et deux points a, b du plan, et donnant en sortie la distance de la droite $D = a + Vect(\overrightarrow{u})$ au point b. On supposera que les vecteurs $\overrightarrow{u}$, $\overrightarrow{ab}$ sont orientés dans le sens trigonométrique (on "tourne" dans le sens trigonométrique pour aller de $\overrightarrow{u}$ à $\overrightarrow{ab}$), et que cet angle est inférieur ou égal à π.



2) On note $L = [a_0, a_1, \ldots, a_k]$ la liste des points de l'enveloppe convexe rangés dans le sens trigonométrique et on note m la taille de L (ici $m = k+1$).
Si i est un entier strictement positif, on écrira "i modulo m" pour désigner le reste de la division euclidienne de i par m (par exemple, (m modulo m) vaut 0, ($m+1$ modulo m) vaut 1, ($m+2$ modulo m) vaut 2, etc.).
Traduire alors le pseudo-code ci-dessous en fonction Python `Diametre` prenant en entrée L et donnant en sortie le diamètre M de l'enveloppe, ainsi qu'un couple (a, b) de deux points séparés d'une distance correspondant à ce diamètre (comprendre aussi ce pseudo-code avec les explications fournies ci-dessus).

Si m = 1, renvoyer M=0 et (a_0, a_0).
Soit $j = 2$
M = 0
Pour tout $i \in [\![0, k]\!]$
$\overrightarrow{u_i} = \overrightarrow{a_i a_{(i+1 \text{ modulo } m)}}$
Tant que `distance`$(a_i + Vect(\overrightarrow{u_i}), a_{(j+1 \text{ modulo } m)})$> `distance`$(a_i + Vect(\overrightarrow{u_i}), a_{(j \text{ modulo } m)})$
on augmente j d'une unité (on "tourne" dans le polygone dans le sens trigonométrique)
Si `distance`$(a_i + Vect(\overrightarrow{u_i}), a_{(j+1 \text{ modulo } m)})$= `distance`$(a_i + Vect(\overrightarrow{u_i}), a_{(j \text{ modulo } m)})$
Si $M < \max(\left\|\overrightarrow{a_i a_{(j \text{ modulo } m)}}\right\|, \left\|\overrightarrow{a_i a_{(j+1 \text{ modulo } m)}}\right\|, \left\|\overrightarrow{a_{(i+1 \text{ modulo } m)} a_{(j \text{ modulo } m)}}\right\|, \left\|\overrightarrow{a_{(i+1 \text{ modulo } m)} a_{(j+1 \text{ modulo } m)}}\right\|)$
on met à jour M et un couple (a, b) tel que $M = \left\|\overrightarrow{ab}\right\|$
sinon si $M < \max(\left\|\overrightarrow{a_i a_{(j \text{ modulo } m)}}\right\|, \left\|\overrightarrow{a_{(i+1 \text{ modulo } m)} a_{(j \text{ modulo } m)}}\right\|)$
on met à jour M et un couple (a, b) tel que $M = \left\|\overrightarrow{ab}\right\|$
renvoyer M et (a, b), et tracer le segment $[a, b]$.

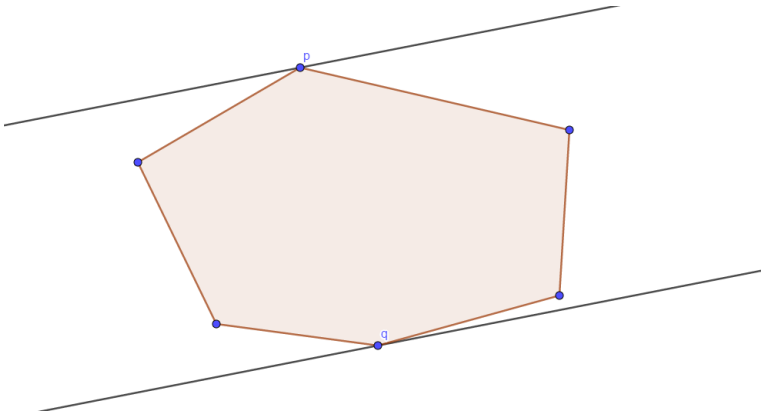
Illustration :



Remarques

- La distance `distance`$(a_i + Vect(\overrightarrow{u_i}), b)$ correspond à la longueur de la hauteur du triangle $(a_i a_{(i+1 \text{ modulo } m)} b)$ qui est donnée par $\frac{\det(\overrightarrow{u_i}, \overrightarrow{ab})}{\|\overrightarrow{u_i}\|}$ d'après la question 1) b) de cet exercice. Ainsi, dans la boucle "Pour tout $i \in [\![0, k]\!]$" du pseudo-code ci-dessus, on pourrait remplacer `distance`$(a_i + Vect(\overrightarrow{u_i}), b)$ par $\det(\overrightarrow{u_i}, \overrightarrow{ab})$ ce qui correspond géométriquement à deux fois l'aire du triangle $(a_i a_{(i+1 \text{ modulo } m)} b)$ (en effet, d'après cette aire est $\frac{\text{base} \times \text{hauteur}}{2}$ et ici base×hauteur= $\|\overrightarrow{u_i}\|$ ×hauteur= $\det(\overrightarrow{u_i}, \overrightarrow{ab})$).
- L'algorithme ci-dessus de calcul du diamètre d'un ensemble fini de points du plan est apparu, pour la première fois, dans la thèse de Michael Ian Shamos (1978) sous une forme plus complexe utilisant des mesures d'angles. Cet algorithme a ensuite été simplifié dans le livre du même auteur et de Franco P.Preparata "Computational Geometry, an introduction" (1985, Springer-Verlag).
  Il utilise le fait (que nous n'avons pas démontré) que deux sommets les plus éloignés d'un polygone convexe sont à chercher parmi les points antipodaux du polygone (deux points p, q d'un polygone convexe sont dits antipodaux s'il existe deux droites parallèles passant l'une par p, l'autre par q et

ne coupant le polygone qu'en ces points. Voir la figure ci-dessous).



Exercice 6 : *Synthèse*

Écrire une fonction Python (sans argument) demandant à l'utilisateur ce qu'il souhaite :

a) Créer un nuage de points (aléatoire rectangulaire ou sous forme de disque ou personnalisé) puis obtenir la plus longue séquence de points alignés (exercices 1 et 2).

b) Créer un nuage de points (aléatoire rectangulaire ou sous forme de disque ou personnalisé) puis obtenir son enveloppe convexe avec animation ou non (exercices 3 et 4).

c) Créer un nuage de points (aléatoire rectangulaire ou sous forme de disque ou personnalisé) puis obtenir son enveloppe convexe, son diamètre et le tracé d'un segment correspondant à ce diamètre (exercices 3, 4 et 5).

Cette fonction devra satisfaire les choix de l'utilisateur.

Pour vous aider, voici un extrait de code à tester (et qui attend une réponse de votre part : 0, 1 ou 2) :

```
L = ["bonjour","au revoir", "bonsoir"]
print("""
0 - Bonjour
1 - au revoir
2 - bonsoir
""")
a = int(input("Faites votre choix:"))
print(L[a])
```

Extensions conseillées :

Écrire les fonctions simples (mais non efficaces) répondant aux tâches des exercices 2 et 5. Autrement dit : écrire la fonction correspondant au pseudo-code 1, page 3 (pour la tâche de l'exercice 2) et au pseudo-code ci-dessous pour celle calculant le diamètre de la clôture :

```
Max = 0
Pour chaque arbre A
   Pour chaque arbre B
      d = AB
      si d > Max
         Max = d
         a, b = A, B
```

Comparer l'efficacité temporelle de ces deux fonctions avec celle des versions optimisées (programmées aux exercices 2 et 5).
On pourra pour cela tracer des diagrammes en barres correspondant aux moyennes des temps d'exécution des fonctions "naïves" et de leur version optimisée (avec 200 arbres pour la tâche de l'exercice 2 et 1000 arbres pour la tâche de l'exercice 5).
On rappelle que pour mesurer le temps d'exécution d'un programme, on peut utiliser le module `time` et sa fonction `clock`. Par exemple, le programme ci-dessous calcule le temps (en seconde(s)) que met Python à calculer les racines carrées des 1000000 premiers entiers naturels non nuls :

```python
from math import sqrt
from time import clock

top_depart = clock()
for k in range(1,10**6+1):
    a = sqrt(k)
top_fin = clock()
print("temps d'exécution:",top_fin-top_depart)
```

On pourra également faire une exploitation statistique de la taille du diamètre du nuage de points en fonction de $R > 0$ si le nuage est aléatoirement réparti dans un disque de rayon R.

Conseils pour ce projet et les questions du jury :
- être parfaitement au point sur les algorithmes de tri au programme (tri par insertion et tri rapide).
- être capable d'illustrer sur un exemple simple l'algorithme de la chaîne monotone d'Andrew, et celui de Shamos (calcul du diamètre).