

MINISTÈRE DES ÉTUDES SUPÉRIEURES ET DE LA RECHERCHE SCIENTIFIQUE
ECOLE NATIONALE POLYTECHNIQUE D'ALGER
DÉPARTEMENT ÉLECTRONIQUE



TP 1 Systèmes audio-visuels : Manipulation d'images et traitements ponctuels

NENNOUCHE Mohamed
ATCHI Abdel Malek

29 décembre 2021

Introduction

Le but de ce TP est la prise en main de l'outil **OpenCV** sous environnement python pour le traitement des images, ainsi ce TP portera sur des manipulations de bases ainsi que du traitement ponctuel d'un jeu d'images. On importe ainsi au début de notre code les modules correspondant :

```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
import cmath
import os
```

Manipulation 1

On a les trois vecteurs suivants :

$$A = (0 \ 4 \ 8 \ 12 \ \dots \ 252)$$

$$B = (0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0)$$

$$C = (0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0)$$

Pour la création de ces vecteurs on utilise **Numpy** avec les méthodes **np.array()** et **np.append()**, alors :

- Pour les vecteurs B et C (étant assez petits), on les crée manuellement avec la méthode **np.array()**.
- Pour le vecteur A, étant un vecteur plus grand le choix d'une boucle est logique, alors on crée un vecteur vide :

```
A = np.array([])
```

Ensuite on utilise la méthode **np.append()** pour ajouter des éléments à la fin du vecteur.

On aura le résultat suivant :

```
In [33]: runfile('C:/Users/pc/.spyder-py3/temp.py', wdir='C:/Users/pc/.spyder-py3')
A = [ 0.  4.  8.  12.  16.  20.  24.  28.  32.  36.  40.  44.  48.  52.
      56.  60.  64.  68.  72.  76.  80.  84.  88.  92.  96.  100.  104.  108.
     112.  116.  120.  124.  128.  132.  136.  140.  144.  148.  152.  156.  160.  164.
     168.  172.  176.  180.  184.  188.  192.  196.  200.  204.  208.  212.  216.  220.
     224.  228.  232.  236.  240.  244.  248.  252.]
B = [0 0 0 0 0 0 0]
C = [0 0 1 1 1 1 0 0]
```

Création des matrices I₁, I₂ et I₃

Ainsi pour avoir les trois matrices I₁, I₂ et I₃ on écrit le code suivant :

```
# Création des matrices I1, I2 et I3
#I1
I1 = A
for i in range(0,63) :
    I1 = np.vstack([I1,A])
#I2
I2 = np.array([B,B,C,C,C,C,B,B])
```

```
#I3
I3 = np.column_stack((I2, I2)) #ajouter les colonnes
I3 = np.vstack((I3, I3)) #ajouter les lignes
print("I1 = \n",I1)
print("Dimensions de I1 = ",I1.shape)
print("I2 = \n",I2)
print("I3 = \n",I3)
```

On aura alors :

- La matrice I₁ avec 64 lignes identiques au vecteur A :

```
I1 =
[[ 0.   4.   8. ... 244. 248. 252.]
 [ 0.   4.   8. ... 244. 248. 252.]
 [ 0.   4.   8. ... 244. 248. 252.]
 ...
 [ 0.   4.   8. ... 244. 248. 252.]
 [ 0.   4.   8. ... 244. 248. 252.]
 [ 0.   4.   8. ... 244. 248. 252.]]
Dimensions de I1 = (64, 64)
```

- La matrice I₂ constitué d'un bloc 4x4 de '1' entouré de deux rangs de '0'

```
I2 =
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 1 1 1 1 0 0]
 [0 0 1 1 1 1 0 0]
 [0 0 1 1 1 1 0 0]
 [0 0 1 1 1 1 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]]
```

- La matrice I₃ étant la reproduction 2x2 de la matrice I₂

```
I3 =
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0]
 [0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0]
 [0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0]
 [0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

Conversion des trois matrices en entier et visualisation

Ainsi pour la transformation en matrice d'entiers non signés on écrit les lignes suivantes :

```
I1 = np.uint8(I1)
I2 = np.uint8(I2)
I3 = np.uint8(I3)
```

Premiere méthode de visualisation

En écrivant le code suivant :

```
cv.imshow("Matrice I1", I1)
cv.waitKey(0)

cv.imshow("Matrice I2", I2)
cv.waitKey(0)

cv.imshow("Matrice I3", I3)
cv.waitKey(0)

cv.destroyAllWindows()
```

Cette première méthode utilisant **OpenCV**, elle affiche les images mais à taille réelle et donc dans notre cas c'est quasiment indiscernable.

Deuxième métho de visualisation

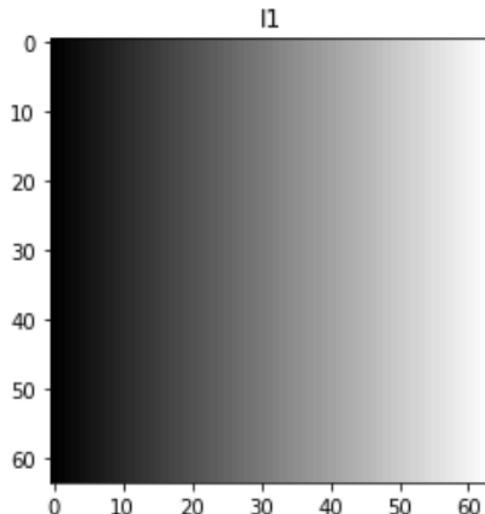
```
plt.imshow(I1, cmap='gray')
plt.title("I1")
plt.show()

plt.imshow(I2, cmap='gray')
plt.title("I2")
plt.show()

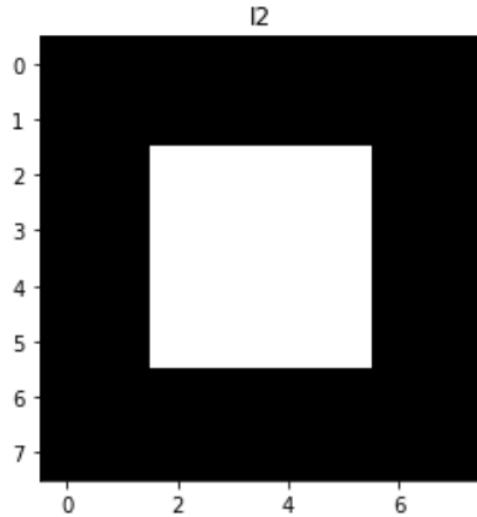
plt.imshow(I3, cmap='gray')
plt.title("I3")
plt.show()
```

Cette méthode utilise pour cette fois la bibliothèque **Matplotlib**, spécialisée dans la visualisation, ainsi on peut observer les images plus convenablement, alors on aura :

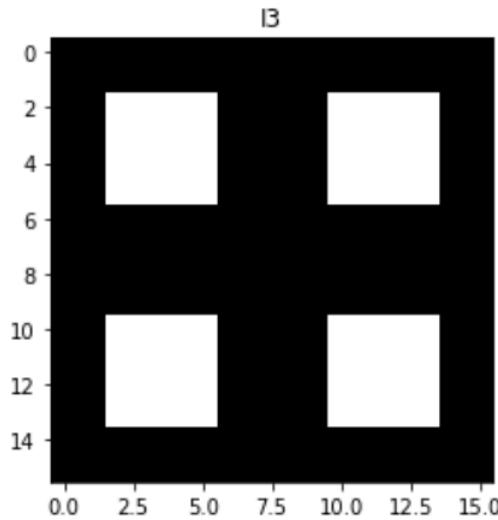
— L'image générée par la matrice I_1 :



— L'image générée par la matrice I_2 :



— L'image générée par la matrice I_3 :



On remarque bien que c'est les résultats attendus :

- L'image générée par I_1 est un dégradé, étant constitué de lignes allant de 0 (noir) à 252 (quasiment du blanc en base 8), le résultat est ce qui était escompté.
- L'image générée par I_2 est aussi un résultat attendu, étant des valeurs binaire (0 ou 1), donc en nuance de gris (**cmap='gray'**) le 0 représente la borne inférieure (noir) et le 1 représente la borne supérieure (blanc) et c'est ce qui est affiché.
- L'image générée par I_3 est la reproduction en 2x2 de l'image précédente et c'est ce qu'on peut observer effectivement.

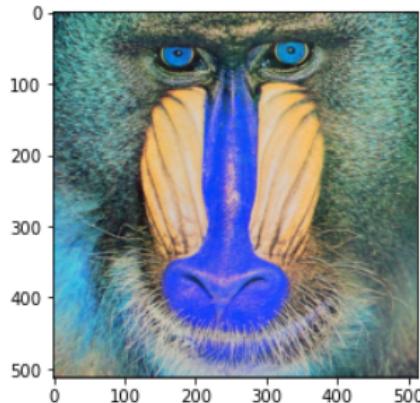
Manipulation 2

Lecture et visualisation de l'image

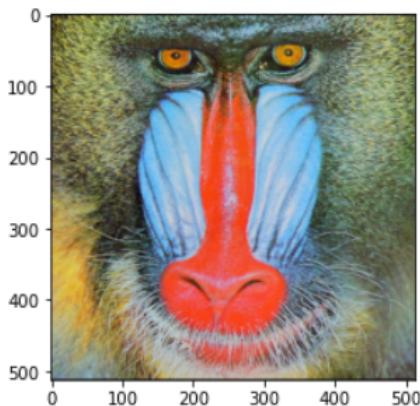
Alors on écrit le code suivant :

```
img = cv.imread('images/mandrill.png')
plt.imshow(img)
img = cv.cvtColor(img, cv.COLOR_BGR2RGB)
plt.imshow(img)
```

Alors on affiche dans un premier temps l'image sans inverser les canneaux des couleurs (donc dans l'ordre B->G->R) et on a le résultat suivant :



En arrangeant les canneaux de couleurs (R->G->B) et on retrouve l'image originale :



Taille et profondeur de l'image

Ainsi on affiche ces informations dans la console avec le code suivant :

```
print("La taille de l'image est : ",img.shape)
print("La profondeur de l'image est : ",img.dtype)
```

On a alors les résultats suivants :

```
La taille de l'image est : (512, 512, 3)
La profondeur de l'image est : uint8
```

- Par rapport à la taille on a alors une image de 512px de large et 512px de hauteur, le 3 c'est le nombre de canneaux (dans notre cas 3 canneaux R, G et B).
- Par rapport à la profondeur on a uint8.

Les valeurs de l'image sont de type uint8

Ils sont de types uint8 car elles sont codées sur 8 bits (chacun des canneaux) et donc 256 niveaux.

Visualisation de chaque canal indépendamment

On écrit le code suivant pour se faire :

```
# On split l'image faisant ressortir les 3 canaux
channels = cv.split(img)

# On crée un canal nul (remplie que de 0 pour substituer l'absence des autres canaux)
zero_channel = np.zeros_like(channels[0])

# On crée les images en fusionnant le canal et substituant les autres avec des canneaux nul
blue_img = cv.merge([zero_channel, zero_channel, channels[2]])
green_img = cv.merge([zero_channel, channels[1], zero_channel])
red_img = cv.merge([channels[0], zero_channel, zero_channel])

# Canal rouge
plt.subplot(1,3,1)
plt.imshow(channels[0],cmap = 'gray')
plt.title("ROUGE")

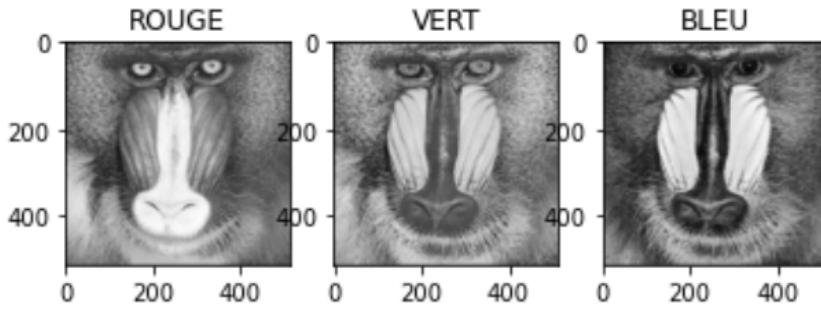
# Canal Vert
plt.subplot(1,3,2)
plt.imshow(channels[1],cmap = 'gray')
plt.title("VERT")

# Canal bleu
plt.subplot(1,3,3)
plt.imshow(channels[2],cmap = 'gray')
plt.title("BLEU")
plt.show()

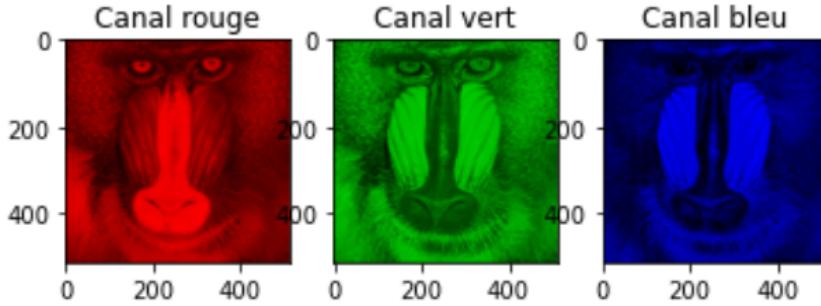
plt.subplot(1,3,1)
plt.imshow(red_img)
plt.title("Canal rouge")
plt.subplot(1,3,2)
plt.imshow(green_img)
plt.title("Canal vert")
plt.subplot(1,3,3)
plt.imshow(blue_img)
plt.title("Canal bleu")
plt.show()

cv.waitKey(0)
```

Alors les résultats sont les suivants, en affichant les canaux seuls (en niveaux de gris) :



En substituant les autres canaux par des canaux nuls (pour les visualiser en couleur), on observe :



Comment apparaissent les yeux du mandrill sur les 3 composantes RGB

- Sur la composante rouge, les yeux sont quasiment blanche ce qui indique que les yeux contiennent une grande proportion de rouge.
- Sur la composante verte, on voit bien que c'est en gris, donc la proportion de vert est modérée.
- Dans la composante bleue, on voit bien que c'est en noir et donc la présence de bleu est très faible dans les yeux du mandrill.

Ainsi les yeux du mandrill sont orange qui est un mélange de rouge et de vert en grande proportion et on y ajoute un peu de bleu. Par conséquent les résultats étaient ce qu'on attendait.

Les différentes combinaisons possibles de RGB

On a 6 combinaisons différentes (RGB, RBG, GRB, GBR, BRG, BGR). Ainsi on écrit le code suivant pour les afficher :

```
# On crée les différentes images en utilisant les canaux déjà fait
GRB_img=cv.merge([channels[1], channels[0], channels[2]])
GBR_img=cv.merge([channels[1], channels[2], channels[0]])
BRG_img=cv.merge([channels[2], channels[0], channels[1]])
RBG_img=cv.merge([channels[0], channels[2], channels[1]])
BGR_img=cv.merge([channels[2], channels[1], channels[0]])
RGB_img=cv.merge([channels[0], channels[1], channels[2]])

# On visualise les images
plt.subplot(2,3,1),plt.imshow(GRB_img),plt.title("GRB")
plt.subplot(2,3,2),plt.imshow(GBR_img),plt.title("GBR")
plt.subplot(2,3,3),plt.imshow(BRG_img),plt.title("BRG")
plt.subplot(2,3,4),plt.imshow(RBG_img),plt.title("RBG")
plt.subplot(2,3,5),plt.imshow(BGR_img),plt.title("BGR")
plt.subplot(2,3,6),plt.imshow(RGB_img),plt.title("RGB")
```

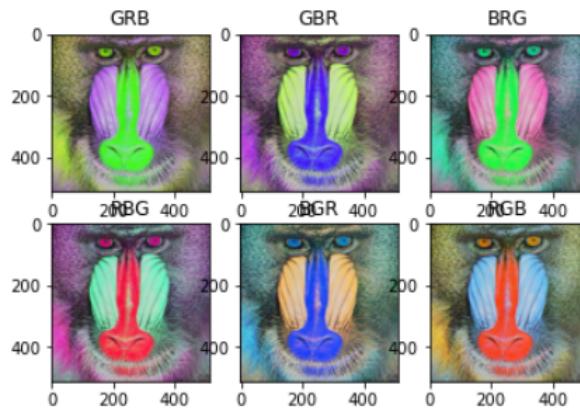
```

plt.show()

cv.waitKey(0)

```

On aura alors le résultat suivant :



On voit bien l'importance de l'ordre des différents canaux lors de la conception de l'image.

Stockage des différents canaux dans différents fichiers

Pour se faire le code python est très simple :

```

cv.imwrite('Red.png',red_img)
cv.imwrite('Green.bmp',green_img)
cv.imwrite('Blue.jpeg',blue_img)
cv.waitKey(0)

```

On a alors le résultat escompté :

Green.bmp	②	22/11/2021 21:34	Fichier BMP	769 Ko
Blue.jpeg	②	22/11/2021 21:34	Fichier JPEG	131 Ko
Red.png	②	22/11/2021 21:34	Fichier PNG	300 Ko

Réalisation d'un passage en niveau de gris

On utilise la formule suivante :

$$luminance = 0.2126 \cdot R + 0.7152 \cdot V + 0.0722 \cdot B$$

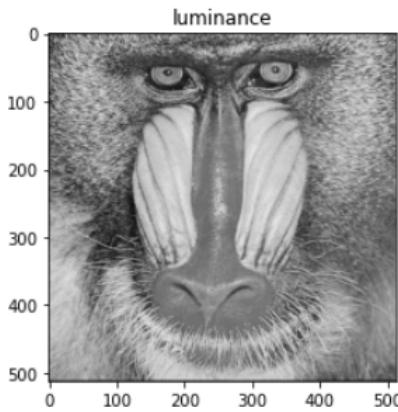
Qu'on traduit avec le code suivant :

```

limun = 0.2126*channels[2]+0.7152*channels[1]+0.0722*channels[0]
plt.title("luminance")
plt.imshow(limun,cmap = 'gray')
plt.show()
cv.waitKey(0)

```

On aura alors l'image en niveaux de gris :

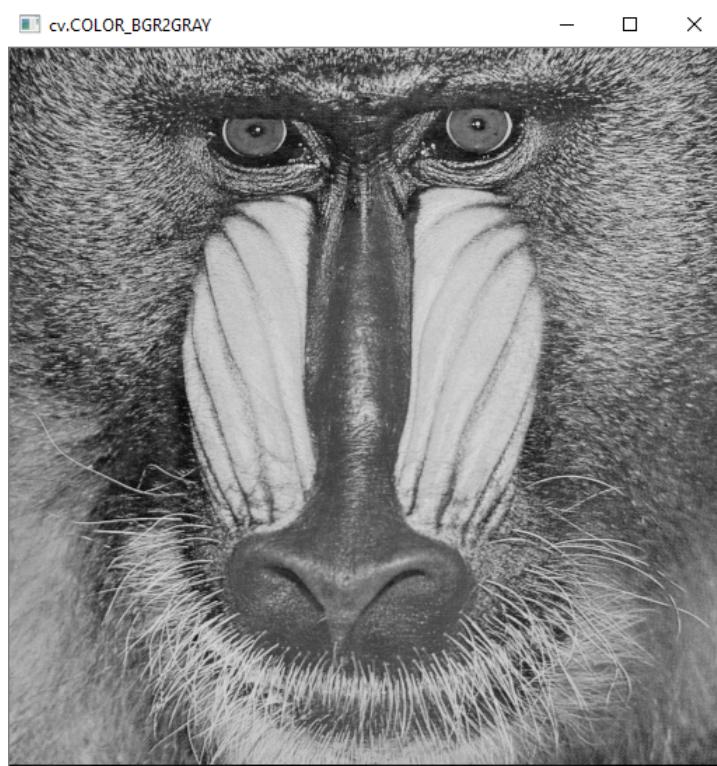


Transformation en niveaux de gris utilisant la fonction par défaut

On écrit le code suivant :

```
img_gray=cv.cvtColor(img,cv.COLOR_BGR2GRAY)
cv.imshow('cv.COLOR_BGR2GRAY', img_gray)
cv.waitKey(0)
```

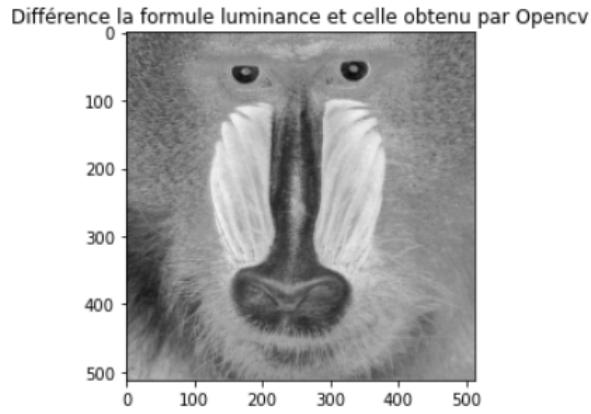
On a alors le résultat suivant :



On voit bien que le résultat est très similaire au résultat retrouvé précédemment. Maintenant en faisant une différence entre les deux résultats et on les affiche avec le code suivant :

```
Diff=img_gray-limun
plt.title("Différence la formule luminance et celle obtenu par OpenCV")
plt.imshow(Diff,cmap = 'gray')
plt.show()
```

On observe alors :



On voit bien que la différence entre les deux est légère, et réside dans les coefficients utilisés pour chaque canal.

Manipulation 3

Changer le pas de quantification de l'image pour obtenir 128, 64, 32, 16, 8, 4 et 2 niveaux de gris

Alors tout d'abord on charge l'image à partir du dossier et on la met en niveaux de gris :

```
img = cv.imread("images/zelda.png")
img = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
```

Ensuite on écrit la fonction suivante, qui permet de changer la quantification de notre image :

```
def quant(imgf, n_niv):
    n_itv = imgf.max()/n_niv
    img0 = imgf.copy()
    h = imgf.shape[0]
    w = imgf.shape[1]
    img0 = np.float32(img0)
    for i in range(h):
        for j in range(w):
            if img0[i, j] == imgf.max():
                img0[i, j] = (img0[i, j]//n_itv)-1
            else:
                img0[i, j] = (img0[i, j]//n_itv)
            img0[i, j] = (img0[i, j]*255)/(n_niv-1)
    image = np.uint8(img0)
    return image
```

Ainsi on calcule les différentes images (avec les différentes quantifications de niveaux de gris) en réutilisant notre fonction :

```
im2 = quant(img, 2)
im4 = quant(img1, 4)
im8 = quant(img1, 8)
im16 = quant(img1, 16)
```

```

im32 = quant(img1, 32)
im64 = quant(img1, 64)
im128 = quant(img1, 128)

```

Afficher l'image correspondante afin de déterminer le seuil minimal de quantification à partir duquel certains faux contours apparaissent

Ainsi on affiche les images avec le code suivant :

```

plt.figure(figsize=(20,7))
plt.subplot(241), plt.imshow(img1, cmap='gray'), plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(242), plt.imshow(im2, cmap='gray'), plt.title('Quantification 2 niveaux')
plt.xticks([]), plt.yticks([])
plt.subplot(243), plt.imshow(im4, cmap='gray'), plt.title('Quantification 4 niveaux')
plt.xticks([]), plt.yticks([])
plt.subplot(244), plt.imshow(im8, cmap='gray'), plt.title('Quantification 8 niveaux')
plt.xticks([]), plt.yticks([])
plt.subplot(245), plt.imshow(im16, cmap='gray'), plt.title('Quantification 16 niveaux')
plt.xticks([]), plt.yticks([])
plt.subplot(246), plt.imshow(im32, cmap='gray'), plt.title('Quantification 32 niveaux')
plt.xticks([]), plt.yticks([])
plt.subplot(247), plt.imshow(im64, cmap='gray'), plt.title('Quantification 64 niveaux')
plt.xticks([]), plt.yticks([])
plt.subplot(248), plt.imshow(im128, cmap='gray'), plt.title('Quantification 128 niveaux')
plt.xticks([]), plt.yticks([])
plt.show()

```

On a alors le résultat suivant :



On voit bien qu'à une quantification à 8 niveaux (sur 3 bits alors) on commence à voir l'apparition de faux contours et donc on fixe le seuil minimum pour la non détection de faux contours à 16 niveaux de gris (et donc un codage sur 4 bits).

Calculer et afficher l'histogramme de chaque quantification

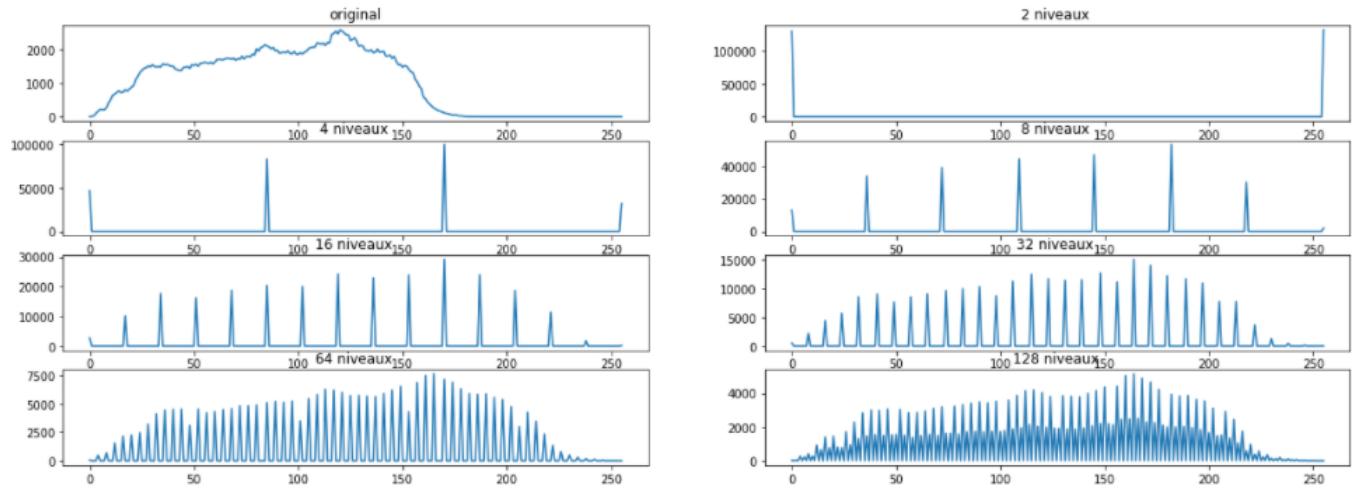
Alors on écrit le code suivant pour le calcul des histogrammes :

```

hist = cv.calcHist([img1], [0], None, [256], [0, 256])
hist2 = cv.calcHist([im2], channels=[0], mask=None,
                    histSize=[256], ranges=[0, 256])
hist4 = cv.calcHist([im4], channels=[0], mask=None,
                    histSize=[256], ranges=[0, 256])
hist8 = cv.calcHist([im8], channels=[0], mask=None,
                    histSize=[256], ranges=[0, 256])
hist16 = cv.calcHist([im16], channels=[0], mask=None,
                     histSize=[256], ranges=[0, 256])
hist32 = cv.calcHist([im32], channels=[0], mask=None,
                     histSize=[256], ranges=[0, 256])
hist64 = cv.calcHist([im64], channels=[0], mask=None,
                     histSize=[256], ranges=[0, 256])
hist128 = cv.calcHist([im128], channels=[0], mask=None,
                      histSize=[256], ranges=[0, 256])

```

On a donc l'affichage suivant qui indique effectivement les niveaux de gris présent :



Manipulation 4

Lecture et affichage de l'image flower.bmp et conversion de l'image en niveaux de gris

On écrit les codes comme précédemment et on se retrouve avec l'image qui suit :



Ainsi que l'image en niveaux de gris :

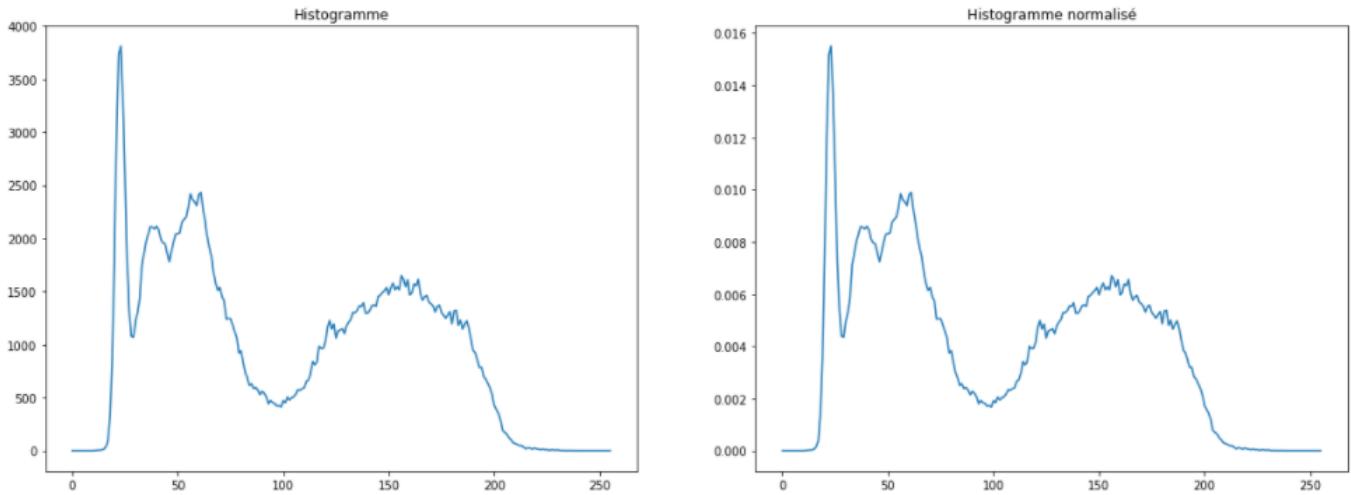


Calcul et affichage de l'histogramme ainsi que l'histogramme normalisé

On calcule et on affiche l'histogramme ainsi que l'histogramme normalisé comme précédemment avec le code suivant :

```
hist = cv.calcHist([img1], [0], None, [256], [0, 256])
histNorm = hist/(img1.shape[0]*img1.shape[1])
plt.figure(figsize=(20,7))
plt.subplot(1,2,1)
plt.plot(hist)
plt.title("Histogramme")
plt.subplot(1,2,2)
plt.plot(histNorm)
plt.title("Histogramme normalisé")
plt.show()
```

On retrouve alors l'histogramme ainsi que l'histogramme normalisé :



Commentaires

On peut observer aisément que l'image est assez déséquilibrer d'après la répartition des niveaux de gris dans notre histogramme, on voit bien une très grande présence des basses valeurs (indiquant des teintes sombres), ce qui peut s'expliquer par la photo en niveaux de gris qu'on a tels que tout le donc ainsi que l'abeille sont assez sombre, néanmoins on voit une valeur d'environ 155 qui est aussi énormément présente et qui est portée par la fleur qui est assez clair.

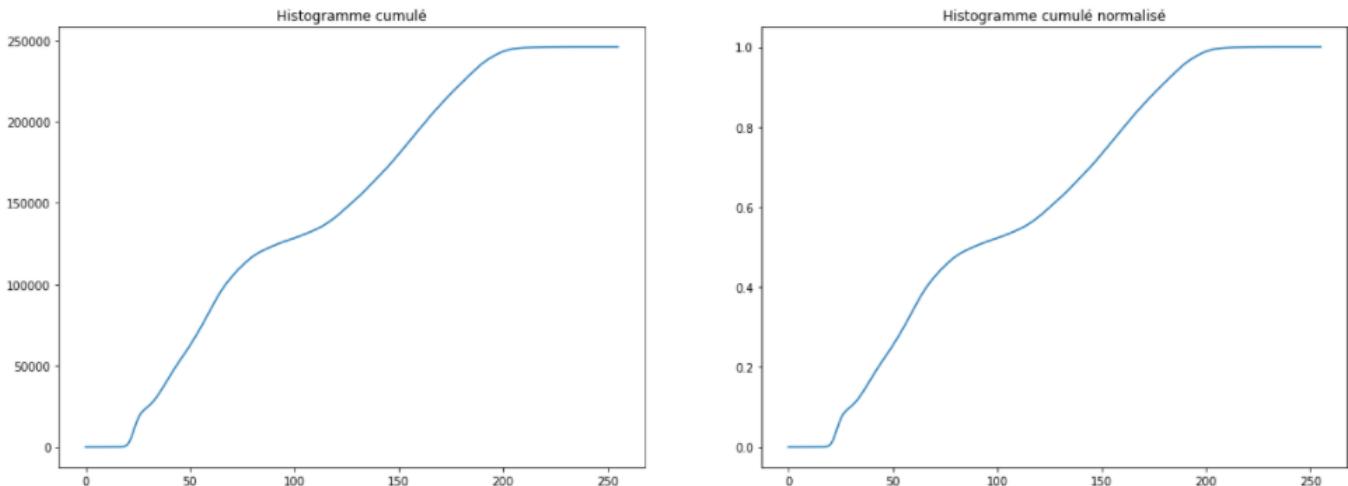
Calcul et affichage de l'histogramme cumulé

Pour se faire on écrit le code python suivant :

```
histcumul = np.zeros((256,1))
for i in range(256) :
    for j in range(256) :
        if j < i :
            histcumul[i] += hist[j]
histcumulNorm = histcumul/(img1.shape[0]*img1.shape[1])

plt.figure(figsize=(20,7))
plt.subplot(1,2,1)
plt.plot(histcumul)
plt.title("Histogramme cumulé")
plt.subplot(1,2,2)
plt.plot(histcumulNorm)
plt.title("Histogramme cumulé normalisé")
plt.show()
```

On a alors l'histogramme cumulé suivant :



Commentaires

On peut ajouter par rapport au commentaire précédent qu'on remarque que les valeurs de gris extrême (noir et blanc) sont complètement absents de notre image, cela s'observe par cet histogramme cumulé tels qu'il est plat dans les zones extrêmes.

Réalisation d'une égalisation de l'histogramme et affichage de l'histogramme égalisé

On procède maintenant à l'égalisation de l'histogramme avec la loi qu'on a abordé en cours qui est la suivante :

$$g(x, y) = \text{round}(g_{\max} H_{nc}(f(x, y)))$$

avec $f(x,y)$ l'image originale et $g(x,y)$ l'image égalisée. On écrit ainsi que le code python pour réaliser cette égalisation et on affiche l'histogramme ainsi que l'histogramme normalisé :

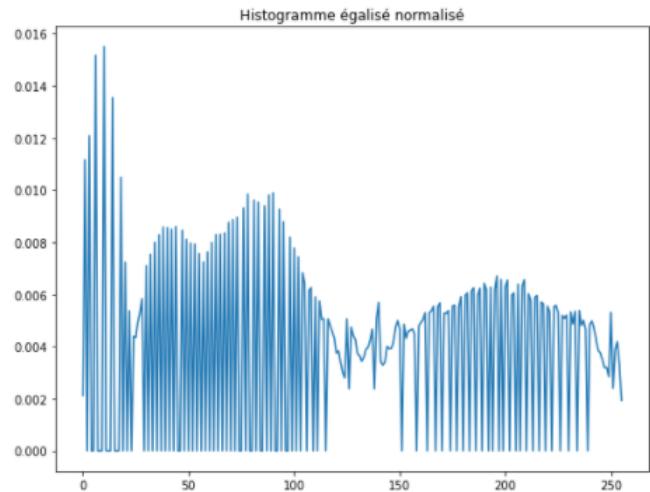
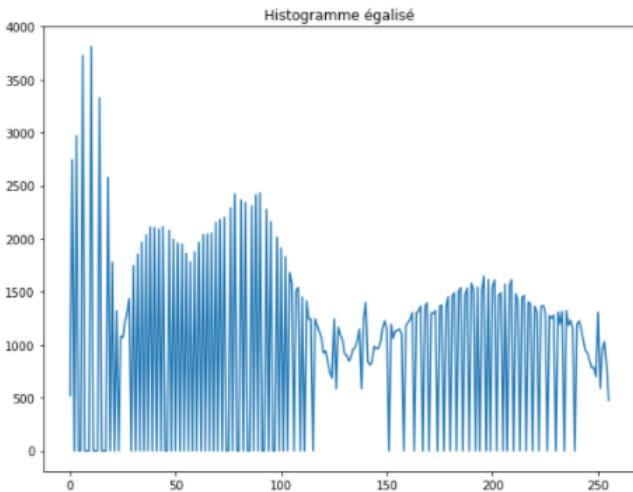
```

nouvellePhoto = np.zeros((480,512))
for i in range(480):
    for j in range(512):
        nouvellePhoto[i,j] = round(255*histcumulNorm[img1[i,j],0])
nouvellePhoto = np.uint8(nouvellePhoto)
histnouveau = cv.calcHist([nouvellePhoto], [0], None, [256], [0, 256])
histnouveauNorm = histnouveau/(img1.shape[0]*img1.shape[1])

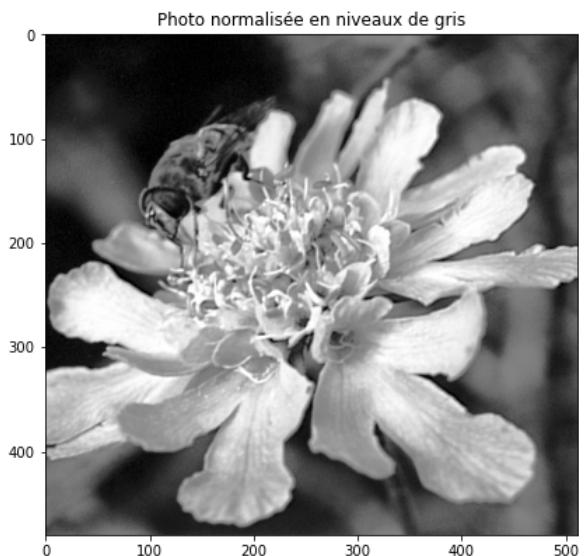
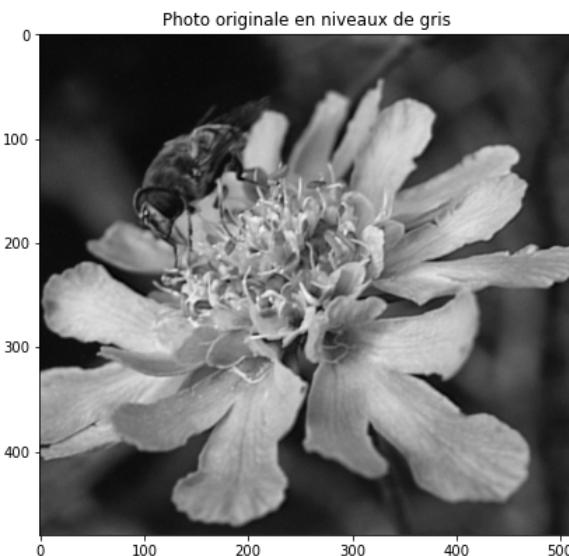
plt.figure(figsize=(20,7))
plt.subplot(1,2,1)
plt.plot(histnouveau)
plt.title("Histogramme égalisé")
plt.subplot(1,2,2)
plt.plot(histnouveauNorm)
plt.title("Histogramme égalisé normalisé")
plt.show()

```

On aura alors :



On compare les deux images avant et après égalisation et on a alors les deux images suivantes :



Commentaires

On observe effectivement que l'histogramme est bien égalisé, il y a une meilleure répartition des niveaux de gris, par contre et cela est dû à la fonction `round()` qui permet d'arrondir il y a des "trous" dans notre histogramme. On observe effectivement que l'égalisation en niveaux de gris est assez bonne, elle est plus clair et on discerne mieux les formes dans l'image égalisée, preuve que l'égalisation a bien été faite.

Comparaison avec l'égalisation réalisée par la méthode `cv.equalizeHist`

On utilise maintenant la fonction par défaut de OpenCV avec le code suivant :

```
nouvellePhoto2 = cv.equalizeHist(img1)
histnouveau2 = cv.calcHist([nouvellePhoto2], [0], None, [256], [0, 256])
histnouveau2Norm = histnouveau2/(img1.shape[0]*img1.shape[1])

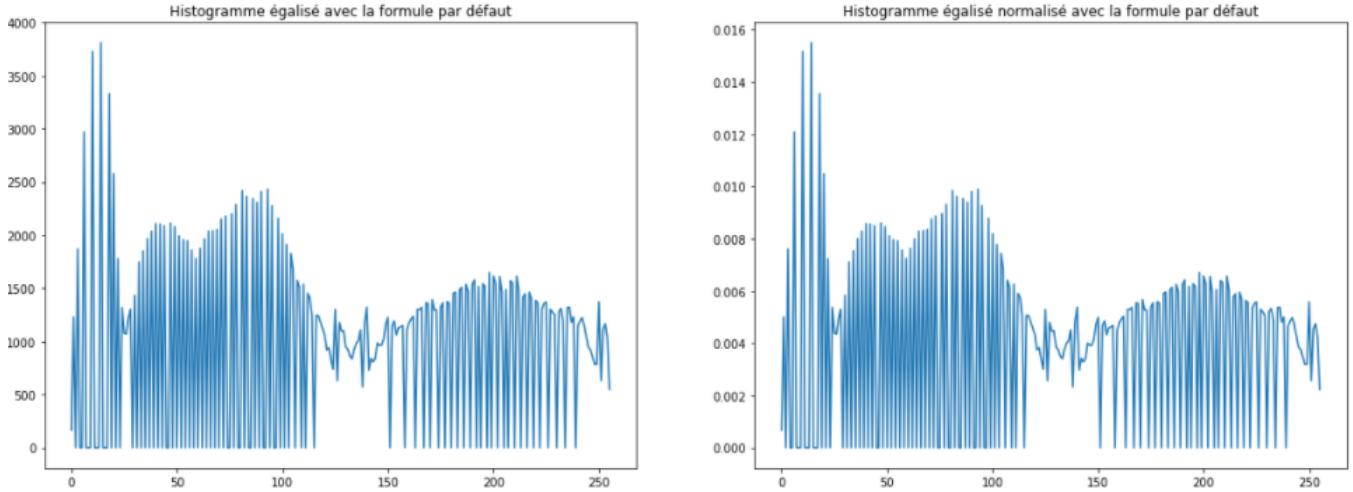
plt.figure(figsize=(20,7))
plt.subplot(1,2,1)
plt.plot(histnouveau2)
```

```

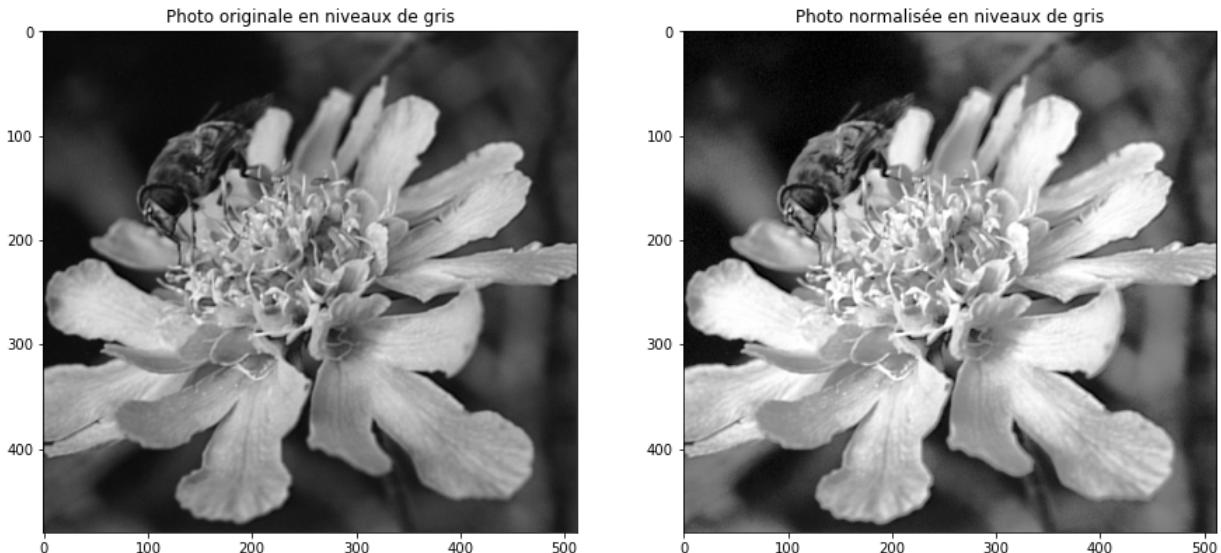
plt.title("Histogramme égalisé avec la formule par défaut")
plt.subplot(1,2,2)
plt.plot(histnouveau2Norm)
plt.title("Histogramme égalisé normalisé avec la formule par défaut")
plt.show()

```

On retrouve ainsi l'histogramme ainsi que l'histogramme normalisé :



On compare les deux images avant et après égalisation :



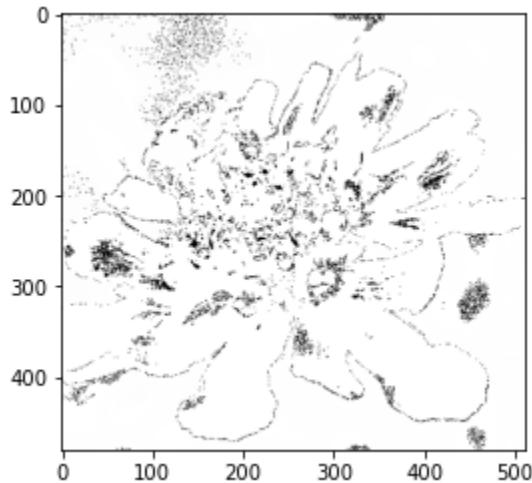
On observe effectivement que les deux images égalisées sont très similaires et on conforte cette idée en écrivant ce bout de code python les comparant :

```

difference = nouvellePhoto-nouvellePhoto2
plt.imshow(difference, cmap='gray')

```

Retrouvant ainsi la figure suivante :



Ce qui confirme que les deux égalisations sont assez similaires en niveaux de gris.

Égalisation des trois composantes indépendamment et affichage de l'image égalisée et calcul de son entropie

En utilisant la fonction par défaut d'OpenCV en écrivant le code suivant :

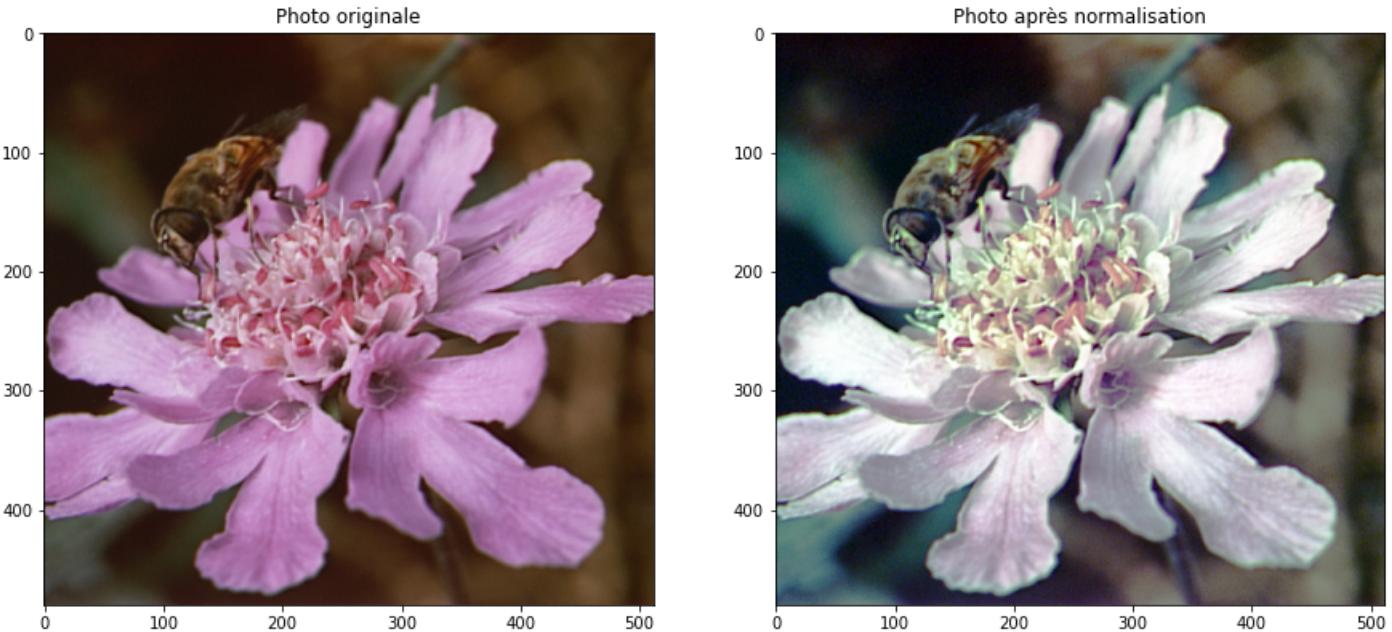
```
blue, green, red = cv.split(img)
B = cv.equalizeHist(blue)
G = cv.equalizeHist(green)
R = cv.equalizeHist(red)

new_image1 = cv.merge((B,G,R))
plt.axis("off")
plt.imshow(cv.cvtColor(new_image1, cv.COLOR_BGR2RGB))
```

On retrouve l'image suivante :



On compare avec l'image originale et on retrouve :



On voit bien que l'égalisation en utilisant la fonction par défaut d'OpenCV ne donne pas le résultat escompté, l'image a moins de couleur, elle est plus uniforme certe mais on perd la couleur rose de la fleur.

On calcule l'entropie de l'image égalisée en utilisant la loi étudiée en cours :

$$S = - \sum_{i=1}^n H_n(i) \log(H_n(i))$$

On écrit le code Python suivant pour l'implémenter

```
img1g = cv.cvtColor(new_image1, cv.COLOR_BGR2GRAY)
hist_img1 = cv.calcHist([img1g], [0], None, [256], [0,256])/(img.shape[0]*img.shape[1])
S1 = 0
for i in range(256):
    S1 = S1 - hist_img1[i]*math.log(hist_img1[i])
print("l'entropie après normalisation est : ", S1)
```

On retrouve :

l'entropie après normalisation est : [5.52032196]

Refaire la même manipulation mais en utilisant l'histogramme cumulé calculé précédemment

Pour se faire on écrit le code Python suivant :

```
imb = np.uint8(np.zeros((img.shape[0],img.shape[1])))
imr = np.uint8(np.zeros((img.shape[0],img.shape[1])))
img = np.uint8(np.zeros((img.shape[0],img.shape[1])))
for i in range(img.shape[0]):
    for j in range(img.shape[1]):
        imb[i,j] = np.round((histcumulNorm[blue[i,j]+1])*255)
        imr[i, j] = np.round((histcumulNorm[red[i, j] + 1]) * 255)
```

```

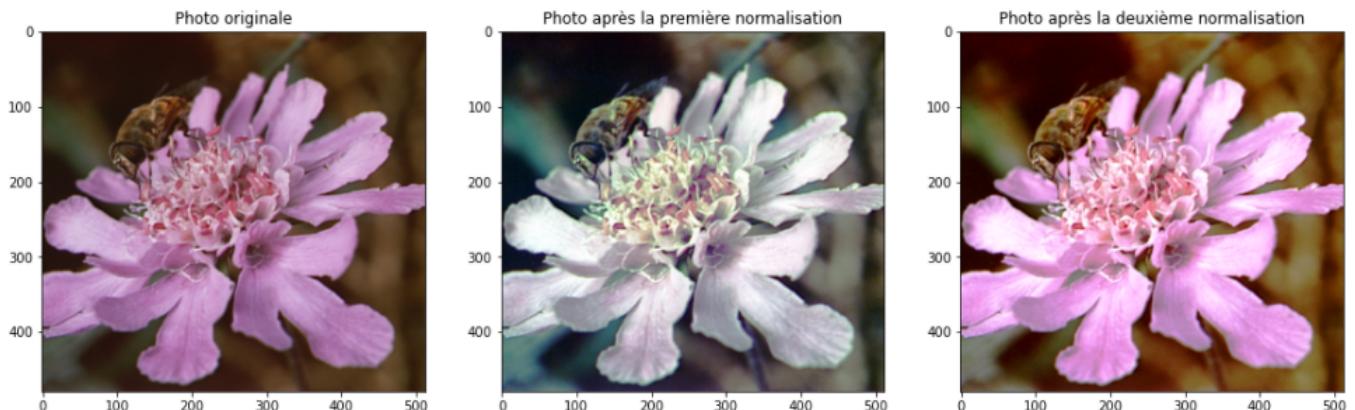
img[i, j] = np.round((histcumulNorm[green[i, j] + 1]) * 255)
im = cv.merge((imb, img, imr))
plt.axis("off")
plt.imshow(cv.cvtColor(im, cv.COLOR_BGR2RGB))

```

On retrouve l'image suivante :



Ce qui nous donne un bien meilleur résultat que précédemment, les couleurs sont plus flamboyantes. On a alors une image d'une meilleure qualité que l'originale, on compare les trois images (l'originale, avec la fonction par défaut et en utilisant l'histogramme normalisé cumulé calculé précédemment) et on retrouve :



On voit bien que la deuxième égalisation nous a donné une image de meilleure qualité. On calcule l'entropie de la même manière que précédemment et on retrouve :

l'entropie 2 est : [5.45161533]

Comparaison des deux entropies et commentaires

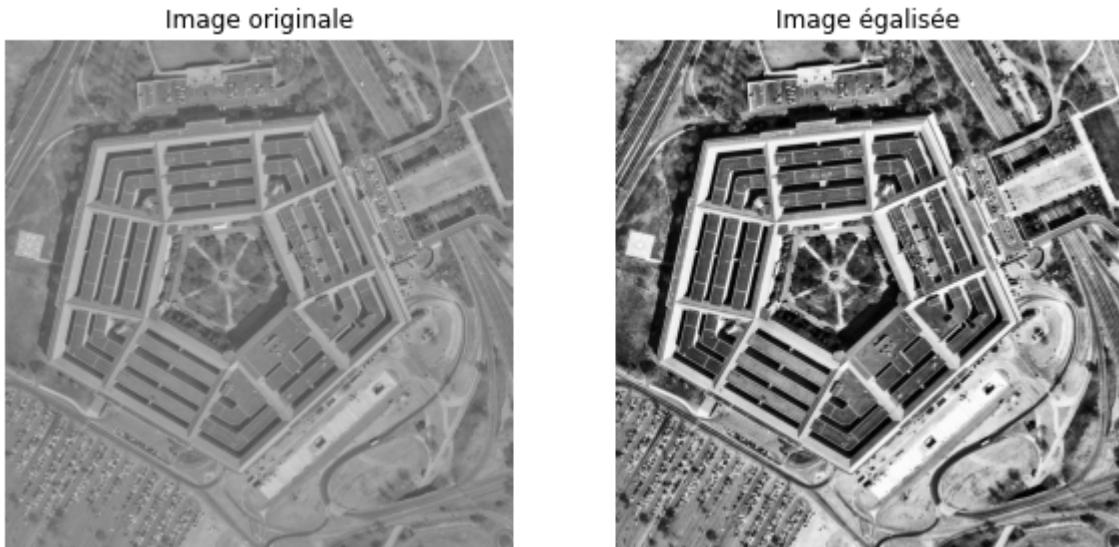
On observe effectivement que l'entropie de la première égalisation est supérieure à l'entropie au niveau de l'égalisation utilisant l'histogramme normalisé cumulé calculé précédemment ceci indique que la première égalisation est d'une meilleure qualité (vis-à-vis de l'histogramme) que la deuxième, car un histogramme totalement plat a pour conséquence une maximisation de l'entropie, mais par contre dans le rendu visuel, on observe que la deuxième égalisation a donné un meilleur résultat.

Lecture et visualisation de l'image ‘pentagon.tif’, ensuite égalisation de son histogramme

Ainsi comme précédemment on effectue toutes les étapes et on écrit ainsi le code Python suivant :

```
image = cv.imread('images/pentagon.tif')
gra = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
graE = cv.equalizeHist(gra)
imageE = cv.cvtColor(graE, cv.COLOR_GRAY2BGR)
fig,ax=plt.subplots(1,2,figsize=(10,6))
ax[0].axis("off")
ax[0].imshow(cv.cvtColor(image, cv.COLOR_BGR2RGB))
ax[0].set_title('Image originale')
ax[1].axis("off")
ax[1].imshow(cv.cvtColor(imageE, cv.COLOR_BGR2RGB))
ax[1].set_title('Image égalisée')
```

Ainsi on retrouve le résultat suivant :



Commentaires

On observe très bien qu'avant égalisation l'image est assez terne et manque de détails, après égalisation on observe une amélioration claire de l'image en terme de contraste et présence des différentes nuances de gris. Ainsi on a clairement améliorer le rendu visuel de notre image en égalisant son histogramme.