CORRIGÉ DM INFO n°3 : CCP PSI 2018

Q1. Dans les formules de l'énoncé, α est fonction de t (ce n'est pas clair dans l'énoncé, car sur la figure 3, α désigne l'angle de frappe au départ...) ; la quantité Ω est, elle, constante (ce qui n'est pas dit non plus...) On a pour tout t :

$$u'(t) = \frac{\mathrm{d}^2x(t)}{\mathrm{d}t} \quad \text{et} \quad v'(t) = \frac{\mathrm{d}^2y(t)}{\mathrm{d}t}$$

et

$$u(t) = V(t)\cos(\alpha(t)) \quad , \quad v(t) = V(t)\sin(\alpha(t)) \quad \text{et} \quad V = \sqrt{u^2+v^2}.$$

En utilisant les formules de l'énoncé on a donc :

$$F(Y,t) = \frac{\mathrm{d}Y(t)}{\mathrm{d}t} = \begin{pmatrix} u' \\ v' \\ x' \\ y' \end{pmatrix} = \begin{pmatrix} -\dfrac{1}{2m}\pi R^2 \rho_{\text{air}} C_1 u\sqrt{u^2+v^2} - \dfrac{1}{m}\rho_{\text{air}} R^3 C_2 \Omega v \\ -g - \dfrac{1}{2m}\pi R^2 \rho_{\text{air}} C_1 v\sqrt{u^2+v^2} + \dfrac{1}{m}\rho_{\text{air}} R^3 C_2 \Omega u \\ u \\ v \end{pmatrix}.$$

En Python, cela pourrait s'écrire ainsi (sous réserve que les constantes aient été initialisées) :

```
piR2 = -pi * R**2 * rho_air * C1 / (2*m)
rhoR3 = rho_air * R**3 * Omega * C2 / m

def F(Y, t):
    u, v, x, y = Y
    V = sqrt(u*u + v*v)
    return array( [piR2 * u * V - rhoR3 * v, -g + piR2 * v * V + rhoR3 * u, u, v] )
```

Q2. La solution approchée Y est représentée par un tableau `numpy` de N colonnes (où N est le nombre de points d'approximation), tel que la colonne i contient les valeurs de la matrice $Y = {}^t\!\begin{pmatrix} u & v & x & y \end{pmatrix}$ à l'instant t_i.

```
def Euler(T, N, F, Y0):
    Y = zeros( (len(Y0), N) ) # tableau des valeurs approchées Y(t_i)
    t = arange(0, T, T/N) # tableau contenant les t_i
    Y[:, 0] = Y0 # conditions initiales
    delta = T/N # intervalle de temps, ne pas le recalculer à chaque fois
    for i in range(N-1):
        Y[:, i+1] = Y[:, i] + delta * F(Y[:, i], t[i])
    return (t, Y)
```

Q3.
- Une image est une liste à trois dimensions, l'expression `len(image1)` renvoie donc la longueur de la 1ère dimension, ici le nombre de coordonnées horizontales, soit 1024.
- `image1[12][244]` correspond à la couleur du pixel de coordonnées $(12, 244)$; cette couleur étant formée de trois éléments, `len(image[12][244])` renvoie la valeur 3.
- `image1[12][244][2]` est un entier entre 0 et 255 représentant l'intensité de la couleur bleue du pixel, calculer sa longueur provoquera donc une erreur.

  Rem : piège ou erreur d'énoncé ?

Q4. Les entiers entre 0 et 255 sont du type `uint8` et sont codés sur 8 bits, soit un octet. Une image occupe donc $1024 \times 768 \times 3 = 2359296$ octets $\approx 2,4$Mo.

Remarque : l'énoncé n'est pas clair ! Rien ne permet d'affirmer a priori, *comme je l'ai fait ci-dessus, que le type utilisé pour les couleurs est bien* `uint8` *; cela doit être forçé lors de la construction du tableau ; sinon par défaut, c'est le type* `int32` *ou* `int64` *qui est utilisé.*

Q5. L'espace de stockage pour un point joué doit pouvoir contenir les 1000 images par seconde de chacune des 10 caméras pendant 10 secondes, donc être de l'ordre de 2,4Mo $\times$ 100000 $\approx$ 240 Go. Pour un set il en faut 100 fois plus, soit 24To.

Q6. Baratin...

Q7. *L'énoncé n'est pas clair : il est dit que la procédure doit renvoyer un tableau, mais on ne sait pas s'il s'agit d'un tableau* ***numpy*** *ou d'un tableau sous forme de liste de listes. Compte tenu du début de la question* **Q.8***, je pense que l'énoncé veut des tableaux n'utilisant que les listes ; d'ailleurs, les images ne sont pas stockées dans des tableaux* ***numpy****, mais dans des variables de type* `list`*, que l'énoncé appelle d'ailleurs tableaux... On s'y perd un peu, d'autant plus qu'au début (***Q2***) et à la fin (***Q14***) du problème, l'énoncé demande clairement d'utiliser des tableaux* ***numpy****.*

On peut commencer par écrire une fonction `distance` qui calcule la distance euclidienne au carré entre deux points de $\mathbb{R}^3$ (ici des listes de longueur 3).

```
def distance(x, y):
    return (x[0] - y[0])**2 + (x[1]-y[1])**2 + (x[2]-y[2])**2

def detection(image1, image2):
    dx, dy = len(image1), len(image1[0])
    diff = [ [0] * dy ] * dx
    # si utilisation de tableaux numpy:
    # dx, dy, _ = image1.shape
    # diff = zeros( (dx, dy) )
    for x in range(dx):
        for y in range(dy):
            delta  = distance(image1[x][y], image2[x][y])
            diff[x][y] = delta if delta < 255 else 255
            # ou, si on a le droit à la fonction min: diff[x][y] = min(delta, 255)
            # si tableau numpy: diff[x, y] au lieu de diff[x][y]
    return diff

image_gray = detection(im1, im2)
```

Remarque : pour des raisons d'homogénéité, il aurait fallu considérer la vraie norme euclidienne (c'est-à-dire la racine de celle calculée ici) ; en effet, cela n'a guère de sens de comparer les carrés *à 255...*

Q8. Dans cette question, l'énoncé est (enfin) clair : on utilise des listes de listes pour représenter les tableaux à 2 dimensions.

```
def filtre(image, seuil):
    dx, dy = len(image), len(image[0])
    result =  [ [0] * dy ] * dx
    for x in range(dx):
        for y in range(dy):
            if image[x][y] >= seuil:
                result[x][y] = 1
    return result
```

Q9. Un pixel est isolé s'il est affecté de la valeur 1 alors que ses 8 voisins sont affectés de la valeur 0.

Les deux premiers tests proposés ne testent que 4 voisins, ils ne conviennent donc pas.

Dans le test numéro 4, chaque `range` ne décrit que 2 valeurs : dans `range(p-1, p+1)]` la valeur `p+1` est exclue. Donc au total seulement 4 pixels sont testés, ce test est incorrect.

Le test numéro 3 parcourt bien le carré de centre (p, c) et compte le nombre de pixels allumés en plus du pixel central (la variable `nb_pixel` est initialisée à -1 car le pixel du centre est allumé). Si le nombre de pixels allumé est 0, le point est considéré comme isolé et il est éliminé.

Q10.
- La ligne 2 permet de retrouver la liste des fichiers du répertoire `sequence_#####` où `#####` est rentré par l'utilisateur.
- La boucle qui commence ligne 3 devrait permettre de trouver le plus grand numéro des images de la liste précédente, mais ne fonctionne pas car la variable `index_max` est remise à zéro à chaque passage!!! L'instruction `index_max=0` doit être faite avant la boucle...
- Dans la boucle qui commence ligne 11, on détermine la 1ère image qui sera analysée, située 3 secondes avant la dernière (celle trouvée auparavant), en espérant qu'il y en a une.... On parcourt ensuite les images entre celle-ci et la dernière, en comparant à chaque fois l'image courante (`image2`) à la précédente (`image1`) et en ajoutant à `seq_balle` le résultat `liste_balles` de cette comparaison (dans le cas de la 1ère image on ne fait rien).

Q11.

```
def deplacement(pos1, pos2):
    return [ pos2[0] - pos1[0], pos2[1] - pos1[1] ]

dep = deplacement(pos1, pos2)
Lx = max( 2*dep[0] + 1, 3)
Ly = max( 2*dep[1] + 1, 3)
```

```
def distance_quad(xc, yc, liste_balle_i):
    liste_distances = []
    for i in range(len(liste_balle_i), 2):
        x, y = liste_balle_i[i], liste_balle_i[i+1]
        liste_distances.append( (x-xc)**2 + (y-yc)**2 )
    return liste_distances
```

```
def cherche_balle(xc, yc, Lx, Ly, liste_balle_i):
    if len(liste_balle_i) == 0:
        return [None, None]
    # on détermine la liste des distances et on en cherche le minimum
    # sans utiliser la fonction min de Python
    liste_distances = distance_quad(xc, yc, liste_balle_i)
    imin =0
    for i in range( 1, len(liste_distances) ):
        if liste_distances[i] < liste_distances[imin]:
            imin = i
    # on teste si la balle trouvée est bien dans la zone
    if abs( liste_balle_i[2*imin] - xc ) <= Lx/2 \
            and abs( liste_balle_i[2*imin+1] - yc ) <= Ly/2:
        return listeballe_i[2*imin:2]
    else:
        return [None, None]
```

```
def traj_balle(seq_balle, vit_init):
    balles_bonnes = [ seq_balle[0] ]
    xc, yc = seq_balle[0]
    dep = vit_init / 1000 # 1/1000 de seconde entre deux images
    Lx, Ly  = max( 2*dep[0] + 1, 3), max( 2*dep[1] + 1, 3)
    for i in range( 1, len(seq_balle) ):
        balle_suivante = cherche_balle( xc, yc, Lx, Ly, seq_balle[i] )
        if balle_suivante == [None, None]:
            # on se contente d'agrandir la zone de recherche
            # l'énoncé suppose que cela ne se produit pas deux fois de suite
            Lx, Ly = 2*Lx-1, 2*Ly-1
            xc, yc = xc + 2*dep[0], yc + 2*dep[1]
        else:
            balles_bonnes.append(balle_suivante)
            dep = deplacement( [xc, yc], balle_suivante )
            xc, yc = balle_suivante
            Lx, Ly  = max( 2*dep[0] + 1, 3), max( 2*dep[1] + 1, 3)
    return balles_bonnes
```

Q12. Le défaut de l'algorithme est qu'il choisit systématiquement la balle candidate la plus proche, qui n'est pas nécessairement la bonne. Si l'objet choisi n'est pas la balle, des erreurs en cascade vont s'ensuivre et l'algorithme perdra la balle. Il faudrait plutôt à chaque étape choisir plusieurs objets et comparer ce qui se passe après un certain nombre d'étapes : si l'algorithme échoue, c'est que l'objet choisi n'est pas la balle.

Q13. Il suffit ici de recopier les formules de l'énoncé.

```python
def pos_loc(e, f, x1, x2, y1, y2):
    d = x1 - x2
    return [e/d*x1, e/d*y1, e/d*f]
```

Q14. On applique les formules de changement de base pour les deux rotations, puis on translate.

```python
def pos_glo(pos1i, T1, Rx1, Ry1):
    # l'énoncé demande de renvoyer une liste donc on convertit le résultat de numpy!
    return list( T1 + Ry1.dot(Rx1.dot(pos1i)) )
```

Q15. On applique la fonction précédente à tous les éléments de la liste.

```python
def traj3D(coord_loc, T1, Rx1, Ry1):
    return [ pos_glo(pos1i, T1, Rx1, Ry1) for pos1i in coord_loc ]
```

Q16. On cherche le premier impact.

```python
def det_impact(coord_glo, eps):
    R = 0.033
    # on regarde d'abord si il existe une position entre R et R+eps dans une image
    for i in range(len(coord_glo)):
        if R <= coord_glo[i][1] <= R + eps:
            return [ coord_glo[i][0], coord_glo[i][2] ]
    # sinon, on cherche quand il y a eu inversion du sens de déplacement
    # en espérant qu'il en existe!
    for i in range(1, len(coord_glo)-1):
        if coord_glo[i][1] < coord_glo[i-1][1] and coord_glo[i][1] < coord_glo[i+1][1]:
            return [ (coord_glo[i-1][0] + coord_glo[i+1][0])/2,\
                (coord_glo[i-1][2] + coord_glo[i+1][2])/2 ]
```

Q17. Quelle horrible idée de renvoyer des chaînes de caractère dans une fonction Python!!!

Attention : si l'on regarde attentivement la figure 3, on s'aperçoit que le terrain est située dans la partie $z \leqslant 0$!

```python
def res_final(impact, L, l):
    return "IN" if 0 <= impact[0] <= L and -l <= impact[1] <= 0 else "OUT"
```

Q18. Question sans intérêt puisqu'il suffit de recopier ce qui est écrit dans l'annexe !

```python
def vis_traj3D(coord_glo):
    x, y, z = coord_glo[:, 0], coord_glo[:, 1], coord_glo[:, 2]
    gca(projection='3d').plot(x, y, z)
    title('Visualisation trajectoire')
```

Q19. Baratin...

Q20.

```sql
SELECT id FROM MATCHS WHERE joueur1='Federer' OR joueur2='Federer'
```

Q21.

```sql
SELECT MAX(nombre) FROM POINTS WHERE mid=4
```

Q22. On utilise une jointure des tables MATCHS et POINTS.

```sql
SELECT p.fichier FROM MATCHS AS m, POINTS AS p
WHERE m.id = p.mid AND m.nom='Federer-Murray'
```

On peut écrire aussi :

```sql
SELECT fichier FROM POINTS
JOIN MATCHS ON MATCHS.id=POINTS.mid
WHERE nom='Federer-Murray'
```