TP3 POO - Python

simon.girel@univ-cotedazur.fr, gilles.scarella@univ-cotedazur.fr

1 Matrices avec numpy

Le code suivant doit être écrit dans le fichier *exo1.py*.

On considère la matrice 4x4 suivante, définie pour tout $k \in \mathbb{N}$

$$\begin{pmatrix} 2k+1 & -5 & 0 & k \\ -5 & 3k+2 & -1 & 3 \\ 0 & -1 & 3k-2 & 1 \\ k & 3 & 1 & 7k-3 \end{pmatrix}$$

- Dans *exo1.py*, écrire une fonction *matM* qui prend un argument k et qui retourne la matrice précédente
- Dans *exo1.py*, écrire une fonction *matN* prenant trois arguments n, k_0 et j_0 (ce sont des entiers). On suppose que n est un multiple de 4.
  matN doit retourner une matrice carrée d'ordre n, initialisée à 0, telle que:
  - elle contient sur sa diagonale *matM(k/4)* pour k=0 à $(n-1)$ par pas de 4 (on raisonne par blocs de matrice).
  - la ligne d'indice k_0 doit contenir la valeur 8 pour les colonnes se trouvant après celles modifiées par l'opération précédente (utilisation de matM).
  - la colonne d'indice j_0 doit contenir la valeur 6 pour les lignes se trouvant après celles modifiées par l'utilisation de matM.

Par exemple, pour $n = 8$, $k_0 = 1$, $j_0 = 2$, on obtient la matrice N suivante:

```
[[ 1 -5  0  0  0  0  0  0]
 [-5  2 -1  3  8  8  8  8]
 [ 0 -1 -2  1  0  0  0  0]
 [ 0  3  1 -3  0  0  0  0]
 [ 0  0  6  0  3 -5  0  1]
 [ 0  0  6  0 -5  5 -1  3]
 [ 0  0  6  0  0 -1  1  1]
 [ 0  0  6  0  1  3  1  4]]
```

- Dans *exo1.py*, on testera *matN* pour $n = 12$, $k_0 = 7$, $j_0 = 4$ puis pour $n = 100$, $k_0 = 12$, $j_0 = 32$
- On vérifiera les dimensions de la matrice obtenue et ses valeurs pour la ligne k_0 et la colonne j_0. Afficher aussi la trace de la matrice obtenue (en utilisant *numpy.trace*).

2 Classe *Droite2d*

Ecrire dans un fichier *Droite2d.py* la définition de la **classe** *Droite2d*, modélisant une droite dans $\mathbb{R}^2$. Un objet de la classe *Droite2d* est composé de plusieurs attributs:

- v, un tableau numpy 1D à 2 valeurs, contenant la pente de la droite et l'ordonnée à l'origine. L'équation de la droite dans le plan s'écrit donc mathématiquement sous la forme $y = v[0] * x + v[1]$.
- *color*, une chaîne de caractères définissant la couleur d'affichage de la droite. Des valeurs possibles sont "b", "g", "r", "k", ... (bleu, vert, rouge, noir)
- *style*, une chaîne de caractères définissant le style d'affichage de la droite. Les valeurs possibles sont "-", ":", "- -", ".-" (trait continu, pointillés, tirets, ...)

Ecrire, dans le fichier python, la définition de la classe en y incluant ce qui suit

- un constructeur de classe prenant 3 arguments en plus de *self*, qui sont v, *color* et *style*. On aura les valeurs par défaut: v=[0,0] dans un tableau numpy, *color*="b" et *style*="-".
- La méthode d'affichage de la classe doit être définie (méthode "__str__"). Le résultat de l'affichage d'une instance de la classe *Droite2d* avec *print* doit ressembler à ceci

```
>>> l = Droite2d(np.array([-2, 1]), color="r", style=":")
>>> print(l)
Droite d'équation y = -2.00*x + 1.00
```

- Définir une méthode *plot* permettant de représenter avec *matplotlib* une instance de la classe *Droite2D*. La couleur choisie et le style d'affichage doivent être pris en compte. L'intervalle d'affichage considéré est $[-10, 10]$ pour toutes les droites.

- Définir une méthode *intersect* avec deux arguments (dont *self*), qui sont des instances de la classe *Droite2d*, qui renvoie un tableau numpy contenant les coordonnées du point d'intersection entre les deux droites. S'il n'y a pas d'intersection, un tableau vide est renvoyé.

On testera la classe *Droite2d* en faisant ce qui suit pour chaque objet à tester:

```
>>> D = Droite2d(...)  # code volontairement incomplet
>>> print(D)
>>> plt.figure(); D.plot(); plt.xlim((-10,10)); plt.show(block=False)
```

On définira et on testera comme décrit précédemment les instances suivantes de la classe, dans le fichier python *Droite2d.py*

- la variable d0 est une instanciation de la classe *Droite2d* avec les attributs par défaut de la classe
- la variable $d1$ vérifie v=[1, 1], elle est représentée par des pointillés et de couleur noire
- la variable $d2$ vérifie v=[2, -1], elle est représentée par des tirets et de couleur verte
- Afficher les coordonnées de l'intersection de $d1$ et $d2$ en utilisant la méthode *intersect*

3 Classe de fonctions $\mathbb{P}_0$ par morceaux

Dans le fichier *pwf.py*, on veut définir la classe *PWF* (PieceWise constant Function) qui permet de modéliser des fonctions 1D constantes ($\mathbb{P}_0$) par morceaux. La classe contient trois attributs:

- y: un tableau numpy ou une liste, qui contient les positions des points de discontinuité de la fonction, càd là où la fonction possède deux valeurs, à gauche et à droite
- v: un tableau numpy ou une liste, de même taille que y, qui contient les différentes valeurs de la fonction (càd les valeurs des paliers). v contient ici les différentes valeurs à droite de la fonction (si on note la fonction par f, il s'agit de $f(y_i^+)$ aux points y_i, $\forall\, i$).
- *taille*: un entier égal à la taille de y ou de v

Plus précisément, si y et v sont de taille n et si on note f la fonction constante par morceaux qui correspond à l'instance de la classe PWF dont les attributs sont y et v, on a

$$\begin{aligned} y &= [y_0, y_1, \cdots, y_{n-1}], \quad v = [v_0, v_1, \cdots, v_{n-1}], \\ f(y) &= \begin{cases} 0, & \forall\, y \text{ tel que } y < y_0, \\ v_i, & \forall\, y \text{ tel que } y_i < y < y_{i+1}, \quad \forall\, 0 \leq i \leq n-2, \\ v_{n-1}, & \forall\, y \text{ tel que } y_{n-1} < y \end{cases} \end{aligned}$$

On suppose que la fonction constante par morceaux f vaut toujours 0 à gauche de y_0 et on prendra toujours la dernière valeur de v égale à 0.

La variable G qui suit est un exemple d'instance de la classe *PWF*, elle est représentée graphiquement sur la figure 1

```
>>> y = np.array([0.1, 0.2, 0.7, 0.8]); val = np.array([2, 1, -1, 0])
>>> G = PWF(y, val)
```

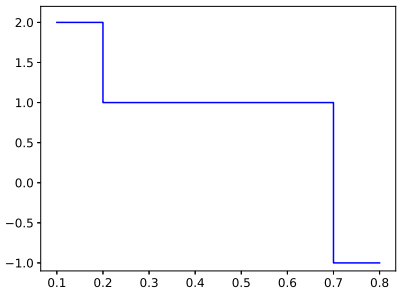

Figure 1: Fonction constante par morceaux correspondant à la variable G (elle vaut 0 en dehors de [0.1,0.8])

3.1 Constructeur et attributs de la classe

- Définir, dans le fichier *pwf.py*, le constructeur de la classe PWF, qui prend deux arguments en plus de self. Ce constructeur doit définir les attributs y, v et taille. Tester l'exemple de base en écrivant dans le même fichier, en dehors de la définition de la classe

  ```
  >>> x = np.array([0.1, 0.2, 0.7, 0.8]); val = np.array([2, 1, -1, 0])
  >>> G = PWF(x, val)
  >>> print(G.y); print(G.v); print(G.taille)
  ```

3.2 Opérateur d'affichage

- Définir, dans le même fichier, l'opérateur d'affichage pour la classe PWF. On souhaite afficher la taille de l'objet, son intervalle de définition ainsi que ses valeurs min et max (celles de l'attribut v), en vue d'obtenir ce qui suit:

  ```
  >>> print(G)
  Fonction constante par morceaux de taille 4,
   définie sur [0.100000,0.800000], de valeurs min -1.000000
   et max 2.000000
  ```

- Tester, dans le fichier *pwf.py*, pour l'objet G déjà défini.

3.3 Opérateur de multiplication par un scalaire

- Définir, dans le même fichier, l'opérateur de multiplication pour la classe PWF, par un scalaire et l'opérateur de multiplication à droite par un scalaire (leur contenu est identique). Ces deux opérateurs modifient uniquement les valeurs de la fonction. Tester comme ce qui suit, sur l'objet G déjà défini,

```
>>> G1 = G*6; G2 = -2*G
>>> print(G1); print(G2)
```

3.4 Fonction *integrate*

Dans le fichier *pwf.py*, définir:

- Une fonction *integrate* dans la classe PWFqui calcule l'intégrale de la fonction constante par morceaux sur son domaine de définition. On suppose toujours avoir $v_{n-1} = 0$.

- Tester la méthode *integrate* sur l'exemple suivant

```
>>> y = np.array([0.2, 0.3, 0.6, 0.65, 0.8])
>>> val = np.array([-1, 2, -1, 4, 0])
>>> Q = PWF(y, val)
>>> print(Q.integrate())
```

3.5 Fonction plot [facultatif]

On veut définir une méthode *plot* pour la classe PWF, qui représente avec *matplotlib* la fonction constante par morceaux associée à l'instance de la classe. Pour cela, on a besoin de deux fonctions, qui ne sont pas définies dans la classe PWF (mais toujours dans le même fichier)

- Définir la fonction *double_y* prenant un tableau numpy y (ou une liste) en argument. Cette fonction renvoie une répétition des valeurs de y, sauf aux premier et dernier indices, telle que

```
>>> y = np.array([y0, y1, .., y_{n-1}])
>>> w = double_y(y); print(w)  # de taille 2n-2
[y0, y1, y1, ..., y_{n-3}, y_{n-3}, y_{n-2}, y_{n-2}, y_{n-1}]
```

- Définir la fonction *double_v* prenant un tableau numpy v en argument. Cette fonction renvoie une répétition des valeurs de v, jusqu'à l'avant-dernière valeur de v incluse (et ne renvoie pas la dernière valeur de v, le résultat a donc la même taille que celui de *double_y*).

- Utiliser les deux fonctions précédentes pour définir la fonction *plot* de la classe PWF. La fonction *plot* contient deux arguments en plus de *self*, à savoir *ls* et *color* pour définir le style de ligne et la couleur et ont des valeurs par défaut ('-' et 'b'). On n'incluera pas l'appel à *plt.show* dans *plot*

- Tester la fonction *plot* sur les objets G et G2 pour obtenir un affichage semblable à celui de la figure 1:

```
>>> plt.figure()
>>> G.plot(); G2.plot(color='r')
>>> plt.show(block=False)
```

3.6 Opérateur d'addition [facultatif]

On souhaiterait définir l'opérateur d'addition pour deux objets de la classe PWF. Cela est compliqué car il faut être générique, les points de discontinuité pouvant être différents pour les deux instances.

- Implémenter cet opérateur et le tester sur au moins un exemple.