TP Python - Noté

SUJET A

Consignes:

- Le TP est à rendre au plus tard pour le 01/11 (inclus). Passé ce délai, il y aura des pénalités.
- Nommer les fichiers comme demandé dans les énoncés
- Si vous utilisez jupyter ou cocalc, sauvegardez régulièrement votre travail
- Envoyer les fichiers python *.py ou *.ipynb dans un mail aux adresses suivantes roland.ruelle@univ-cotedazur.fr, gilles.scarella@univ-cotedazur.fr, avec le sujet [M1 POO] TP noté python.

  Vous pouvez aussi envoyer vos fichiers dans une archive (au format .tar, .zip, .rar, .gz). N'oubliez pas de fichiers!

Votre travail doit être **propre à votre binôme/monôme**! Vous ne pouvez pas changer de sujet sous peine d'obtenir 0.

Les exercices sont indépendants et peuvent être traités dans n'importe quel ordre. Le barème est indicatif mais non définitif.

Tous les exemples demandés, mêmes basiques, doivent figurer dans votre code. S'il manque des exemples, vous n'aurez pas la totalité des points même si vos fonctions sont justes.

1 Classe *Triangle* [6pts]

Le code devra être écrit dans le fichier *triangle.py*.

Définir préalablement dans ce fichier, la fonction *triangle_surface* calculant la surface d'un triangle quelconque dans $\mathbb{R}^2$. On supposera que les coordonnées des sommets du triangle se trouvent dans un tableau numpy à 3 lignes et 2 colonnes.

Pour rappel, la surface d'un triangle ABC s'obtient par la formule:

$$\begin{aligned} surface &= \frac{1}{2} \left| \overrightarrow{AB} \wedge \overrightarrow{AC} \right| \\ \text{où } \overrightarrow{AB} \wedge \overrightarrow{AC} &= (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x) \quad \text{(on est en 2D)} \end{aligned}$$

Tester l'exemple suivant:

```
>>> P = np.array([[0,0],[2,0],[1,1]]); print(triangle_surface(P))
```

Vous devez obtenir: 1.0

Ecrire ensuite la définition de la **classe** *Triangle*, modélisant un triangle dans $\mathbb{R}^2$.

- La classe devra posséder l'attribut *nb_cotes*, correspondant au nombre de côtés d'un triangle. La valeur de *nb_cotes* sera donc identique pour n'importe quel objet de la classe.

Un objet de la classe *Triangle* est composé de deux attributs:

- P, un tableau numpy 2D (matrice) à 3 lignes, 2 colonnes contenant les coordonnées des sommets du triangle dans $\mathbb{R}^2$.
- *nom*, une chaîne de caractères pour nommer le triangle

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

- un constructeur de classe prenant 2 arguments en plus de *self*, qui sont un tableau numpy 2D, P, et une chaîne de caractères, *nom*. On aura les valeurs par défaut: *P=np.array([[0,0],[1,0],[0,1]])* et *nom*="T".
- Définir une méthode *surface*, pour tout objet de la classe, retournant la surface du triangle.
- Définir une méthode *perimetre*, pour tout objet de la classe, retournant le périmètre du triangle.
- Définir une méthode *barycentre* pour tout objet de la classe, retournant un tableau numpy 2D correspondant au barycentre (ou centre de gravité) du triangle. Pour rappel, le barycentre G vérifie:

  $\overrightarrow{OG} = \frac{1}{3}(\overrightarrow{OA} + \overrightarrow{OB} + \overrightarrow{OC})$ où O est le point de coordonnées (0,0)

- La méthode d'affichage de la classe doit être définie. L'affichage d'une instance de la classe *Triangle* doit ressembler à ceci (il faut utiliser les méthodes *surface*, *perimetre* et *barycentre* précédentes).

```
>>> P1 = np.array([[0,0], [2,0], [1,1]])
>>> T1 = Triangle(P1, "T1"); print(T1)
Objet Triangle à 3 côtés, de nom T1, de centre de gravité
 [1.000000,0.333333], de surface 1.000000 et de périmètre 4.828427
```

- Définir une méthode *plot* permettant de représenter avec *matplotlib* une instance de la classe *Triangle*, on reliera en trait continu les sommets du triangle de manière à obtenir un triangle.
  <u>*Indication*</u>: Vous devez utiliser l'attribut P décrit précédemment. Mais n'oubliez pas de bien "fermer" le triangle. On n'incluera pas l'appel à *plt.show()* dans *plot*.

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

```
>>> t = Triangle(...)  # code volontairement incomplet
>>> print(t)
>>> plt.figure(); t.plot(); plt.axis('equal'); plt.show(block=False)
```

- On définira et testera les instances suivantes de la classe:

– la variable $T0$ est l'instanciation de la classe *Triangle* avec les attributs par défaut de la classe

– la variable $T1$ représente le triangle de nom "T1" et défini par les sommets P1 = np.array([[0,0], [2,0], [1,1]])

– la variable $T2$ représente le triangle de nom "T2" et défini par les sommets P2 = np.array([[0.5,0], [2,1], [1,3]])

2 Résolution d'EDOs avec *scipy.integrate* [4pts]

2.1 EDO d'ordre 1

Dans le fichier *edoA.py*, on veut résoudre numériquement l'équation différentielle suivante où y est une fonction dépendant de t, définie sur l'intervalle $[1, 5]$.

$$\begin{cases} (t+4)\,\dfrac{dy}{dt} &= (-t^2 - 3t + 6)\,y \\ y(1) &= 25 \end{cases} \tag{1}$$

- Résoudre en utilisant *scipy.integrate.odeint* l'EDO (1). L'intervalle $[1, 5]$ sera discrétisé par n points (n=51) et on appelle x le tableau *numpy* contenant les points de discrétisation de $[1, 5]$.

- Définir la fonction python *yex* correspondant à la solution exacte de (1), telle que:

$$y_{ex}(t) = (t+4)^2\ \exp(-\frac{1}{2}\,(t-1)^2)\ \ \forall t\ \in [1, 5]$$

- Afficher avec *matplotlib* sur la même figure la solution exacte (en trait continu) et la solution numérique obtenue avec *scipy.integrate.odeint* (avec un marqueur ou un symbole de votre choix), en utilisant le tableau x.

2.2 EDO d'ordre 2

Toujours dans le fichier *edoA.py*, on veut résoudre numériquement l'équation différentielle suivante où z est une fonction dépendant de t, définie sur l'intervalle $[0, 10]$.

$$\begin{cases} 25\,\dfrac{d^2z}{dt^2} - z &= -2 \\ z(0) &= 2, \dfrac{dz}{dt}(0) = \dfrac{1}{5} \end{cases} \tag{2}$$

- Résoudre en utilisant *scipy.integrate.odeint* l'EDO (2). L'intervalle $[0, 10]$ sera discrétisé par n points (n=41) et on appelle v le tableau *numpy* contenant les points de discrétisation de $[0, 10]$.

- Définir la fonction python *zex* correspondant à la solution exacte de (2), telle que:

$$z_{ex}(t) = \sinh(\frac{t}{5}) + 2 = \frac{1}{2}\left(\exp(\frac{t}{5}) - \exp(-\frac{t}{5})\right) + 2, \quad \forall\, t \in [0, 10]$$

- Dans une nouvelle figure différente de celle de (1), afficher avec *matplotlib* sur la même figure la solution exacte (en trait continu) et la solution numérique obtenue avec *scipy.integrate.odeint* (avec un marqueur ou un symbole de votre choix), en utilisant le tableau v.

3 Tirages aléatoires et tableaux numpy [3pts]

Le code devra être écrit dans le fichier *exo3A.py*.

Dans cet exercice, on considère quatre variables aléatoires X_1, X_2, X_3, X_4 telles que

- X_1 suit $\mathcal{N}(10, 2)$ (loi normale de paramètres $\mu = 10$ et $\sigma = 2$)
- X_2 suit $\mathcal{B}(20, 0.6)$ (loi binômiale, $\mathcal{B}(m, p)$, avec $m = 20$, $p = 0.6$)
- X_3 suit $\mathcal{P}(5)$ (loi de Poisson de paramètre $\lambda = 5$)
- X_4 suit $\mathcal{U}(-1, 2)$ (loi uniforme sur $[-1, 2]$)

On considère le paramètre $n = 1000$.

- Définir une matrice M, tableau numpy 2d, à n lignes et 4 colonnes dont la i-ème colonne contient le tirage de X_i, de taille n.
- Ajouter quatre lignes supplémentaires à la matrice M:

- la première contient, dans chaque colonne i, la moyenne du tirage de X_i
- la deuxième contient dans chaque colonne i, l'écart-type du tirage de X_i
- la troisième contient, dans chaque colonne i, la différence en valeur absolue entre l'espérance (théorique) et la moyenne du tirage de X_i
- la quatrième contient, dans chaque colonne i, la différence en valeur absolue entre l'écart-type (théorique) et l'écart-type du tirage de X_i

- Faire afficher les quatre dernières lignes de M

4 Résolution de sytèmes linéaires [2pts]

Le but de cet exercice est la résolution de systèmes linéaires.

- Dans le fichier *exo4A.py*, écrire une fonction *mat44* prenant un seul argument k qui calcule x, solution du système linéaire suivant:

$$\begin{pmatrix} 2\,k+1 & -1 & k & 0 \\ 2 & k & -1 & -1 \\ k-2 & k-1 & 3\,k & 2 \\ -1 & 3 & -2k & k^2+1 \end{pmatrix} x = \begin{pmatrix} -k \\ 1 \\ k+2 \\ -2 \end{pmatrix}$$

- Tester la fonction en faisant afficher la solution du système linéaire pour $k = 2$ et $k = 5$

5 numpy.unique sur les lignes d'une matrice [5pts]

Le code sera écrit dans le fichier *uniqueA.py*.

On a besoin ici d'une fonction qui rend uniques les lignes d'une matrice (ou tableau numpy 2D).

5.1 Introduction (partie à lire simplement)

On a vu en cours la fonction *numpy.unique* qui permet de rendre uniques les éléments d'un tableau numpy 1D. Jusqu'à la version 1.13 de *numpy*, cette méthode ne fonctionnait que sur des tableaux 1D. Depuis cette version, si on considère M comme un tableau numpy 2D, le code suivant permet d'obtenir le tableau Mu souhaité

```
>>> import numpy as np
>>> Mu = np.unique(M, axis=0)
```

Mu est la matrice obtenue après élimination des lignes en double de M et Mu possède le même nombre de colonnes que M.
Par exemple,

```
>>> M = np.array([[1,4], [1,5], [1,6], [1,4], [1,5], [1,4]])
>>> Mu = np.unique(M, axis=0); print(Mu)
array([[1, 4],
       [1, 5],
       [1, 6]])
```

Le but de cet exercice est de coder votre propre fonction pour rendre uniques les lignes d'une matrice. Il n'est pas demandé que les lignes soient triées.
NB: Il ne faut pas utiliser *np.unique* avec l'option *axis=0* pour répondre à l'exercice!

- On pourra utiliser dans cet exercice la fonction *numpy.setdiff1d*.
  Cette fonction utilise deux arguments qui sont des tableaux numpy 1D et *numpy.setdiff1d*(t_1,t_2) renvoie le tableau t_1 privé de ses éléments figurant dans t_2. Pour illustrer,

  ```
  >>> print(np.setdiff1d(np.arange(1,10), np.array([0,2,3])))
  [1 4 5 6 7 8 9]
  ```

- On pourra aussi utiliser dans cet exercice (c'est autorisé) la fonction *numpy.unique* avec l'option *return_index=True*.
  numpy.unique(t_1, *return_index=True*) renvoie les éléments de t_1 qui sont uniques, après tri dans l'ordre croissant, ainsi que les indices correspondants à ces éléments dans t_1.

  ```
  >>> print(np.unique(np.array([0, 0, 2, 4, 4, 3]), return_index=True)
  (array([0, 2, 3, 4]), array([0, 2, 5, 3]))
  ```

NB: Si vous trouvez un algorithme qui répond à la question et qui fonctionne avec une version de numpy $\leq$ 1.12, ce sera accepté. Il vous faut néanmoins tester les exemples demandés à la fin de l'exercice.

5.2 Algorithme proposé

On pourra utiliser l'algorithme suivant.

- Créer d'abord une fonction *get_row_number_double(M)*, où M est un tableau numpy 2D, qui renvoie les indices de lignes de M qui sont en double dans M. Par exemple, on veut obtenir

```
>>> M = np.array([[1,4], [1,5], [1,6], [1,4], [1,5], [1,4]])
>>> get_row_number_double(M)
array([3, 4, 5]) # lignes d'indices 3, 4 et 5 en double
```

Idée d'algorithme:

- Principe: on recherche l'ensemble des indices de lignes de M qui correspondent à des lignes en double. On note x le tableau numpy (ou une liste) des indices recherchés. On va commencer par la première colonne de M et aller jusqu'à la dernière, sauf si x devient vide.
- Pour commencer, trouver tous les indices des valeurs en double de la 1ère colonne de M, qu'on affecte dans x. Ces indices sont des indices de ligne.
- faire une boucle sur les autres colonnes de M à l'intérieur de laquelle on cherche les indices des valeurs en double dans chaque colonne; au fur et à mesure on met à jour x en enlevant éventuellement des indices (la taille de x décroît en général et ne peut jamais croître)
- l'algo s'arrête, soit quand on a parcouru toutes les colonnes, soit quand le tableau x des indices est vide

On doit obtenir les résultats suivants:

```
>>> M1 = np.array([[1,2,3], [2,4,5], [1,2,4]])
>>> get_row_number_double(M1)  # pas de ligne en double
array([])
>>> M2 = np.array([[1,2,3], [2,4,5], [1,2,3]])
>>> get_row_number_double(M2)
array([2])  # ligne d'indice 2 en double
```

- En utilisant *numpy.setdiff1d* et la fonction *get_row_number_double*, définir la fonction *my_unique_row* qui rend uniques les lignes d'une matrice
- Tester les exemples suivants (M1 et M2 déjà définis)

```
>>> print(my_unique_row(M1))
>>> print(my_unique_row(M2))
>>> M3 = np.ones((50,20))
>>> print(my_unique_row(M3))
>>> M4 = np.ones((50,20)); M4[12,13]=0; M4[42,10]=-1; M4[29,4]=2;
>>> print(my_unique_row(M4))
```