Séance 15 :

Tri par insertion

Avant toute chose :

- Pour cette séance j'ai fait deux classes que j'ai construites au fur et à mesure des questions.
- Voici la strucutre de la séance :

```
>
  > ressources
    > ellipse.py
    > liste.py

  > main.py
```

  - `ressources/ellipse.py` contient la classe ellipse
  - `ressources/liste.py` contient la classe liste
  - `main.py` contient tout le code à exécuter

1. Début

(a) ***Ecrire cette procédure*** `tri_debut(A, i)` ***en utilisant la variable locale*** `V = A[i]` ***et la variable globale A.***

Du coup pour moi la variable globale `A` sur le paramètre `A` de l'objet `Ellipse`.

> ressources/liste.py

```python
# Imports ------------------------------------------------------------
import numpy as np
import matplotlib.pylab as plt
import time

# Classe Liste -------------------------------------------------------
class Liste(list):
    def __init__(self, liste=[]):
        self.A = liste

    def _triDebut(self, i):
        V = self.A[i]
        while _plusPetitQue(V, self.A[i-1]) and i-1 >= 0:
            self.A[i] = self.A[i-1]
            i -= 1
        self.A[i] = V

# Helpers ------------------------------------------------------------
def _plusPetitQue(a, b):
    if type(a) == list:
        return a[1] < b[1] or (a[1] == b[1] and a[0] < b[0])
    else:
        return a < b
```

2. Tri insertion

(a) ***Ecrire une procédure*** `tri_insertion` ***qui trie une liste*** `A` ***suivant la méthode précédente en parcourant la liste*** `A` ***. Tester avec :*** `A = [9, 11, 3, 7, 8, 6, 1, 3]`

```
> ressources/liste.py
class Liste(list):
    # ...
    def trier(self):
        for i in range(1, len(self.A)):
            self._triDebut(i)
```

```
> ressources/liste.py
# Imports -------------------------------------------------------------------
from ressources.liste import Liste
from ressources.ellipse import Ellipse
import numpy as np

# Question 2
    -----------------------------------------------------------------
A = Liste([9, 11, 3, 7, 8, 6, 1, 3])
A.trier()
print(A.A)
```

Ce qui retourne : `[1, 3, 3, 6, 7, 8, 9, 11]`

3. Complexité

(a) ***Montrer que la complexité du tri par insertion est de n^2 comparaisons dans le pire des cas.*** Il y a deux boucles imbriquées l'une dans l'autre. Dans le pire des cas, elles font chacunes n tours. Donc la complexité est n^2.

4. Vérification de la complexité

(a) ***Ecrire une fonction*** `test_tri(n)` ***qui :***

- ***Enregistre dans*** `Tab` ***un tableau de n nombre aléatoires compris entre 0 et n avec la fonction*** `randint` ***du module*** `numpy.random`
- ***Affiche les 6 premiers termes et les 6 derniers termes de*** `Tab`
- ***Trie*** `Tab` ***et donne la durée du tri en secondes***
- ***Affiche les 6 premiers termes et les 6 derniers termes de*** `Tab` ***trié***
- ***Retourne la durée du tri***

Avec cette fonction calculer la durée du tri pour $n \in \{10; 100; 200; 500; 1500\}$ et vérifier que la durée du tri est proportionnelle à n^2

Je commence par rajouter deux méthodes à la classe `Liste` :

- ❏ `generateRadomList` qui génère une liste de n entiers aléatoires
- ❏ `time` qui génère et trie des listes de taille allant de 1 à n, enregistre le temps, et le trace en fonction de n^2 si le paramètre `show` est passé à `True`.

Remarque : Je n'ai pas affiché les 6 premiers et derniers termes, mais il suffirait de rajouter les fonctions :

- `print(self.A[:6])` pour afficher les 6 premiers
- `print(self.A[-6:])` pour les 6 derniers

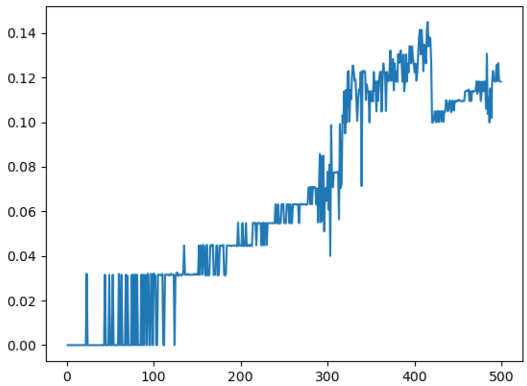
Ensuite, dans `main.py` j'exécute la fonction `time` sur l'objet `A`
Remarque : Mon programme teste n listes et non pas 5 comme demandé.

> ressources/liste.py

```python
class Liste(list):
    def trier(self):
        for i in range(1, len(self.A)):
            self._triDebut(i)

    def time(self, n, show=False):
        X, T = [], []
        for i in range(1, n):
            self.generateRadomList(i)
            start = time.time()
            self.trier()
            end = time.time()

            X.append(i)
            T.append(np.sqrt(end - start))
        if (show):
            plt.plot(X, T, label='$\sqrt{t}$')
            plt.show()
```

> ressources/liste.py

```python
A.time(500, show=True)
```



On a une mesure très bruitée, mais qui a plutôt tendance à être une droite, ce qui prouverait la complexité quadratique.

5. Application : Remplissage d'un ellipse

(a) On considère l'ellipse définit par le point $\overrightarrow{OM}(t) = (x(t), y(t) \in \mathbb{R}^2)$ où $\left\{ \begin{array}{lcl} x' & = & 2x - 5y \\ y' & = & x - 2y \end{array} \right.$ avec les conditions initiales $M(0) = (1, 2)$.
Tracer la courbe des points $M(t)$ ***pour*** $t \in [0, 2\pi]$ ***avec la méthode d'Euler en utilisant*** `N = 80` ***points.***
C'est là que je crée la classe `Ellipse` :

> ressources/ellipse.py

```python
# Imports -------------------------------------------------------------------
import numpy as np
import matplotlib.pylab as plt
from ressources.liste import Liste
import cmath
```

> ressources/ellipse.py

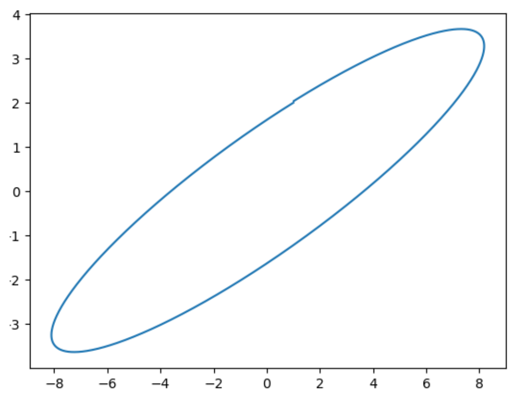
```
# Classe Ellipse ------------------------------------------------------------
class Ellipse():
    def __init__(self):
        self.X = []
        self.Y = []
        self.Filled = []
        self.conditionInitales = (0, 0)
        self.Eulerized = False

    def Eulerize(self, n):
        h = 2*np.pi/n
        self.X = [self.conditionInitales[0]]
        self.Y = [self.conditionInitales[1]]

        for i in range(n):
            self.X.append(self.X[i] + h * (2*self.X[i]-5*self.Y[i]))
            self.Y.append(self.Y[i] + h * (self.X[i]-2*self.Y[i]))
        self.Eulerized = True

    def Tracer(self):
        if (self.Eulerized):
            l = len(self.Filled)
            if (l > 0):
                colonne = int(np.sqrt(l))
                ligne = l // colonne
                ligne = ligne if l % colonne == 0 else ligne + 1
                for i in range(l):
                    plt.subplot(colonne, ligne, i+1)
                    plt.plot(self.Filled[i][0], self.Filled[i][1])
                    plt.plot(self.X, self.Y)
            else:
                plt.plot(self.X, self.Y)
            plt.show()
        else:
            raise NameError('Veuillez d\'abord lancer la procedeure d\'Euler')
```

Et je teste tout ça :

> main.py

```
E = Ellipse()
E.conditionInitales = (1, 2)
E.Eulerize(1000)
E.Tracer()
```

Pour obtenir :



(b) ***Trier tous les points de la courbe par abscisse croissante de manière à obtenir un quadrillage vertical de la courbe.*** Aux vues des questions suivantes, je fais tous les remplissages d'un coup :

> ressources/ellipse.py

```python
class Ellipse():
    def GetListOfPoint(self, n, inverted=False, radial=False):
        A = []
        l = len(self.X)
        for i in range(0, l, l//n):
            if inverted:
                A.append([self.Y[i], self.X[i]])
            else:
                A.append([self.X[i], self.Y[i]])
        return A

    def Fill(self, pas, methode):
        if (methode == 0):
            X, Y = [0], [0]
        elif (methode == 1):
            A = Liste(self.GetListOfPoint(pas))
            A.trier()
            X, Y = A.separate()
        elif (methode == 2):
            A = Liste(self.GetListOfPoint(pas, True))
            A.trier()
            Y, X = A.separate()
        elif (methode == 3):
            A = Liste()
            for i in range(0, len(self.X), pas):
                r, tetha = cmath.polar(complex(self.X[i], self.Y[i]))
                A.append([tetha, r])
            A.trier()
            X, Y = [], []
            for a in A:
                X.append(a[1] * np.cos(a[0]))
                Y.append(a[1] * np.sin(a[0]))
                X.append(0)
                Y.append(0)
        self.Filled.append([X, Y])
```

Du coup je crée la mathode `separate` de la classe `Liste` :

> ressources/liste.py

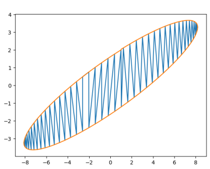
```python
class Liste(list):
    def separate(self):
        X, Y = [], []
        for a in self.A:
            X.append(a[0])
            Y.append(a[1])
        return (X, Y)
```

Je trace tout ça :

> main.py

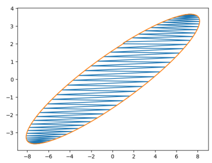
```python
E = Ellipse()
E.conditionInitales = (1, 2)
E.Eulerize(1000)
E.Fill(80, 2)
E.Tracer()
```

Et j'obtiens :



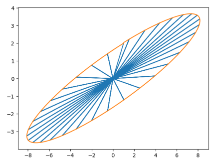
(c) ***Trier tous les points de la courbe par ordonnée croissante de manière à obtenir un quadrillage horizontal de la courbe.***

```
> main.py
```
```python
E.Fill(80, 1)
E.Tracer()
```



(d) ***Trier tous les points de la courbe de manière à obtenir un quadrillage radial de la courbe.***

```
> main.py
```
```python
E.Fill(30, 3)
E.Tracer()
```



6. Rotation et mesure de l'ellipse ε

(a) ***Déterminer la longueur du grand rayon a de l'ellipse et son inclinaison θ_i sur l'axe des x.***

```
> ressources/ellipse.py
```
```python
class Ellipse():
    def GetPolar(self):
        rMax = 0
        phiMax = 0
        for i in range(len(self.X)):
            r, phi = cmath.polar(complex(self.X[i], self.Y[i]))
            if r > rMax:
                rMax = r
                phiMax = phi
        return (rMax, phiMax)

  def GetGrandRayon(self):
        X = np.array(self.X)
        Y = np.array(self.Y)
        return(max(np.sqrt(X**2 + Y**2)))
```

```
> main.py
```
```python
a = E.GetGrandRayon()
r, t = E.GetPolar()
print(a, t*180/np.pi)
```

J'obtiens :

$$a = \boxed{8.839835695792114}$$
$$\theta_i = \boxed{22.555972447051808}^\circ$$

(b) ***A l'aide d'une matrice de rotation, faites tourner le dessin de l'ellipse de $-\theta_i$ pour trouver une ellipse horizontale. Donner alors la longueur du petit rayon b de l'ellipse.***

> ressources/ellipse.py

```python
class Ellipse():
    def Rotate(self, t):
        X, Y = rotate(self.X, self.Y, t)
        self.Filled.append([X, Y])

    def GetPetitRayon(self, t):
        X, Y = rotate(self.X, self.Y, t)
        return(min(np.sqrt(X**2 + Y**2)))

# Helpers ------------------------------------------------------------------


def rotate(Xa, Ya, t):
    R = np.array([
        [np.cos(t), -np.sin(t)],
        [np.sin(t), np.cos(t)]
    ])
    X, Y = [], []
    for i in range(len(Xa)):
        u = np.array([Xa[i], Ya[i]])
        v = np.dot(R, u)
        X.append(v[0])
        Y.append(v[1])
    return np.array(X), np.array(Y)
```

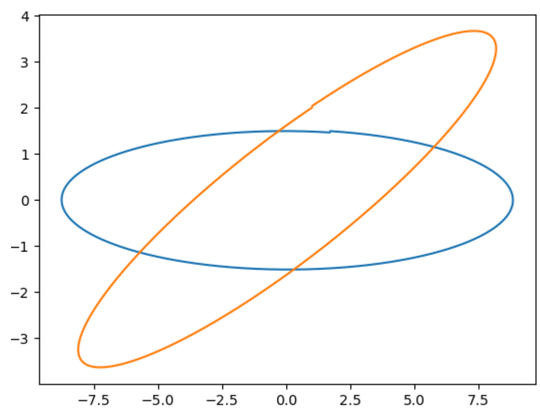
> main.py

```python
r, t = E.GetPolar()
b = E.GetPetitRayon(-t)

E.Rotate(-t)
E.Tracer()


print(b)
```

J'obtiens :

$$b = \texttt{1.4943892268344021}$$

En orange, l'ellipse originale, et en bleu l'ellipse tournée de $-\theta_i$:



(c) ***Après rotation de ε et en la supposant symétrique par rapport à l'axe des x, calculer l'aire contenue dans l'ellipse ε et vérifier que l'erreur avec la formule théorique est de 0.4 % pour*** `N = 1 000` ***points.***

> ressources/ellipse.py

```
class Ellipse():
    def GetAire(self, t):
        A = []
        aire = 0
        R = np.array([
            [np.cos(t), -np.sin(t)],
            [np.sin(t), np.cos(t)]
        ])
        for i in range(len(self.X)):
            u = np.array([self.X[i], self.Y[i]])
            v = np.dot(R, u)
            if(v[0] > 0 and v[1] > 0):
                A.append(v)

        for i in range(len(A)-1):
            aire += A[i+1][1] * abs(A[i+1][0] - A[i][0])
        return aire * 4
```

> main.py

```
A1 = E.GetAire(-t)
A2 = np.pi*a*b

print(abs(A2-A1)/A2*100)
```

J'obtiens :

$$r = \texttt{0.047054748966392786}\%$$

<u>Remarque</u> : J'obtiens bien une erreur de 0.4% pour `N = 10 000` et non `N = 1 000`...