DS1 - DETECTION D'OBJETS DANS LE CADRE DE LA CONDUITE AUTONOME

CORRIGÉ

Partie I - Localisation dans l'espace

Q1. 3 pts (dico, str, liste vide) :

```
dico={'adulte' :[],'enfant' :[], 'animaux' : [], 'véhicule' :[], 'indéterminé'
:[]}
```

Q2. 2 pts (un peu laborieux mais facile, tuple accepté même mieux, append aussi) :

```
dico['adulte']=(175*0.8,175*1.2)
dico['enfant']=( 88 , 132)
dico['animaux']=( 40 , 60)
dico['véhicule']=( 160 , 240)
dico['indéterminé']=( 0 , 40)
```

Q3. Obstacles_rencontres (4 pts) :

```
def obstacles_rencontres(dico,hauteur) :
    s=[]
    for cle in dico :
        if dico[cle][0] < valeur < dico[cle][1] :
            s.append(cle)
    return s
```

Q4. 5 pts (for → 1,calcul → 1, h→ 1, append→1,return liste →1) :

```
def focale(faces_detec,Dr,Hr) :
      L = [ ]
      for (x, y, h, w) in faces_detec:
            L.append(Dr * h / Hr)
      return L
```

Q5. 4 pts (fonction, calcul de la somme, moy, sortie texte) : **moyenne toute simple OK**

```
def moy(L) :
     return" la valeur de la focale pour n="+str(n)+ " est :" + str(sum(L) / n) +
" pixels")
```

Partie II - Traitement de l'image

II.1 - Calcul de luminance

Q6. Espace mémoire (3pts)

Une composante de couleur est un nombre compris entre 0 et 255. Il se stocke sur 1 octet.
Un pixel stocké au format RVB est composé de 3 nombres compris entre 0 et 255. Un pixel nécessite donc 3 octets de stockage.) 1
Une image de 48 MPixels utilisera donc $3 \times 48 \times 10^6$ octets, c'est à dire 144Mo.) 2

Q7. fonction Clinear(C) (3pts)

```
def Clinear(C):
    C1 = C/255
    if C1<=0.04045:
         Clin=C1/12.92
    else:
         Clin=((C1+0.055)/1.055)**2.4    attention puissance ^    → -1
    return Clin
```

Q8. fonction Intensite(pix) (4pts)

```
def Intensite(pix):
    Ylin  =  0.2126*Clinear(pix[0])  +  0.7152*  Clinear(pix[1])  +  0.0722*
Clinear(pix[2])
    if Ylin<=0.0031308:
         Y=12.92*Ylin
    else:
         Y=1.055*Ylin**(1/2.4)-0.055
    return int (255*Y)
```

Q9. fonction init (3pts) **ATTENTION aux listes liées→ 1pt**

```
def init(h,w):
     L=[]
     for x in range(h):
          L.append([0]*w)
     return L
```

```
M = []
for i range(h) :
     ligne = []
     for j in range(w) :
            ligne.append(0)
     M.append(ligne)
```

Q10. fonction NiveauxGris(I) (4pts)

```
def NiveauxGris(I):
     taillex,tailley,nb_couleurs = len(I),len(I[0]),len(I[0][0])
     Igris=init(taillex,tailley) # on crée l'image en niveaux de gris nulle
     for x in range(taillex):
          for y in range(tailley):
               Igris[x][y]=Intensite(I[x][y])
     return Igris
```

Q11. Complexité (2pts) :
O(h*w) les 2 boucles imbriquées (quadratique OK)

II.2 - Le concept d'image intégrale

Q12. 2 pts : Les 2 lignes permettent de récupérer la taille de l'image et de définir la taille des sections en prenant la partie entière de la division de la taille.

```
n = len(image)//3
p = len(image[0])//3
```

Q13. 4 pts (2 boucles for x,y inclus et somme) :

```
def coef_integral(M,x,y):
    s = 0
    for i in range(0,x + 1):
        for j in range(0,y + 1):
            s += M[i][j]
    return s
```

Q14. 4 pts (2 init, 1 pour les 2 boucles, 1 appel fonction) :

```
M_int=np.zeros(M.shape) #init
for i in range(0,len(M)):
      for j in range(0,len(M[i])):
            M_int[i][j] = coef_integral(M,i,j)
```

Q15. 2 pts (le calcul est donné) : *attention indice -1 → 1pt*

```
def lumi_section(M_int,x,y,n,p):
    return (M_int[x + n - 1][y + p - 1] - M_int[x + n - 1][y - 1] - M_int[x - 1][y + p - 1] + M_int[x - 1][y - 1]) / (n * p)
```

II.3 - Détection des obstacles : algorithme de Viola-Jones

Q16. 6 pts (Algo viola jones)

```
def detection(M_int, n, p, C_ref, seuil) :
      L_bord = []    1 pour la liste
      h,w=len(M_int), len(M_int[0])      1
      for x in range(1,h - n + 1):
            for y in range(1,w - p + 1):  1 pour les 2 boucles
                  lum_g = lumi_section(M_int,x,y,n,int(p / 2))
                  lum_d = lumi_section(M_int,x,y + int(p / 2), n ,int(p / 2))
1 pour les sections gauche et droite
                  C_obs = max(lum_g,lum_d) - min(lum_g,lum_d)
                  correlation = round(C_obs / C_ref,2) #arrondi pas obligatoire
1 pour le calcul de correlation
                  if correlation> seuil:
                        L_bord.append([x,y]) 1 pour test et remplissage de L
     return L_bord
```

Partie III - Optimisation - Méthode KNN

III.1 - Données

Q17. Fonction min,max (3pts)

```
def min_max(X) :
     mini,maxi = X[0],X[0]
     for e in X :
         if e < mini :
             mini = e
         if e > maxi :
             maxi = e
     return mini,maxi
```

Q18. fonction `coeff_normalise(data)` (6pts)

```
def coeff_normalise(data):
    L=[] 1
    for i in range(len(data)): 1
        ligne=data[i]
        mini,maxi = min_max(ligne) 1
        L.append([(data[i][j]-mini) /(maxi-mini)  for j in range(len(data[i]))])
    return L              1 pour le calcul, 1 pour le parcours de la ligne,
                          1 pour la construction de L
```

Q19. une fonction `distance(z,data)` (4pts)

```
def distance(z,data) :
    distances =[] 1
    for i in range(len(data)) :
        d=0
        for j in range(len(z)) : 1
            d +=((z[j]-data[i][j]))**2     1
        distances.append(sqrt(d))    1
    return distances
```

III.2 - Détermination des K plus proches voisins

Q20. Fonction fusion(3pts)

```
def fusion(T1,T2) :
    if T1==[] :
        return T2 #ligne 1 à compléter
    if T2==[] :
        return T1 #ligne 2 à compléter    1 pour les 2 return
    if T1[0][0]< T2[0][0] :
        return [T1[0]]+fusion(T1[1:],T2)
    else :
        return [T2[0]]+fusion(T1,T2[1:]) #ligne 3 à compléter   2
```

Q21. Performances (3pts)

Le code du tri fusion proposé ici a une complexité en $O(n^2 \log n)$ ce qui est très grand. En effet ce tri est efficace si la fonction fusion, fusionne deux listes triées T1 et T2 en une nouvelle liste avec la complexité $O(len(T1) + len(T2))$.

Or, avec les listes Python, le slicing T1[1:n] a une complexité en $O(n-1)$: le temps de recopie des éléments dans une nouvelle liste. La complexité de la fonction fusionne sera donc $O(n + (n-1) + (n-2) + \ldots + 2 + 1) = O(n^2)$ 1

Comme la fonction tri a une complexité en $O(\log n)$, la complexité totale est $O(n^2 \log n)$

Si on raisonne avec des tableaux numpy, le slicing fonctionnant sans copie, il n'y n'aurait pas ce problème et la complexité du tri fusion serait bien quasi linéaire $O(n \log n)$ 1

L'autre problème est que le nombre d'appels récursifs est limité en Python. Pour les grandes listes, même en modifiant la profondeur de récursion, Python plantera faute d'un espace mémoire suffisant pour les stocker. 1

Q22. algo KNN (3pts → 1 par partie)

partie1 : construit une liste de listes contenant la distance euclidienne entre les attributs de l'obstacle à classer et les attributs de chaque obstacle déjà classé, ainsi que l'indice de l'obstacle classé. Cette liste est triée par ordre des distances croissantes.

`T` : liste de listes des distances et indices

`dist` : liste des distances entre les attributs de l'obstacle à classer et ceux des obstacles déjà classés

partie 2 : comptabilise, parmi les K premiers obstacles, le nombre d'obstacles pour chaque type.

`select` : liste de nb éléments, chaque élément i contient le nombre d'obstacles (parmi les K premiers) possédant le type i.

partie 3 : recherche le type d'obstacles correspondant au groupe le plus nombreux

`ind` : numéro du type d'obstacle possédant le groupe le plus nombreux

Q23. C'est une Matrice de confusion (*on le verra un peu plus tard)*

La diagonale correspond aux tests positifs, le nombre de prédiction correcte du type d'obstacle.) 1

Les tests de la première ligne montrent que pour 41 obstacles du premier type, il y a eu 27 trouvés au premier type (donc correct), 4 pour le second, 3 pour le troisième, 5 pour le 4^{ième} et 2 pour le 5^{ième} .

Performances en pourcentage de réussite (27+18+40+36+19)/ (200) =70 %) 1

30% d'erreurs, ça commence à faire beaucoup !

Soin/Présentation : (4 pts)

FIN