Corrigé concours blanc

[1]:
```
from numpy import *
from matplotlib.pyplot import *
```

Q18

Chaque composante nécessite 1 octet, donc chaque pixel en nécessite 3. Ainsi, pour stocker une image de 48 millions de pixels, il faut $48 \times 3 = 144$ Mo

Q 19 On considère que val est un nombre flottant

[2]:
```
def Clinear(val):
    if val <= 0.04045:
        clin = val / 12.92
    else:
        clin = ((val + 0.055) / 1.055)**2.4
    return clin
```

Q 20 Un première réponse, en utilisant directement la fonction précédente

[3]:
```
def Y(pix):
    r, v, b = pix  # assignation multiple
    rlin = Clinear(r)
    vlin = Clinear(v)
    blin = Clinear(b)
    ylin = 0.2126 * rlin + 0.7152*vlin + 0.0722*blin
    if ylin <= 0.0031308:
        y = 12.92 * ylin
    else:
        y = 1.055*ylin**(1/2.4) - 0.055
    return y
```

Une deuxième version qui prend en compte le fait que les formules de l'énoncé semblent faire référence à des flottants (un entier $\leqslant 0.04045$?). On converti donc nos composantes en flottant entre 0 et 1 en les divisant par 255

[4]:
```
def Y2(pix):
    # on considère que pix est un vecteur numpy
    r, v, b = pix / 255  # assignation multiple
    rlin = Clinear(r)
    vlin = Clinear(v)
```

```
    blin = Clinear(b)
    ylin = 0.2126 * rlin + 0.7152*vlin + 0.0722*blin
    if ylin <= 0.0031308:
        y = 12.92 * ylin
    else:
        y = 1.055*ylin**(1/2.4) - 0.055
    return y
```

Q 21

[5]:
```
def NiveauxGris(I):
    # on crée une nouvelle image, ce qui n'était pas forcément demandé.
    gris = zeros((len(I), len(I[0])))
    # compatible avec numpy ou une liste de liste
    for i in range(len(I)):
        for j in range(len(I[0])):
            gris[i][j] = Y(I[i][j])
    return gris
```

[6]:
```
# un petit test
import matplotlib.image as mpimg
image = mpimg.imread("nuages.png")
image.dtype
```

[6]:
```
dtype('float32')
```

Chaque pixel de l'image lue est déjà un flottant, et on peu constater que les valeurs sont entre 0 et 1

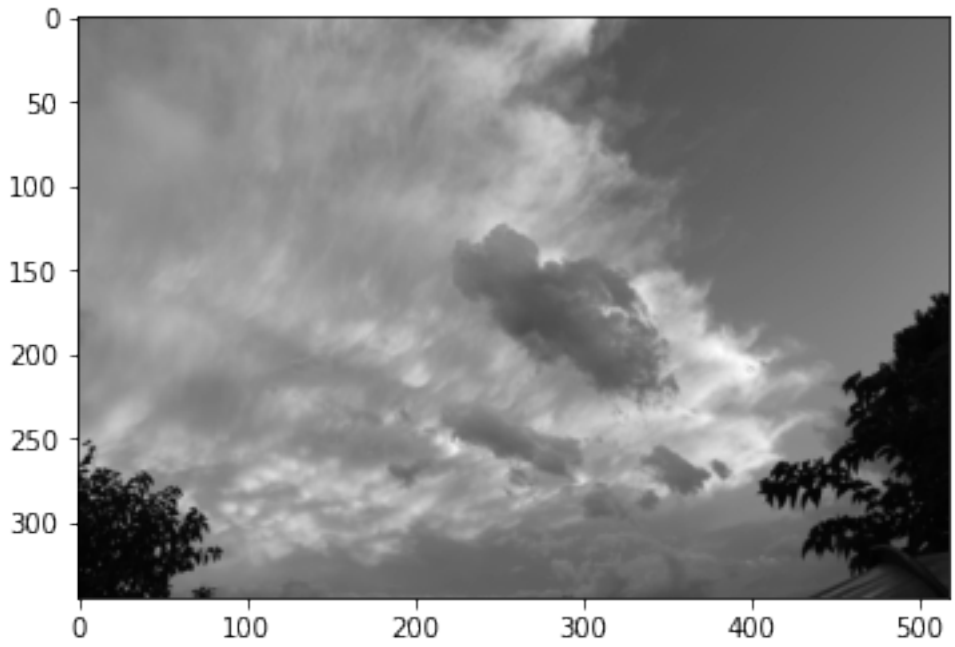
[7]:
```
image[0:2, 0:2]
```

[7]:
```
array([[[0.36078432, 0.3372549 , 0.3529412 ],
        [0.36078432, 0.3372549 , 0.34901962]],

       [[0.3647059 , 0.34509805, 0.3529412 ],
        [0.37254903, 0.34509805, 0.35686275]]], dtype=float32)
```

[8]:
```
imshow(NiveauxGris(image), cmap="gray")
```

[8]:
```
<matplotlib.image.AxesImage at 0x7f80a77cb7b8>
```



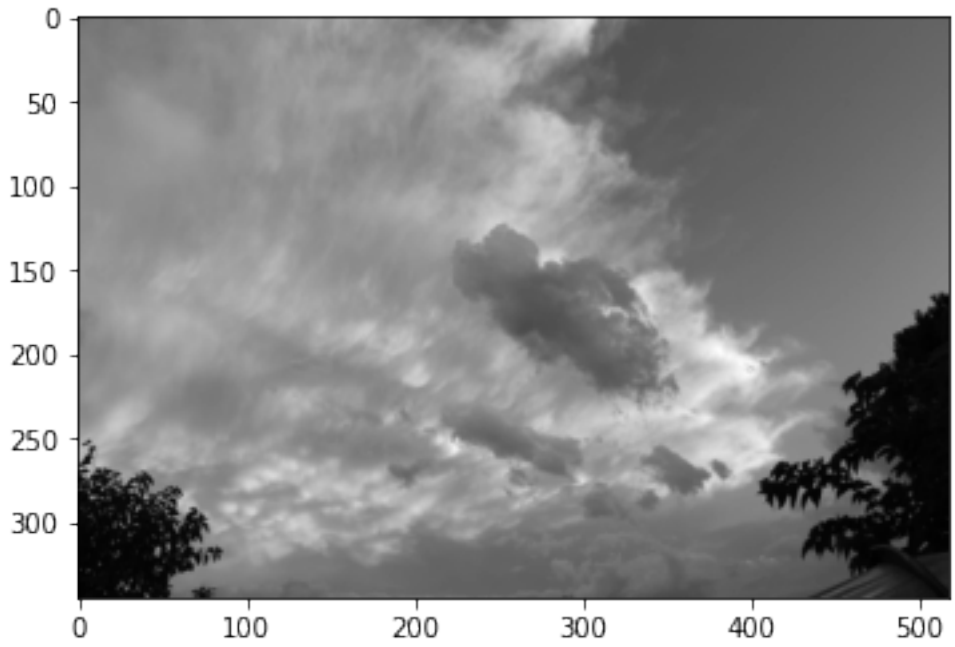
Si les données lues sont des entiers et que l'on souhaite obtenir des niveaux de gris entiers, on utilisera plutôt

[9]:
```
def NiveauxGris_entiers(I):
    # on considère que I est un vecteur numpy ici
    gris = np.zeros(I.shape[:2], dtype=uint8)
    for i in range(len(I)):
        for j in range(len(I[0])):
            gris[i][j] = int(Y2(I[i][j]) * 255)
    return gris
```

[10]:
```
gris = NiveauxGris(image)
gris_entiers = NiveauxGris_entiers(image*255)
```

[11]:
```
imshow(gris_entiers, cmap="gray")
```

[11]:
```
<matplotlib.image.AxesImage at 0x7f80a98cf128>
```



Q 22

On applique directement la formule de l'énoncé

[12]:
```
def convolution(A, B):
    # version liste de liste
    s = 0
    for i in range(3): # les matrices sont de taille 3
        for j in range(3):
            s = s + A[i][j]*B[i][j]
    return s

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

[13]:
```
A = random.random((3, 3)) # numpy.random.random
B = random.random((3, 3))
convolution(A, B) == convolution2(A, B)
```

[13]:
```
True
```

Q 23

[14]:
```
# on définit les matrices une fois pour toute
Gx = array([[-1, 0, 1], [-2,0,2], [-1,0,1]])
Gy = array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

# version liste de liste
def contraste_pixel(I, i, j):
    sous_image = []
    for x in range(i-1, i+2):
        ligne = []
        for y in range(j-1, j + 2):
            ligne.append(I[x][y])
        sous_image.append(ligne)
    c1 = convolution(sous_image, Gx)
    c2 = convolution(sous_image, Gy)
    return int(sqrt(c1**2 + c2**2))
```

[16]:
```
#  version numpy
def contraste_pixel(I, i, j):
    sous_image = I[i-1: i+2, j-1: j+2] # selection des lignes et colonnes␣
 ↪voulues
    c1 = convolution(sous_image, Gx)
    c2 = convolution(sous_image, Gy)
    return int(sqrt(c1**2 + c2**2))
```

[17]:
```
[contraste_pixel(gris, 10, j) for j in range(1, 20)]
```

[17]:
```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Comme on peut s'y attendre avec des images stockées sous forme de flottants, le contraste calculé est nul partout, il faut ici utiliser l'image stockée sous forme d'entiers.

[18]:
```
[contraste_pixel(gris_entiers, 10, j) for j in range(1, 20)]
```

[18]:
```
[2, 4, 4, 3, 3, 5, 4, 4, 3, 3, 7, 9, 5, 1, 2, 3, 8, 10, 8]
```

Q 24

On calcule d'abord la somme des constrastes puis on divise par le nombre d'éléments sommés.

[19]:
```
def contraste(I):
    cref = 0
    n = len(I) # nombre de lignes
    p = len(I[0]) # nombre de colonnes
    nb_elt = (n - 2) * (p - 2)  # nombre de pixels pris en compte
    for i in range(1, n - 1): # on évite les bords
        for j in range(1, p - 1):
            cref = cref + contraste_pixel(I, i, j)
    return cref / nb_elt
```

[20]:
```
contraste(gris), contraste(gris_entiers)
```

[20]:
```
(0.008164395326236806, 21.341576830067574)
```

Q 25

A priori, reglage ne dépend d'aucun paramètre et n'a pas besoin de communiquer de valeur de retour, car le focus aura été fait. Ainsi la valeur de retour sera **None**

[21]:
```
def reglage():
    pas = 0
    position_objectif(pas)
    image = prise()
    cref = contraste(NiveauxGris(image))
    maxi = -1  # nos contrastes seront toujours positifs
    # stocke la valeur maximale obtenue jusqu'alors
    while pas < 1000 and cref > maxi:
        maxi = cref
        # avancer d'un pas et recalculer le contraste
        pas = pas + 1
        position_objectif(pas)
        image = prise()
        cref = contraste(NiveauxGris(image))
    # on suppose que le maximum existe, on est pas sorti de la boucle while par␣
 ↪la
    # première condition
    # reculons d'un pas, car le contraste n'a plus augmenté
    position_objectif(pas - 1)
    # pas vaut au moins 1 car on est sûr de passer une fois dans la boucle while
    # grâce à la valeur initiale de maxi.
```

Q 26

[22]:
```
# on utilise des "tranches"
def extraction(L1, L2, dec):
    if dec >= 0:
        return (L1[0:-dec], L2[dec:])
    return (L1[-dec:], L2[:dec])
```

[23]:
```
# version sans tranche de listes
def extraction2(L1, L2, dec):
    if dec >=0:
        dep1 = 0 # indice de départ dans L1
        fin1 = len(L1) - dec # fin dans L1, exclu
        dep2 = dec
        fin2 = len(L2)
    else:
        dep1 = -dec  # indice positif car dep < 0
```

```
        fin1 = len(L1)
        dep2 = 0
        fin2 = len(L2) + dec # < len(L2)
    L3 = [L1[i] for i in range(dep1, fin1)]
    L4 = [L2[i] for i in range(dep2, fin2)]
    return L3, L4
```

[24]:
```
extraction([1,2,3,4,5], [-1,-2,-3,-4,-5], 2)
```

[24]: ([1, 2, 3], [-3, -4, -5])

[25]:
```
extraction([1,2,3,4,5], [-1,-2,-3,-4,-5], -2)
```

[25]: ([3, 4, 5], [-1, -2, -3])

Q 27

On compare tous les éléments des listes un a un et on s'arrête dès qu'il y a une différence.

[26]:
```
def comparaison(L1, L2):
    i = 0
    n = len(L1)  # == len(L2)
    while i < n and L1[i] == L2[i]:
        i = i + 1
    return (i == n) # ceci s'évalue comme un booléen
```

Q 28 On va comparer les sous-listes obtenues pour toutes les valeurs de décalage données

[27]:
```
def recherche_decalage(L1, L2):
    for dec in range(-80, 81):
        sous_L1, sous_L2 = extraction(L1, L2, dec)
        if comparaison(sous_L1, sous_L2):
            return dec # on a trouvé des sous listes égales
    # si on sort du for, on a pas trouvé de correspondance
    return None
```

Q 29

Nous allons estimer le nombre de comparaisons entre éléments des listes (et pas les comparaisons entre entiers)

Le cas le plus favorable est celui où le décalage cherché est le premier testé (ici -80). Dans ce cas, les deux sous-listes testées sont de longueur $n-80$ et on doit comparer chacun des éléments de ces listes. On obtient $n-80$ comparaisons et la complexité est en $O(n)$ (linéaire en la taille des listes).

Le cas le pire est celui où on doit tester tous les décalages. On considère aussi le cas le pire pour la fonction comparaison, c'est à dire qu'on teste tous les éléments des sous-listes à chaque fois. Dans ce cas on effecue m appels à la fonction comparaison qui font tous de l'ordre de n comparaisons pour un complexité totale en $O(mn)$ (pour être précis, dans notre exemple, on effectue

$n-80+n-79+\cdots+n-1+n+n-1+\cdots+n-80$ comparaisons soit $161n-2\sum_{k=1}^{80}k$. Si on considère $m=2p+1$ décalages symétriques par rapport à 0, on obtient dans le cas général $mn-2\sum_{k=1}^{p}k=mn-p(p+1)$ comparaisons).

Q 30

Une première version numpy :

[28]:
```
def erreur(L1, L2):
    return sum((L1 - L2)**2) / len(L1)
```

Version liste

[29]:
```
def erreur_liste(L1, L2):
    s = 0
    n = len(L1)
    for i in range(n):
        s = s + (L1[i] - L2[i])**2
    return s / n
```

Q 31

On va calculer tous les décalage et chercher celui qui conduit à une erreur minimale

[30]:
```
def recherche_decalage2(L1, L2):
    # pas de valeur initiale évidente pour la variable contenant
    # le minimum actuel
    sous_L1, sous_L2 = extraction(L1, L2, -80)
    # minimum actuel
    mini = erreur_liste(sous_L1, sous_L2)
    # décalage correspondant à l'erreur minimale
    dec_mini = -80
    for dec in range(-79, 81):
        sous_L1, sous_L2 = extraction(L1, L2, dec)
        err = erreur_liste(sous_L1, sous_L2)
        if err < mini:
            mini = err
            dec_mini = dec
    return dec_mini
```

Q 32

La question semble vague. Attendons-nous une réponse basée sur le traitement numérique des données ? Sur le matériel utilisé ?

D'un point de vue traitement numérique, la deuxième méthode semble beaucoup plus rapide (mais sans connaître la longueur des capteurs CCD supplémentaires, ceci reste hypothétique). Dans la première méthode, et pour une photo de 48×10^6 pixels, on effectue au moins 28 opérations arithmérique par pixel (pour la transformation en gris, plus les convolution et la somme des contrastes).

Ceci donne 1296×10^6 opérations pour chacun des 1000 pas (au pire), soit environ $1,3 \times 10^{12}$ opérations. Ce nombre semble être hors de portée d'un appareil photo.

Q 33

On considère que l'alimentation des bobines est cohérentes avec le pas courant, ou plus précisément que **pas_actuel % 8** donne l'état actuel des variables IN dans le tableau de l'énoncé.

On remarque que les changements d'états présentés sont alternés dans le sens ou on change alternativement une sortie à vrai puis une autre à faux puis à vrai.... En particulier, pour les pas pairs on change vers la valeur vrai pour passer à un pas impair.

[31]:
```python
def faire_un_pas_positif(pas_actuel):
    # liste, dans l'ordre du tableau des sorties à changer pour passer à
    # l'état suivant
    # Pour passer de l'état i à l'état i + 1, on modifie a_changer[i]
    a_changer = [IN3, IN1, IN2, IN3, IN4, IN2, IN1, IN4]
    modif_sortie(a_changer[pas % 8], (pas % 2 == 0))
    # le booléen à affecter est vrai quand le pas est pair et faux sinon
    return pas_actuel + 1
```

Q 34

On prend comme convention de ne rien faire si le pas objectif n'est pas dans l'intervalle de validité.

[32]:
```python
def position_objectif(pas):
    if pas < 0 or pas > 100:
        return
    if pas < pas_courant:
        # on choisit la fonction et le pas de modification
        fonction = faire_un_pas_negatif
    else:
        fonction = faire_un_pas_positif
    actuel = pas_courant
    while actuel != pas:
        actuel = fonction(actuel)  # retourne la valeur du pas courant
```

Q 35

Une clé primaire est composée d'une ou plusieurs colonne et permet d'identifier de manière unique chaque ligne d'une table sur laquelle elle est définie.

Q 36

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

Q 37

```
SELECT nom, prenom
FROM photographes
```

```
    JOIN photos ON photographes.id = photos.idp
WHERE date = "20181111"
```

Q 38

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

Q 39

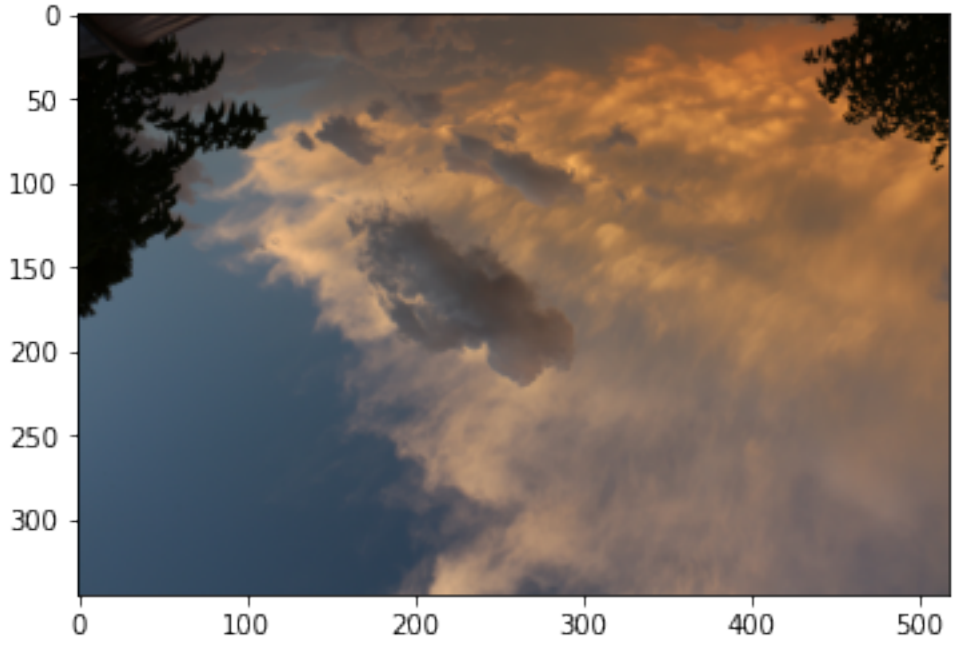
On note n le nombre de lignes de l'image et p son nombre de colonnes.

Dans une rotation de 180 degrés (autour du centre de la photo), le pixel de coordonnées (i, j) passe en position $(n-1-i, p-1-j)$.

[35]:
```
def rotation_180(image):
    cp = copy(image)
    n, p = image.shape[0:2]  # shape contient n, p, 3
    # qui sont les tailles successives des tableaux numpy imbriqués
    # alternative : n = len(image)
    # p = len(image[0]) car le premier élément est une ligne ed la photo.
    for i in range(n):
        for j in range(p):
            cp[n - 1 - i][p - 1 - j] = image[i][j]
    return cp
```

[36]:
```
imshow(rotation_180(image))
```

[36]:
```
<matplotlib.image.AxesImage at 0x7f80a9826eb8>
```



Q 40

Cette fois les nouvelles coordonnées sont $(p-1-j, i)$ (la première colonnes devient la dernière ligne, et ainsi de suite)

[37]:
```
def rotation_90(image):
    n, p = image.shape[0:2]
    cp = zeros((p, n, 3))
    for i in range(n):
        for j in range(p):
            cp[p - 1 - j][i] = image[i][j]
    return cp
```

[38]:
```
imshow(rotation_90(image))
```

[38]:
```
<matplotlib.image.AxesImage at 0x7f80a5974c50>
```