Concours blanc MPSI **Mardi 19 Mai 2015**

Informatique pour tous : solution

Exercice :

a) Pour une fonction $f \in \mathcal{C}([0,x],\mathbb{R})$ la méthode des rectangles (à gauche) consiste à considérer la suite (S_n) où :

$$S_n(f) = \frac{x}{n}\sum_{i=0}^{n-1} f\left(\frac{ix}{n}\right).$$

Pour f de l'énoncé, on programme le calcul de $S_n(f)$ à l'aide d'une boucle, par exemple comme suit :

```
def erfR(x,n):
    S=0
    p=x/n
    for i in range(n):
       S=S+np.exp(-(i*p)**2)
    return p*S
```

ou avec un accumulateur pour les `i*p` :

```
def erfR(x,n):
    S=0
    p=x/n
    acc=0
    for i in range(n):
        S=S+np.exp(-(acc)**2)
        acc=acc+p
    return p*S
```

N.B. L'utilisation de `np.exp`, autrement dit de la fonction `exp` de `numpy` sera utile pour la suite, mais à ce stade on pourrait aussi utiliser celle du module `math`.

b) Avec les mêmes hypothèses, la méthode des trapèzes consiste à considérer la suite (T_n) où :

$$T_n(f) = \frac{x}{n}\left(\frac{f(0)}{2} + \sum_{i=1}^{n-1} f\left(\frac{i.x}{n}\right) + \frac{f(x)}{2}\right).$$

On a séparé les deux termes extrêmes qui n'interviennent que dans un trapèze. On peut donc définir :

```
def erfT(x,n):
    p=x/n
    S=(1+np.exp(-x**2))/2 # contribution des valeurs extremales
    for k in range(1,n):
        S = S + np.exp(-(k*p)**2)
    return p*S
```

c) L'objet `x` créé par `np.linspace` est un `np.array` : *tableau* `numpy`. Il en est de même pour `y` qui est un tableau de même longueur que `x` et contient les images de chaque entrée de `x` par notre fonction.

Remarque : Les opérations `+,x,**2` s'appliquent entrée par entrée à ces tableaux. Pour ce qui est de la fonction `exp`, la fonction `np.exp` s'appliquera aussi entrée par entrée à un tel tableau. Celle du module `math` ne conviendrait pas.

d) Par rapport à la formule du b), on change un peu de point de vue. Maintenant, on va sommer les aires de trapèzes une par une et stocker à chaque étape la valeur de la somme.

```
def erfG(x,n):
    pas=x/n # le pas
    X=[0] # initialisation du tableau des abscisses
    Y=[0] # initialisation du tableau des ordonnées.
    xcourant=0
    ycourant=0
    for k in range(0,n):
        xsuivant=xcourant+pas
        ycourant=ycourant+pas*(np.exp(-xcourant**2)+np.exp(-xsuivant**2))/2
```

```
            # on a rajouté à  ycourant l'aire du trapèze dont les bases sont d'abscisse
            #xcourant  et xsuivant.
            xcourant=xsuivant
            X.append(xcourant)
            Y.append(ycourant)
        plt.clf()
        plt.plot(X,Y)
        plt.show() # l'affichage
        return Y[-1] # la valeur finale
```

Problème

Question 1.

```
def plusBas(tab):
    xmin=tab[0][0]
    ymin=tab[1][0]
    j=0
    for i in range(len(tab[0])):
        if tab[1][i]<ymin :
            j=i
            xmin=tab[0][i]
            ymin=tab[1][i]
        elif tab[1][i]==ymin:
            if tab[0][i]<xmin:
                j=i
                xmin=tab[0][i]
                ymin=tab[1][i]
    return j
```

Question 2. Le test d'orientation donne +1 pour $i = 7$, $j = 3$, $k = 4$.
Il donne −1 pour $i = 8$, $j = 9$, $k = 10$.

Question 3.

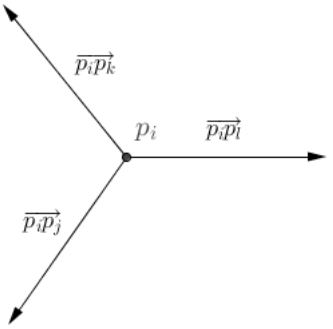
```
def orient(tab,i,j,k):
    pi=np.array([tab[0][i],tab[1][i]])
    pj=np.array([tab[0][j],tab[1][j]])
    pk=np.array([tab[0][k],tab[1][k]])
    vec1=pj-pi
    vec2=pk-pi
    det=vec1[0]*vec2[1]-vec1[1]*vec2[0]
    if det>0:
        return 1
    elif det==0:
        return 0
    else :
        return -1
```

Question 4.

— réflexivité : soit $j \neq i$, par déf. `orient`(tab,i, j, j) = 0 a fortiori `orient`(tab,i, j, j) ≤ 0 donc $p_j \preceq p_j$.

— Antisymétrie : soit $j, k \neq i$, tels que $p_j \preceq p_k$ et $p_k \preceq p_j$. Alors $\det(\overrightarrow{p_ip_j}, \overrightarrow{p_ip_k}) \leq 0$ et $\det(\overrightarrow{p_ip_k}, \overrightarrow{p_ip_j}) \leq 0$
Or $\det(\overrightarrow{p_ip_j}, \overrightarrow{p_ip_k}) = -\det(\overrightarrow{p_ip_k}, \overrightarrow{p_ip_j})$ donc ici $\det(\overrightarrow{p_ip_k}, \overrightarrow{p_ip_j}) = 0$.
Ceci signifie que les points p_i, p_j, p_k sont alignés. Or par l'hypothèse de position générale de l'énoncé, ceci signifie que deux d'entre eux sont confondus. Enfin comme p_j et p_k sont distincts de p_i on conclut que $p_j = p_k$.

— transitivité : soit $j, k, l \neq i$, tels que $p_j \preceq p_k$ et $p_k \preceq p_l$.
On a donc $\det(\overrightarrow{p_ip_j}, \overrightarrow{p_ip_k}) \leq 0$ et $\det(\overrightarrow{p_ip_k}\overrightarrow{p_ip_l}) \leq 0$.
Attention : sans hypothèse particulière sur p_i on ne pourrait rien conclure comme le montre le dessin suivant :



Dans ce dessin, pour la relation $\preceq$ associée à p_i, on a bien : $p_j \preceq p_k$ (on tourne dans le sens inverse-trigo de p_j à p_k) de même $p_k \preceq p_l$ mais on n'a pas $p_j \preceq p_l$.

La raison en est qu'en considérant les représentants dans $[-\pi, \pi]$ des angles orientés $(\widehat{\overrightarrow{p_ip_j}, \overrightarrow{p_ip_k}}) \equiv \theta_1 \in [-\pi, 0]$, $(\widehat{\overrightarrow{p_ip_k}, \overrightarrow{p_ip_l}}) \equiv \theta_2 \in [-\pi, 0]$, on a $\theta_1 + \theta_2 < -\pi$.
Mais ici, un tel phénomène ne peut pas se produire car :
• Pour le premier point p_i choisi : il est par déf. en bas à gauche du nuage de points. Donc tous les points p_j, p_k, p_k sont dans le demi-plan (fermé) délimité par la droite horizontale passant par p_i.
• Pour les autres point p_i insérés : si p_r est le point inséré avant p_i alors par construction de p_i tous les points $p_j \in P \setminus \{p_i, p_r\}$ seront dans le demi plan à gauche de (p_rp_i).

— Totalité : soit $j, k \neq i$. Alors si $\det(\overrightarrow{p_ip_j}, \overrightarrow{p_ip_k}) \leq 0$, on a $p_j \preceq p_k$, sinon $\det(\overrightarrow{p_ip_j}, \overrightarrow{p_ip_k}) \geq 0$ et $\det(\overrightarrow{p_ip_k}, \overrightarrow{p_ip_j}) \leq 0$ et donc $p_k \preceq p_j$.

Question 5.

```
def prochainPoint(tab,i):
    if i==0:
        temoin=1
    else:
        temoin=0
    n=len(tab[0])# le nombre d'entrées dans une ligne
    for j in range(n):
        if j!=i:
            if orient(tab,i,temoin,j)<0:
                temoin=j
    return temoin
```

N.B. La distinction de cas au départ pour l'initialisation de `témoin` est nécessaire car sinon, pour `i` et `témoin` égaux, on aurait *toujours* `orient(tab,i,temoin,j)=0`.

Question 6. On a `i=10` par hypothèse. Ainsi on commence avec `temoin=0` ce qui signifie qu'on va considérer au départ les orientations de $(\overrightarrow{p_{10}p_0}, \overrightarrow{p_{10}p_j})$.

• Quand `j=0`, `orient(tab,i,temoin,j)<0` a la valeur `False` car $\det(\overrightarrow{p_{10}p_0}, \overrightarrow{p_{10}p_0}) = 0$.

• Quand `j=1`, `orient(tab,i,temoin,j)<0` a la valeur `True` car $\det(\overrightarrow{p_{10}p_0}, \overrightarrow{p_{10}p_1}) < 0$.
Donc changement de témoin : `temoin=1`.

• Quand `j=2`, `orient(tab,i,temoin,j)<0` a la valeur `True` car $\det(\overrightarrow{p_{10}p_1}, \overrightarrow{p_{10}p_2}) < 0$.
Donc changement de témoin : `temoin=2`.

• pour `j=3,4`, on a $\det(\overrightarrow{p_{10}p_2}, \overrightarrow{p_{10}p_j}) > 0$. Pas de changement de témoin.

• pour `j=5`, on a $\det(\overrightarrow{p_{10}p_2}, \overrightarrow{p_{10}p_5}) < 0$.

Donc changement de témoin : `temoin=5`.

• pour tous les `j` supérieurs ou égaux à 6, on a $\det(\overrightarrow{p_{10}p_5}, \overrightarrow{p_{10}p_j}) \geq 0$.

Il n'y a plus de changement de témoin. La fonction renvoie donc `5`.

Question 7.

```
def convJarvis(tab,n):
    pointInitial=plusBas(tab,n)
    L=[]
    pointcourant=pointInitial
    while True:
        L.append(pointcourant)
        pointcourant=prochainPoint(tab,n,pointcourant)
        if pointcourant==pointInitial:
            return L
```

Question 8. Comme la fonction finale `convJarvis` appelle les fonctions `plusBas` et `prochainPoint`, on examine la complexité de ces deux fonctions :

• Dans `plusBas`, la boucle `for` fait n tours de boucle. A chaque tour de boucle le nombre d'opérations est majoré par 6 (3 test booléens, 3 affectations), en tout cas indépendant de n.

Ainsi la complexité de la fonction `plusBas` est en $O(n)$ (et en fait inférieure à $6n$).

(On n'a pas compté le calcul de `len(tabl[0])` dont on ne connaît la complexité que si on sait comment cette donnée est codée, mais de toute façon cette complexité n'est pas plus que linéaire.)

• La fonction `orient` est de complexité constante indépendante de la longueur de `tab` i.e. en $O(1)$.

• Dans `prochainPoint` on a encore une seule boucle `for` avec n tours de boucles et un nombre majoré par une constante d'opérations à chaque tour de boucles donc `prochainPoint` est en $O(n)$.

• Enfin dans `convJarvis` on a donc :

— Un appel à `plusBas` en $O(n)$,

— une boucle `while` qui fait exactement m tours de boucles (où m est le nombre de points du bord de conv(C)).

— à chaque tour de boucle : un appel à `prochainPoint` en $O(n)$ et 2 autres opérations (test booléen et ajout dans une liste) ; donc une complexité encore en $O(n)$ à chaque tour.

Au total la boucle coûte $O(nm)$ opérations et donc :

la fonction `convJarvis` a une complexité en $O(mn) + O(n) = O(mn)$.