BANQUE PT 2021 : CORRIGE DE LA PARTIE INFORMATIQUE DE L'EPREUVE D'INFORMATIQUE ET MODELISATION DE SYSTEMES PHYSIQUES

ETUDE D'UN SYSTEME AUTOFOCUS D'APPAREIL PHOTO NUMERIQUE

III. Partie informatique

Avec la synthaxe suivante, les fonctions importées sont utilisables sans préfixe :

```
from numpy import *
from matplotlib.pyplot import *
```

III.1. Mesure du contraste

Q18. La valeur d'une composante est représentée par un entier allant de 0 à 255. Il y a donc $256 = 2^8$ valeurs possibles, c'est codable sur 8 bits = 1 octet. L'espace mémoire nécessaire pour stocker la valeur d'*une composante* est donc *un octet*.
Pour chaque pixel, il y a 3 composantes (RVB). L'espace mémoire nécessaire pour stocker la valeur d'*un pixel* est donc *3 octets*.
Pour une image, il y a 48 MPixels, et chaque pixel nécessite 3 octets. L'espace mémoire nécessaire pour stocker la valeur d'*une image* est donc *144 Mo* ($48 \times 3 = 144$).

Q19. Il manque des explications dans l'énoncé… Pour les images que nous manipulons, l'intensité de chaque couleur est codée par un flottant entre 0 et 1 (les formules de l'énoncé concernent assez clairement des flottants), et pas par un entier compris entre 0 et 255, comme le laisserait penser le texte d'introduction…
On considère donc que « val » est un flottant.

```
def Clinear(val):
    if val <= 0.04045:
        val_lin = val / 12.92
    else:
        val_lin = ((val + 0.055) / 1.055)**2.4
    return val_lin
```

Q20.

```
def Y(pix):
    # on extrait d'abord les 3 couleurs en échelle non linéaire :
    R = pix[0]
    V = pix[1]
    B = pix[2]
    # on calcule ensuite les valeurs linéarisées :
    R_lin = Clinear(R)
    V_lin = Clinear(V)
    B_lin = Clinear(B)
    # on calcule ensuite la valeur du niveau de gris en échelle linéaire :
    Y_lin = 0.2126*R_lin + 0.7152*V_lin + 0.0722*B_lin
    # on repasse ensuite dans l'espace non linéaire :
    if Y_lin <= 0.0031308:
        Y_non_lin = 12.92 * Y_lin
    else:
        Y_non_lin = 1.055 * Y_lin**(1/2.4) - 0.055
    return Y_non_lin
```

Q21. Là encore, l'énoncé n'est pas clair. Faut-il mettre les 3 coefficients RVB à la même valeur pour avoir une image en niveau de gris (si le format de l'image reste un format d'image en couleurs), ou ne faut-il qu'une seule valeur par pixel (ce qui réduit le nombre d'octets nécessaire au codage) ?
Dans les 2 premières solutions proposées ci-dessous, les fonctions mettent les 3 couleurs à la valeur souhaitée (valeur du niveau de gris) (le format de l'image reste un format d'image en couleurs) :

La version proposée ci-dessous utilise des tableaux Numpy :

```
def NiveauxGris(I):
    J = copy(I)    # on copie I pour ne pas modifier l'image originale
    nb_lignes = J.shape[0]      # nombre de lignes
    nb_colonnes = J.shape[1]    # nombre de colonnes
    for i in range(nb_lignes):          # on parcourt toutes les lignes
        for j in range(nb_colonnes):    # on parcourt toutes les colonnes
            Y_non_lin = Y(J[i][j])      # on calcule le niveau de gris du pixel
            # on affecte le résultat aux 3 couleurs
            J[i][j] = [Y_non_lin, Y_non_lin, Y_non_lin]
    return J
```

Une version compatible avec des tableaux Numpy et des listes de listes de listes Python :

```
def NiveauxGris2(I):
    nb_lignes = len(I)          # nombre de lignes
    nb_colonnes = len(I[0])    # nombre de colonnes
    J = zeros((nb_lignes, nb_colonnes, 3))  # 3 pour les 3 couleurs RVB
    for i in range(nb_lignes):          # on parcourt toutes les lignes
        for j in range(nb_colonnes):    # on parcourt toutes les colonnes
            Y_non_lin = Y(I[i][j])      # on calcule le niveau de gris du pixel
            # on affecte le résultat aux 3 couleurs
            J[i][j] = [Y_non_lin, Y_non_lin, Y_non_lin]
    return J
```

Ci-dessous une solution qui ne retourne qu'une valeur par pixel (ce qui réduit le nombre d'octets nécessaire au codage). C'est en fait sans doute ce qui est demandé quand on lit la suite de l'énoncé (« convolution » etc) :

```
def NiveauxGris3(I):
    nb_lignes = len(I)
    nb_colonnes = len(I[0])
    J = zeros((nb_lignes, nb_colonnes))  # pas de 3 car 1 seule valeur / pixel
    for i in range(nb_lignes):
        for j in range(nb_colonnes):
            # on affecte le résultat au pixel
            J[i][j] = Y(I[i][j])
    return J
```

Q22.

```
def convolution(A, B):
    somme = 0      # initialisation de la somme à effectuer
    for i in range(3):
        for j in range(3):
            somme = somme + A[i][j] * B[i][j]
    return somme
```

Une version utilisable uniquement avec les tableaux Numpy :

```
def convolution2(A, B):
    produit = A * B     # produit terme à terme
    # sum(produit) utilise la fonction numpy à cause du import *
    return sum(produit)
```

Q23. A ce stade, il faudrait sans doute à nouveau considérer des images stockées sous forme d'entiers codés sur 8 bits… Car sinon « c » sera très souvent nul… Enoncé peu clair… !!

```
def contraste_pixel(I, i, j):
    Gx = array([-1, 0, 1], [-2, 0, 2], [-1, 0, 1])
    Gy = array([-1, -2, -1], [0, 0, 0], [1, 2, 1])
    Ie = I[i-1:i+2][j-1:j+2]    # on extrait la matrice Ie de I
    c = int(sqrt((convolution(Ie, Gx))**2 + (convolution(Ie, Gy)**2)))
    return c
```

Q24.

```
def contraste(I):
    somme = 0   # on initialise la somme
    nb_lignes = I.shape[0]      # nombre de lignes
    nb_colonnes = I.shape[1]    # nombre de colonnes
    for i in range(1, nb_lignes - 1):  # on ne considère pas les bords
        for j in range(1, nb_colonnes - 1):  # on ne considère pas les bords
            somme += contraste_pixel(I, i, j)
    nb_pixels = (nb_lignes - 2) * (nb_colonnes - 2)     # sans les bords
    c_ref = somme / nb_pixels
    return c_ref
```

Q25.

Pour les deux propositions ci-dessous, il n'y a pas d'argument à la fonction « reglage », et la fonction ne retourne rien (« None »). Son rôle est de positionner correctement l'objectif.

1ère possibilité, avec une boucle for :

```
def reglage():
    val = 0     # on initialise la position
    position_objectif(val)    # on positionne l'objectif
    imageRVB = prise()      # on prend une photo au format RVB
    Niv_gris = NiveauxGris(imageRVB)     # on passe en niveau de gris
    c = contraste(Niv_gris)     # on calcule le contraste

    for val in range(1, 1001):
        position_objectif(val)    # on positionne l'objectif
        imageRVB = prise()      # on prend une photo au format RVB
        Niv_gris = NiveauxGris(imageRVB)     # on passe en niveau de gris
        c1 = contraste(Niv_gris)     # on calcule le nouveau contraste
        if c1 < c:      # si le contraste a diminué
            position_objectif(val-1)    # on recule d'un pas
            return None     # ce return sert à sortir sans finir la boucle for
        else:
            c = c1      # on réinitialise c
```

2ème possibilité, avec une boucle while :

```
def reglage2():
    val = 0     # on initialise la position
    position_objectif(val)    # on positionne l'objectif
    imageRVB = prise()      # on prend une photo au format RVB
    Niv_gris = NiveauxGris(imageRVB)     # on passe en niveau de gris
    c = contraste(Niv_gris)     # on calcule le contraste

    val = 1     # position suivante
    position_objectif(val)    # on positionne l'objectif
    imageRVB = prise()      # on prend une photo au format RVB
    Niv_gris = NiveauxGris(imageRVB)     # on passe en niveau de gris
    c1 = contraste(Niv_gris)     # on calcule le contraste

    if c1 < c:      # si le contraste a diminué
        position_objectif(0)    # on recule d'un pas
        return None             # on sort de la fonction

    while c < c1:   # tant que le contraste augmente
        c = c1      # on réinitialise c
        val = val + 1
        position_objectif(val)    # on positionne l'objectif
        imageRVB = prise()      # on prend une photo au format RVB
        Niv_gris = NiveauxGris(imageRVB)     # on passe en niveau de gris
        c1 = contraste(Niv_gris)     # on calcule le contraste

    position_objectif(val-1)    # on recule d'un pas
    return None                 # on sort de la fonction
```

III.2. Détection de phase

Q26.

```
def extraction(L1, L2, dec):
    if dec == 0:
        return L1, L2
    if dec > 0:
        # on retire les dec derniers éléments à L1 et les dec premiers à L2
        return L1[:-dec], L2[dec:]
    else:   # si dec < 0
        # on retire les -dec premiers éléments à L1 et les -dec derniers à L2
        return L1[-dec:], L2[:dec]
```

Q27. En partant du principe que les deux listes L1 et L2 ont même longueur :

```
def comparaison(L1, L2):
    compteur = 0
    while compteur < len(L1) and L1[compteur] == L2[compteur]:
        compteur += 1
    if compteur != len(L1):
        return False
    return True
```

Autre possibilité :

```
def comparaison2(L1, L2):
    compteur = 0
    while compteur < len(L1) and L1[compteur] == L2[compteur]:
        compteur += 1
    return compteur == len(L1)
```

Q28.

```
def recherche_decalage(L1, L2):
    for dec in range(-80, 81):
        sous_L1, sous_L2 = extraction(L1, L2, dec)
        # si les 2 listes extraites avec un décalage dec sont identiques
        if comparaison(sous_L1, sous_L2):
            return dec
    return None
```

Q29.

Dans le meilleur des cas : (le décalage cherché est le 1er testé : dec = - 80)

- Si dec = - 80, la fonction « recherche_decalage » n'utilise qu'une fois la fonction « comparaison », et qu'une fois la fonction « extraction ».
- Si dec = - 80, les deux sous-listes testées sont de longueur (*n – 80).*
- La fonction « comparaison », qui retournera True par hypothèse, effectue (*2 × (n – 80) + 1)* comparaisons (à chaque fois, comparaison entre compteur et len(L1), et entre L1[compteur] et L2[compteur], et la comparaison finale entre compteur et len(L1)).
- La fonction « extraction » effectue *2* comparaisons.

Il y aura alors *2 + (2 × (n – 80) + 1)* ≈ ***2n*** **comparaisons** (*n* est grand).

Dans le pire des cas : il faut alors tester les *m* valeurs possibles pour dec. En ordre de grandeur, il y aura ***2nm*** **comparaisons**.

Dans tous les cas, la complexité est linéaire.

Q30.

```
def erreur(L1, L2):
    somme = 0
    n = len(L1)
    for i in range(n):
        somme += (L1[i] - L2[i])**2
    return somme / n
```

Autre possibilité avec les tableaux Numpy :

```
def erreur2(L1, L2):
    return sum((L1 - L2)**2) / len(L1)
```

Q31.

```
def recherche_decalage_2(L1, L2):
    sous_L1, sous_L2 = extraction(L1, L2, -80)
    err_mini = erreur(sous_L1, sous_L2)
    dec_mini = -80
    for dec in range(-79, 81):
        sous_L1, sous_L2 = extraction(L1, L2, dec)
        err = erreur(sous_L1, sous_L2)
        if err < err_mini:
            err_mini = err
            dec_mini = dec
    return dec_mini
```

III.3. Comparaison des deux méthodes

Q32.

Autofocus à mesure de contraste :
- Avantages : plus compact (moins d'optique), moins fragile (moins d'optique), moins coûteux
- Inconvénients : moins performant pour les sujets en déplacement rapide
- Adapté aux appareils compacts, peu encombrants et peu coûteux, matériel grand public

Autofocus à détection de phase :
- Avantages : plus performant pour les sujets en déplacement rapide
- Inconvénients : plus encombrant (plus d'optique), plus fragile (plus d'optique), plus coûteux
- Adapté aux appareils réflex, matériel professionnel

D'un point de vue traitement numérique, la méthode par détection de phase semble beaucoup plus rapide. Dans la méthode à mesure de contraste, on effectue beaucoup d'opérations arithmétiques par pixel (pour la transformation en niveaux de gris, les convolutions, la somme des contrastes). Et il y a au pire 1001 pas à effectuer, l'un après l'autre…

III.4. Commande du moteur pas à pas

Q33.

```
def faire_un_pas_positif(pas_actuel):
    if pas_actuel % 8 == 0:    # %8 pour se ramener aux 7 cas expliqués
        modif_sortie(IN3, True)
    elif pas_actuel % 8 == 1:
        modif_sortie(IN1, False)
    elif pas_actuel % 8 == 2:
        modif_sortie(IN2, True)
    elif pas_actuel % 8 == 3:
        modif_sortie(IN3, False)
    elif pas_actuel % 8 == 4:
        modif_sortie(IN4, True)
    elif pas_actuel % 8 == 5:
        modif_sortie(IN2, False)
    elif pas_actuel % 8 == 6:
        modif_sortie(IN1, True)
    else:
        modif_sortie(IN4, False)
    return (pas_actuel + 1)
```

Q34.

```
def position_objectif(pas):
    assert 0 <= pas <= 1000    # on vérifie que le pas demandé est atteignable
    if pas_courant < pas:
        while pas_courant < pas:
            faire_un_pas_positif(pas_courant)
            pas_courant += 1
    if pas_courant > pas:
        while pas_courant > pas:
            faire_un_pas_negatif(pas_courant)
            pas_courant -= 1
```

IV. Gestion des photographies

Q35. Les lignes d'une table sont toujours deux à deux distinctes. On appelle clé primaire d'une table tout sous-ensemble minimal de ses attributs (colonnes) permettant d'identifier de façon unique les lignes.

Q36. L'énoncé dit que la date est de type texte. Mais le format proposé est de type entier… Il y a une incohérence…

```
SELECT id
FROM photos
WHERE date = '20181111'
```

Q37.
```
SELECT DISTINCT nom, prenom
FROM photographes
JOIN photos ON photographes.id = photos.idp
WHERE date = '20181111'
```

J'ai écrit SELECT DISTINCT pour ne pas faire apparaitre plusieurs fois les noms et prénoms d'un même photographe (il est probable que des photographes aient pris plusieurs photos ce jour-là !).

Q38.
```
SELECT nom, prenom, heure
FROM photographes
JOIN photos ON photographes.id = photos.idp
WHERE date = '20181111'
```

Je n'ai pas écrit SELECT DISTINCT car il est improbable qu'un même photographe ait pris deux photos au même instant !

Q39.

Avant rotation (image originale) :

0,0	0,1			
1,0				
			i, j	

Après rotation :

0,0	0,1			
1,0				

En notant nb_lignes le nombre de lignes et nb_colonnes le nombre de colonnes dans l'image originale : le pixel qui se trouvait à la ligne i et à la colonne j se retrouve à présent :

- à la ligne (nb_lignes – i – 1)
- à la colonne (nb_colonnes – j – 1)

Les nouvelles coordonnées d'un pixel de coordonnées (i, j) après une rotation de 180° sont donc :

((nb_lignes – i – 1), (nb_colonnes – j – 1))

```
def rotation_180(image):
    nb_lignes = image.shape[0]      # nombre de lignes
    nb_colonnes = image.shape[1]    # nombre de colonnes
    Rotation = copy(image)          # on copie pour ne pas modifier image
    for i in range(nb_lignes):
        for j in range(nb_colonnes):
            Rotation[nb_lignes - i - 1][nb_colonnes - j - 1] = image[i][j]
    return Rotation
```

Q40.

Avant rotation (image originale) :

0,0	0,1			
1,0				
			i, j	

Après rotation :

0,0	0,1					
1,0						

En notant nb_lignes le nombre de lignes et nb_colonnes le nombre de colonnes dans l'image originale : le pixel qui se trouvait à la ligne i et à la colonne j se retrouve à présent :

- à la ligne (nb_colonnes – j – 1)
- à la colonne i

Les nouvelles coordonnées d'un pixel de coordonnées (i, j) après une rotation de 90° dans le sens trigonométrique sont donc :

((nb_colonnes – j – 1), i)

```
def rotation_90(image):
    nb_lignes = image.shape[0]      # nombre de lignes
    nb_colonnes = image.shape[1]    # nombre de colonnes
    nb_couleurs = image.shape[2]    # nombre de couleurs (= 3)
    # on commence par créer un tableau de bonne dimension :
    Rotation = zeros((nb_colonnes, nb_lignes, nb_couleurs))
    for i in range(nb_lignes):
        for j in range(nb_colonnes):
            Rotation[nb_colonnes - j - 1][i] = image[i][j]
    return Rotation
```