

Partie E - Algorithmique & programmation II

Chapitre n°3 VERSION COMPLÉTÉE Traitement d'images

L'objectif de ce cours, et du TD qui va avec, est d'aborder quelques algorithmes classiques de traitement d'images, et de les mettre en œuvre avec **Python**.

1 Les images informatiques

1.1 Format matriciel

Une manière de représenter informatiquement une image, est de l'échantillonner en petits éléments (picture element= pixel) supposés de couleur uniforme, suivant un quadrillage.

L'image est alors représentée informatiquement par une matrice de couleurs (bitmap), chaque case de la matrice donnant la couleur du pixel correspondant du quadrillage.

Les formats les plus connus d'images bitmap sont JPEG, BMP, GIF, TIFF et PNG.

La résolution de l'image dépend bien sûr de la finesse du quadrillage, c'est-à-dire de la taille de la matrice de couleurs. En zoomant suffisamment, on voit toujours apparaître la trame sous la forme d'une mosaïque de carrés de couleur unie (les pixels!).

On convient de numérotter les colonnes en partant de la gauche, et les lignes en partant du haut de l'image. Un pixel de la trame est repéré par un couple (i, j) où i et j sont respectivement les numéros de la ligne et de la colonne à l'intersection desquelles se trouve le pixel.

1.2 Les couleurs

Selon le format de l'image bitmap, la palette de couleurs est encodée sur un certain nombre de bits, appelée profondeur de l'image (exprimée en bpp = bits par pixel).

- ♠ Dans le cas d'une image en noir et blanc, chaque couleur (noir ou blanc) est encodée sur 1 bit : par exemple, on peut convenir que le bit 0 encode le noir et le bit 1 encode le blanc.
- ♡ Dans le cas des images en niveau de gris, chaque couleur (du noir au blanc en passant par des gris plus ou moins foncés) est encodée sur 1 octet.
(On rappelle qu'un octet est une suite de 8 bits, et que les nombres qui s'écrivent en binaire avec au plus 8 bits sont exactement les nombres entiers entre 0 et 255 en numérotation décimale. Le bit le plus à gauche d'un octet est le bit de poids fort, et celui le plus à droite est le bit de poids faible.)
Donc 256 nuances de gris pour ces images...
- ◇ On peut également encoder une image couleur avec 1 octet pour chaque pixel. Chaque valeur entre 0 et 255 représente une couleur de la palette.

- ♣ Nous considérerons quant à nous des images couleur au format RGB (Red, Green, Blue). Chaque pixel est décrit par une quantité de rouge, une quantité de vert, et une quantité de bleu. La couleur résultante est obtenue par synthèse additive (rouge+bleu+vert=blanc). On utilise 1 octet pour la quantité de rouge, 1 octet pour la quantité de vert, et 1 octet pour la quantité de bleu, si bien qu'une couleur est représentée sur 3 octets=24 bits. En pratique, on donne un triplet de nombres entre 0 et 255.

	(255, 0, 0)	rouge
	(0, 255, 0)	vert
	(0, 0, 255)	bleu
	(0, 0, 0)	noir
	(255, 255, 255)	blanc
Par exemple :	(0, 255, 255)	cyan
	(255, 0, 255)	magenta
	(255, 255, 0)	jaune
	(148, 129, 43)	kaki
	(0, 86, 27)	vert impérial
	(248, 142, 85)	couleur saumon
		etc.

On peut aussi donner ces 3 octets sous forme de 6 chiffres hexadécimaux (#FF0000 pour le rouge, #94812B pour le kaki, etc.)

Remarques. 1) Certaines images sont encodées en RGBA. Dans ce cas, un 4ème octet est utilisé pour coder la **transparence** de l'image (A pour "alpha") : de 0 pour un pixel totalement transparent à 255 pour un pixel totalement opaque. Nous n'en dirons pas plus. Il faut alors **32 bits** pour chaque pixel.

2) Sur une image en niveau de gris encodé en RGB, chaque pixel a ses quantités de rouge, de vert et de bleu égales (donc redondance d'informations...).

Question 1. Combien de couleurs sont possibles en format RGB ?

Solution: $2^{24} = 16\,777\,216$ (donc presque 17 millions)

Question 2. Combien d'images RGB de 1000 pixels par 1500 pixels différentes existe-t-il ?

Solution: $(2^{24})^{1000 \times 1500} = 2^{36\,000\,000} = (2^{10})^{3\,600\,000} \approx (10^3)^{3\,600\,000} = 10^{10\,800\,000}$

1.3 Du RGB au niveau de gris

Pour convertir une image couleur en niveau de gris, il faut remplacer, pour chaque pixel, les trois valeurs représentant les niveaux de rouge, de vert et de bleu, par une seule valeur représentant sa

luminance (sensation visuelle associée à la luminosité).

On définit la luminance d'un pixel à partir d'un triplet (R, G, B) de la manière suivante :

$$L = R \times \frac{299}{1000} + G \times \frac{587}{1000} + B \times \frac{114}{1000}$$

La somme des 3 coefficients est égale à 1, mais ces coefficients, très différents, tiennent compte de la manière dont l'œil humain perçoit les couleurs : une lumière verte apparaît plus claire qu'une lumière rouge, et encore plus qu'une lumière bleue.

2 En pratique

Rien qu'en **Python**, il existe de nombreux modules et sous-modules dédiés au traitement d'images, le module de référence étant PIL et son fork Pillow... que nous n'utiliserons pas ensemble !

2.1 Lecture

Pour lire et écrire des images (c'est-à-dire pour passer du fichier image à la matrice de couleurs), on utilisera en TD le module `imageio`.

Plus précisément, la commande `imread` de `imageio` renvoie un objet d'une classe contenant un tableau **numpy** (vu en MPSI!) et un champ contenant des informations annexes sur l'image. En pratique, on pourra utiliser tout ce qu'on a vu en MPSI sur les tableaux **numpy**.

```
import imageio
M = imageio.imread("mont_saint_michel.png")
print(M, type(M))
```

```
[[[ 79  69  77]
   [143 133 141]
   [136 126 134]
   ...,
   [133 133 121]
   [136 135 130]
   [ 90  85  91]]
 ...,
 [[129 124 120]
  [193 188 184]
  [196 191 187]
  ...,
  [183 188 184]
  [189 194 190]
  [147 153 149]]] <class 'imageio.core.util.Image'>
```

Il pourra être intéressant de transformer les objets `imageio` en "vrais" tableaux **numpy**, pour accélérer les programmes.

```
import imageio
import numpy as np
import matplotlib.pyplot as plt

monimage = imageio.imread("mont_saint_michel.png")
print(type(monimage)) # format imageio

monimage2 = np.array(monimage)
print(type(monimage2)) # format numpy
```

2.2 Manipulations

Le tableau **numpy** d'une image a 2 ou 3 dimensions.

Les images de profondeur 256 bpp n'ont que 2 dimensions (chaque pixel est encodé par un nombre entre 0 et 255), tandis que les images RGB et RGBA ont 3 dimensions (chaque pixel est encodé par 3 ou 4 nombres entre 0 et 255).

- La première dimension est la **hauteur** de l'image
(donc la première coordonnée est le numéro de la **ligne** du pixel).
- La deuxième dimension est la **largeur** de l'image
(donc la deuxième coordonnée est le numéro de la **colonne** du pixel).

Ainsi `M[45,118]` désigne le pixel ligne 45 et colonne 118 de l'image de matrice des couleurs `M`.

On peut récupérer la dimension de l'image avec la méthode **shape** de **numpy**.

```
import imageio
import numpy as np

image1 = imageio.imread("mont_saint_michel.png")
print(np.shape(image1))
image2 = imageio.imread("tux.png")
print(np.shape(image2))
```

```
(1016, 1519, 3)
(256, 256, 4)
```

On peut utiliser le slicing pour transformer une image RGBA en RGB.

```
import imageio
import numpy as np

monimage = imageio.imread("tux.png")
print(np.shape(monimage))

monimage2 = imageio.imread("tux.png")[:, :, :3]
print(np.shape(monimage2))
```



```
(256, 256, 4)
(256, 256, 3)
```

Pour modifier une image, il suffit de modifier sa matrice de couleurs, ou bien directement, ou bien en créant une matrice **numpy** possédant le même nombre de lignes et de colonnes que la matrice associée à l'image sur laquelle on travaille, et qu'on remplit par la suite (pour l'application de masques par exemple).

On rappelle comment créer un tableau **numpy** avec que des 0. Le type des éléments (de tous les éléments) du tableau est par défaut **float64** (flottant sur 64 bits).

```
import numpy as np

# création d'un tableau à 100 lignes et 200 colonnes
# dont chaque case est un tableau à 3 éléments
# et rempli initialement de zéros
A = np.zeros((100,200,3))

print(A.dtype)
```

```
float64
```

On dispose aussi de la fonction **ones...** vous rappelez-vous ?

On peut forcer le type d'un tableau. Les données qu'on insère dans les tableaux A et B ci-dessous sont automatiquement transtypées en entiers non-signés sur 8 bits.

```
import numpy as np

A = np.zeros((100,200,3),dtype=np.uint8)
print(A.dtype)
A[0,0,0]=3.65
print(A[0,0,0])

B = np.uint8( [ 3.14, 2.65, 1.27 ])
print(B)
```

```
uint8
3
[3 2 1]
```

2.3 Sauvegarde

Les modifications d'une matrice de couleurs non suivies d'une sauvegarde, n'affectent pas le fichier ouvert. Pour enregistrer des modifications d'une image, on utilisera la commande **imsave** du module **imageio**.

Par exemple, pour symétriser une image par rapport à un axe médian vertical et enregistrer (dans un autre fichier) le résultat, on peut faire...

```
import imageio
import numpy as np
monimage = imageio.imread("bambi.png")[:, :, :3]
N,M,_ = np.shape(monimage)
monimage_s = np.zeros((N,M,3),dtype=np.uint8) # image symétrique
for i in range(N):
    for j in range(M):
        monimage_s[i,j] = monimage[i,M-1-j]
imageio.imwrite("bambi_s.png",monimage_s)
```



2.4 Affichage

Si vous avez sauvegardé l'image obtenue dans un fichier, vous avez la possibilité de l'ouvrir avec un lecteur externe.

Plus simplement, on utilisera le sous-module `pyplot` de `matplotlib` (déjà étudié en MPSI) pour afficher les images.

Sans option, la commande `imshow` de `matplotlib.pyplot` permet d'afficher des images couleurs RGB et RGBA (à partir d'une matrice au format $N \times M \times 3$, $N \times M \times 4$ avec coefficients sur 8 bits).

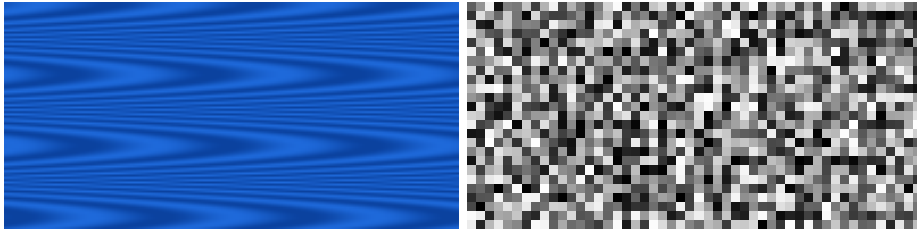
```
import numpy as np
import matplotlib.pyplot as plt

A = np.uint8 ([[
    [ 21 + 10*np.sin( (i**2+j)/10 ),\
      86 + 20*np.sin( (i**2+j)/10 ),\
      189 + 30*np.sin( (i**2+j)/10 ) ]\
for j in range(200) ] for i in range(100) ])

print(A)
plt.imshow(A)
plt.axis('off')
plt.show()
```

On pourra aussi afficher des images $N \times M$ en niveaux de gris sur 8 bits, avec l'option `cmap='gray'`.

```
import numpy as np
import matplotlib.pyplot as plt
A=np.random.randint(0,256,(25,50))
plt.imshow(A,cmap='gray')
plt.axis('off')
plt.show()
```



Cinquième partie - Algorithmique & programmation II

TD n°IV CORRIGÉ

Traitement d'images

Si vous travaillez avec **Pyzo**, comme on l'a déjà déjà fait plus tôt dans l'année, et en MPSI :
— donnez-vous un répertoire de travail dans lequel vous placerez les images-mystères ;
— sauvegardez un programme **Python** dans ce répertoire de travail, et exécutez-le une fois en tant que script (Ctrl+Shift+E) afin de changer le répertoire courant.

1 Création de quelques images simples

Question 1. Faites afficher avec **Python** le drapeau français, à partir d'un tableau **numpy** que vous aurez créé (on suppose que le drapeau est 50% plus large que haut, et que les 3 bandes de couleur sont de tailles égales).

Solution:

```
import matplotlib.pyplot as plt
import numpy as np

N=40
M=int(N*1.5)
L=M//3

drapeau = np.zeros((N,M,3),dtype=np.uint8)

for i in range(N):
    for j in range(L):
        drapeau[i,j]=[0,0,255]
    for j in range(L,2*(L)):
        drapeau[i,j]=[255,255,255]
    for j in range(2*(L),M):
        drapeau[i,j]=[255,0,0]

plt.imshow(drapeau,interpolation='nearest')
plt.axis('off')
plt.show()
```

On pouvait aussi penser à cette solution plus courte.

```
import matplotlib.pyplot as plt
import numpy as np

N=40
```

```

M=int(N*1.5)
L=M//3

drapeau = np.zeros((N,M,3),dtype=np.uint8)
drapeau[:, :L]=[[[255,0,0]]*L]*N
drapeau[:, L:2*L]=[[[255,255,255]]*L]*N
drapeau[:, 2*L:]=[[[0,0,255]]*L]*N

plt.imshow(drapeau,interpolation='nearest')
plt.axis('off')
plt.show()

```

Question 2. ★ Idem avec le drapeau japonais (on suppose que le disque a un diamètre égal aux $\frac{3}{5}$ de la hauteur du drapeau).

Solution:

```

import matplotlib.pyplot as plt
import numpy as np

N=200
M=int(N*1.5)
r=1/2*3/5.*N*1.

x=M//2
y=N//2

drapeau = 255*np.ones((N,M,3),dtype=np.uint8)
# le drapeau est blanc... plus qu'à dessiner le disque rouge

for i in range(N):
    for j in range(M):
        if (j-x)**2+(i-y)**2<=r**2:
            drapeau[i,j]=[255,0,0]

plt.imshow(drapeau,interpolation='nearest')
plt.axis('off')
plt.show()

```

Pour toute la suite, vous allez travailler avec une image couleur au format png (ou jpeg si ça fonctionne, taper au préalable `import PIL`) de votre choix trouvée sur internet ou amenée par vous, pas trop grosse pour les traitements (pas plus de 800 pixels de largeur et pas plus de 800 pixels de hauteur, ce serait bien...).

Vérifiez qu'à l'importation vous obtenez bien un tableau $N \times M \times 3$ (éventuellement $N \times M \times 4$), et que les couleurs sont bien codées sur 8 bits. Voici ce que vous devriez obtenir :

```

import imageio
import numpy as np

```

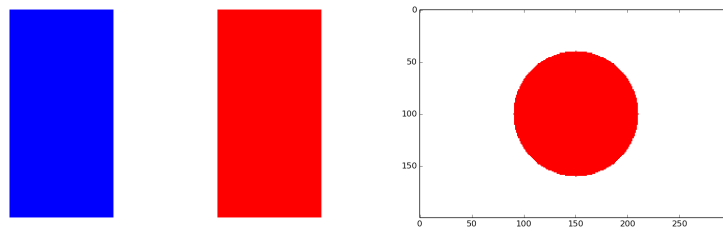


FIGURE 1 – Drapeau français et drapeau japonais

```
monimage = imageio.imread("tux.png")
print(np.shape(monimage), monimage.dtype)
monimage2 = imageio.imread("mont_saint_michel.png")
print(np.shape(monimage2), monimage2.dtype)
```

```
(256, 256, 4) uint8
(1016, 1519, 3) uint8
```

Evitez les dessins sur fond blanc... il pourrait y avoir de l'information cachée par transparence...

```
import imageio
import numpy as np
import matplotlib.pyplot as plt

plt.imshow ( list ( imageio.imread("licorne.png")[:, :, :3] ) )
plt.show()
```



FIGURE 2 – Les licornes cachées...

(L'objet `imageio` est plus complexe qu'un simple tableau **numpy** : il contient une information sur les contours du contenu de l'image. La conversion en liste permet d'“oublier” ce contour.)

2 Découpage et symétrie

Question 3. Enregistrer dans le fichier `coin_bas_gauche.png` le coin en bas à gauche de l'image de 200 pixels de hauteur par 300 pixels de largeur (utiliser le slicing vu en MPSI).

Solution:

```
import imageio
monimage = imageio.imread("caen.png")
imageio.imwrite("coin_bas_gauche.png", monimage[-200:,:300])
```



FIGURE 3 – Découpage du coin en bas à gauche

Question 4. Faites afficher un carré de 200 pixels de largeur et 200 pixels de hauteur centré au centre de l'image.

Solution:

```
import imageio
import matplotlib.pyplot as plt
import numpy as np

monimage = imageio.imread("mont_saint_michel.png")
N,M,_ = np.shape(monimage)
plt.imshow(monimage[N//2-100:N//2+100,M//2-100:M//2+100])
plt.axis('off')
plt.show()
```

Question 5. Ecrivez une fonction `rotation(I)` qui prend en argument un tableau de couleurs ($N \times M$ ou $N \times M \times 3$ ou $N \times M \times 4$) et qui renvoie une matrice de couleur correspondant à l'image tournée d'un quart de tour dans le sens horaire.

Solution:



FIGURE 4 – Découpage du centre

```
import imageio
import numpy as np
monimage = imageio.imread("tux.png")[:, :, :3]
N, M, _ = np.shape(monimage)
monimage_r = np.zeros((M, N, 3), dtype=np.uint8) # image retournée
for i in range(M):
    for j in range(N):
        monimage_r[i, j] = monimage[N-1-j, i]
imageio.imwrite("tux_r.png", monimage_r)
```



FIGURE 5 – Rotation d'un quart de tour

3 Travail sur les couleurs

Question 6. Faites afficher votre image sans sa composante rouge. Et ensuite avec seulement la composante bleue. Et ensuite faites en sorte que le rouge devienne vert, le vert bleu et le bleu rouge.

Question 7. Faites afficher le négatif de votre image.

Par exemple, si un pixel a pour composantes RGB (232, 12, 120), alors son pixel correspondant dans le négatif a pour composantes RGB $(255 - 232, 255 - 12, 255 - 120) = (23, 243, 135)$.

Solution:

```

import imageio
import numpy as np
import matplotlib.pyplot as plt

monimage = imageio.imread("mont_saint_michel.png")
N,M,_ = np.shape(monimage)

monimage1 = np.copy(monimage)
for i in range(N):
    for j in range(M):
        monimage1[i,j,0]=0
plt.axis('off')
plt.imshow(monimage1)
plt.show()
imageio.imwrite("mont_saint_michel_sans_rouge.png",monimage1)

monimage2 = np.copy(monimage)
for i in range(N):
    for j in range(M):
        monimage2[i,j,0]=0
        monimage2[i,j,1]=0
plt.axis('off')
plt.imshow(monimage2)
plt.show()
imageio.imwrite("mont_saint_michel_seulement_bleu.png",monimage2)

monimage3 = np.copy(monimage)
for i in range(N):
    for j in range(M):
        monimage3[i,j]=[monimage3[i,j,2],monimage3[i,j,0],monimage3[i,j,1]]
plt.axis('off')
plt.imshow(monimage3)
plt.show()
imageio.imwrite("mont_saint_michel_permutation_couleurs.png",monimage3)

monimage4 = np.copy(monimage[:,:,:3])
for i in range(N):
    for j in range(M):
        monimage4[i,j]=255-monimage4[i,j]
plt.axis('off')
plt.imshow(monimage4)
plt.show()
imageio.imwrite("mont_saint_michel_negatif.png",monimage4)

```

Question 8. Ecrire une fonction `luminance` prenant en argument une liste (ou un tableau `numpy`) $[R, G, B]$ et renvoyant la luminance calculée grâce à la formule donnée dans le cours.

Solution:

```

def luminance(L):
    R,G,B=L
    return R*299/1000 + G*587/1000+B*114/1000

```

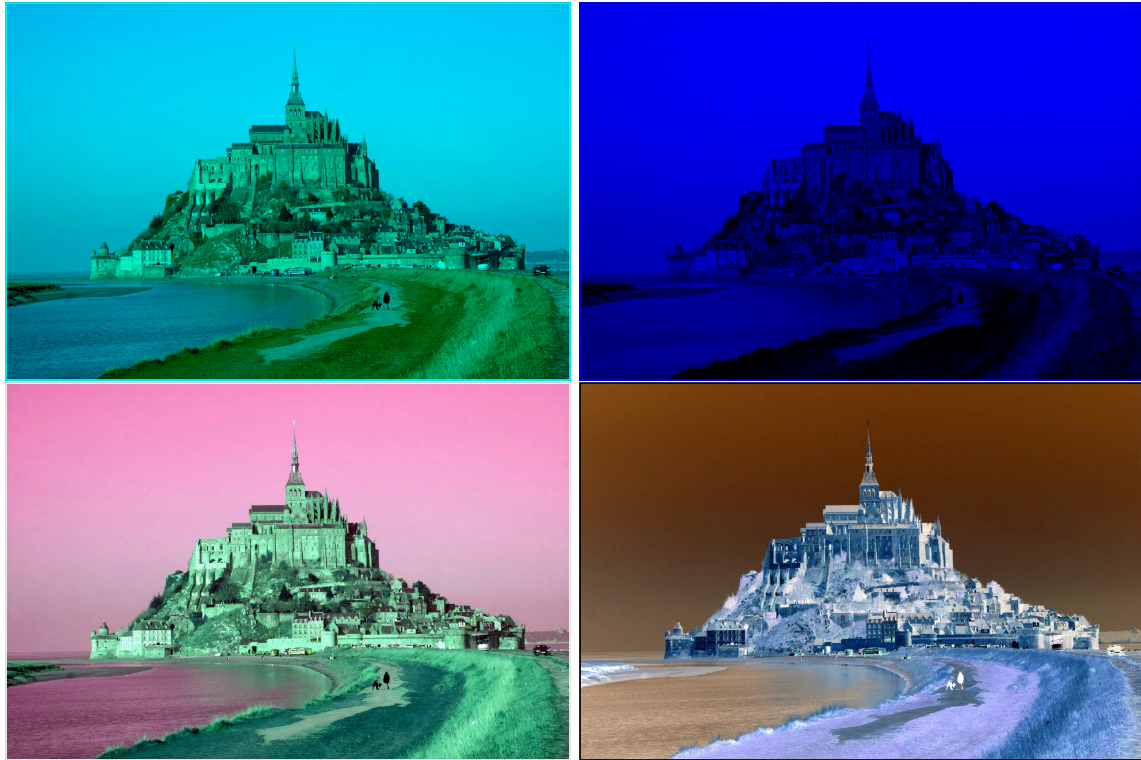


FIGURE 6 – Travail sur les couleurs

```
if __name__=="__main__":
    print(luminance([34,200,198]))
```

Question 9. Faire afficher la version en niveau de gris de l'image que vous avez choisie.

Solution:

```
import matplotlib.pyplot as plt
import numpy as np
import imageio
from luminance import *

def niveaugris(I0):
    I=np.copy(I0)
    N,M,_ = np.shape(I)
    return np.array([ [ luminance(I[i,j]) for j in range(M) ] for i in range(N)
    ])

if __name__=="__main__":
    monimage = np.copy(imageio.imread('tux.png')[:, :, :3])
    monimage_niveaugris=np.uint8(niveaugris(monimage))
    plt.imshow(monimage_niveaugris,cmap='gray')
    plt.axis('off')
    plt.show()
```

```
imageio.imsave('tux_ngris.png',monimage_niveaugris)
```

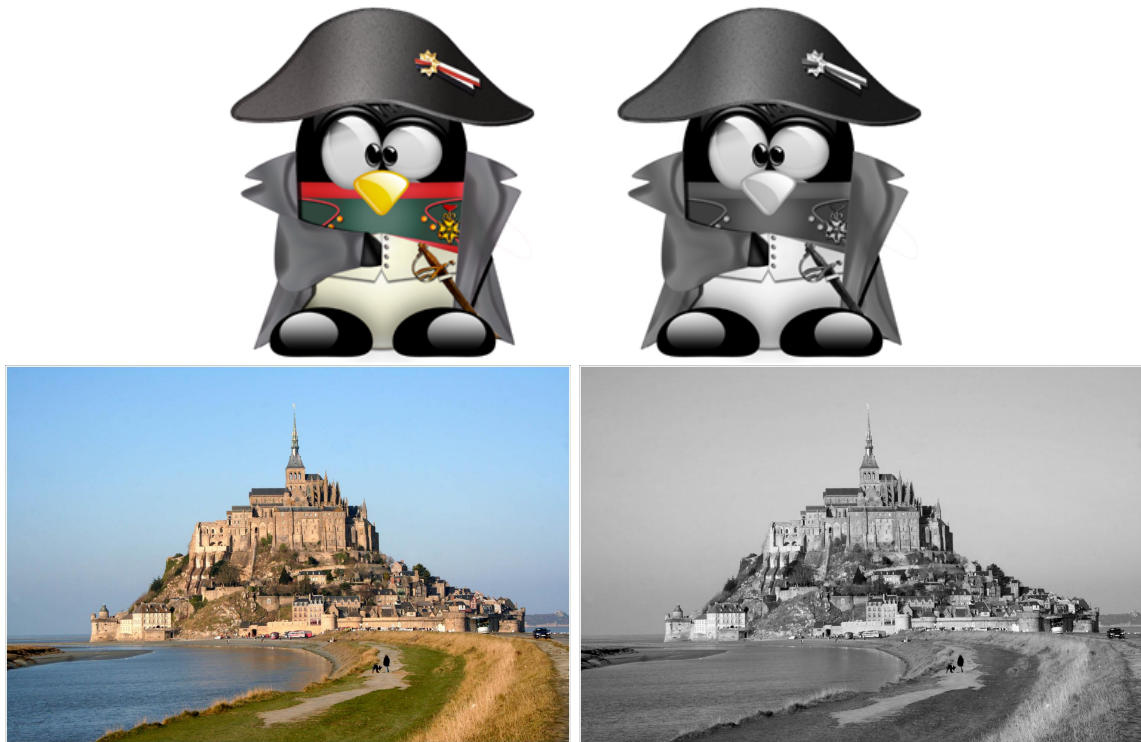
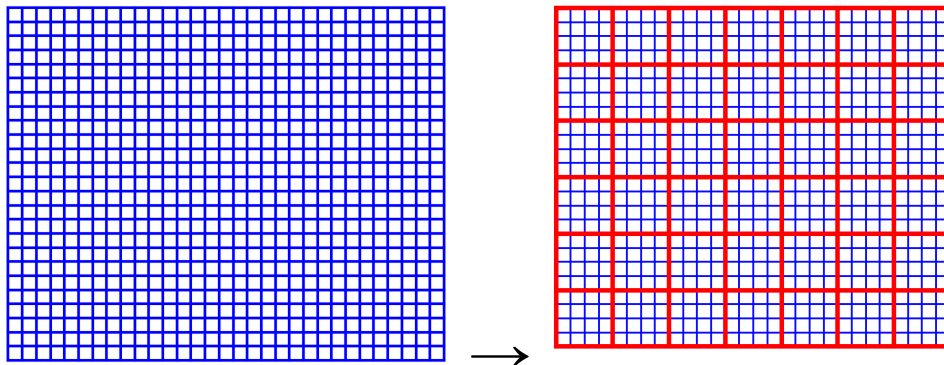


FIGURE 7 – Conversion en niveau de gris

4 Pixellisation

Pour réduire la résolution d'une image trop volumineuse, on peut la dégrader en réduisant sa résolution. Pour cela, on la pixellise : l'opération consiste à regrouper des pixels voisins et à les remplacer par un seul pixel moyen.

On regroupe ainsi des carrés de p lignes par p colonnes. La moyenne est donc réalisée sur p^2 pixels. Par exemple sur le dessin illustratif ci-dessous, $p = 4$.



Question 10. * Ecrire une fonction `pixellisation(image,p)` prenant en argument un tableau **numpy** $N \times M \times 3$ ou $N \times M \times 4$ et renvoyant un tableau **numpy** $N \times M \times 3$ correspondant à l'image pixellisée par carrés de p pixels par p pixels.

On pourra utiliser la fonction `mean` du module **numpy** pour réaliser des moyennes.

Solution:

```
import imageio
import matplotlib.pyplot as plt
import numpy as np

def pixellisation(image,p):
    N,M,_ = np.shape(image)
    N2,M2 = N//p,M//p
    pixel = np.zeros((N2,M2,3),dtype=np.uint8)

    for i in range(N2):
        for j in range(M2):
            pixel[i,j] = [ np.mean( image[i*p:(i+1)*p,j*p:(j+1)*p,k] ) for k in
range(3) ]
    return pixel

monimage = imageio.imread("jacquard.png")[:, :, :3]

monimage10=pixellisation(monimage,10)
plt.axis('off')
plt.imshow(monimage10)
plt.show()

monimage20=pixellisation(monimage,20)
plt.axis('off')
plt.imshow(monimage20)
plt.show()

monimage50=pixellisation(monimage,50)
plt.axis('off')
plt.imshow(monimage50)
plt.show()

monimage100=pixellisation(monimage,100)
plt.axis('off')
plt.imshow(monimage100)
plt.show()
```

5 Masques

Plusieurs transformations d'images intéressantes se font par application d'un masque, c'est-à-dire par produit de convolution. L'idée est de remplacer chaque pixel de l'image par une combinaison linéaire de ce pixel et de ses voisins.

5.1 Lissage

On parle aussi de floutage, ou de filtre moyennneur.

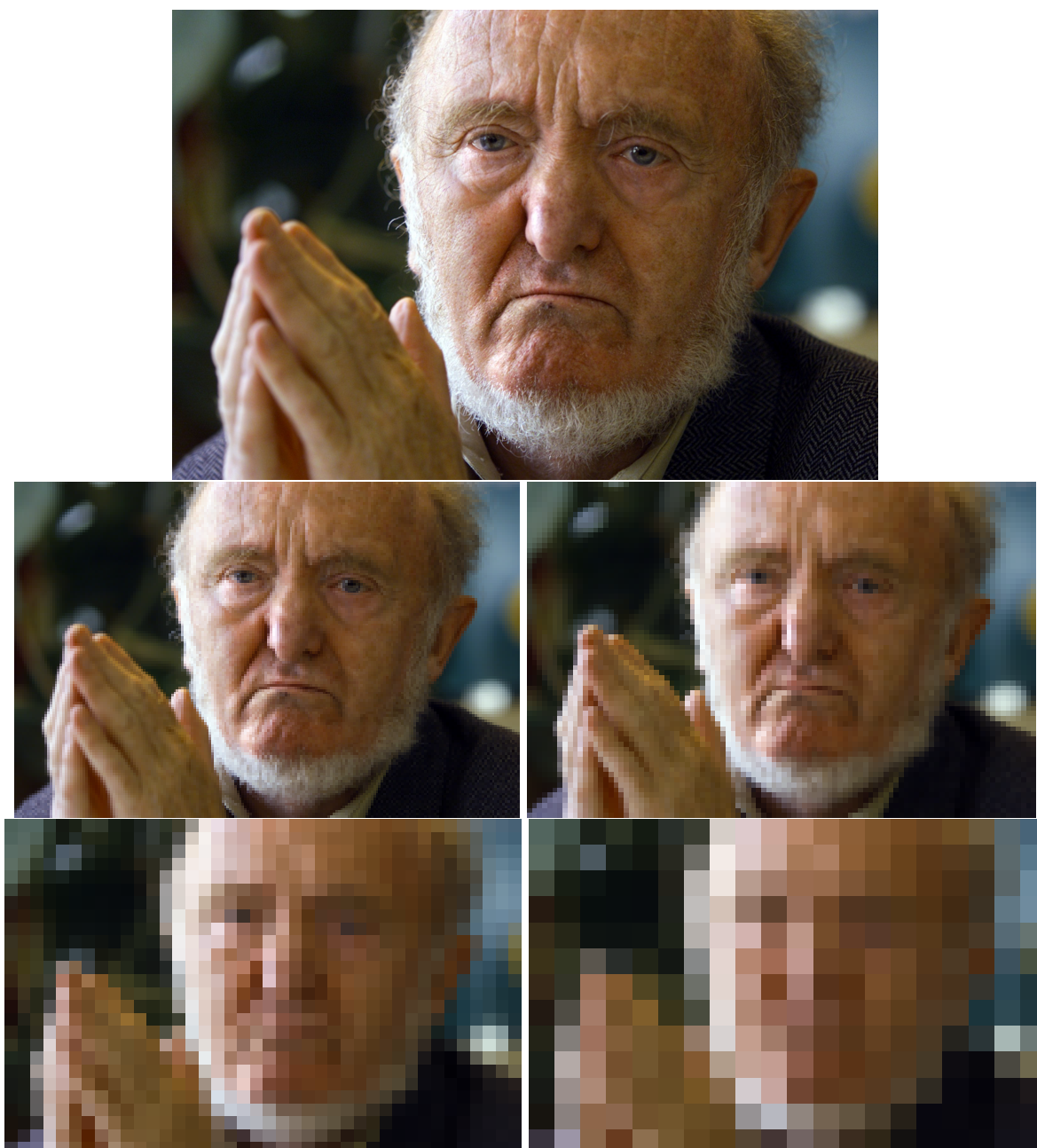


FIGURE 8 – Pixellisation ($p = 10, 20, 50, 100$)

Question 11. ★ Remplacez chaque pixel par la moyenne de lui-même et de ses 8 voisins (attention aux bords!). Là encore, on pourra utiliser la fonction `mean` du module `numpy`.

Remarque. On a en fait appliqué le masque $\frac{1}{9} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$ à l'image. Voyez-vous ce que ça signifie ?

Question 12. ★ Tentez d'appliquer plusieurs fois le processus, afin d'accentuer l'effet de floutage.

Solution:

```
import numpy as np
import imageio
import matplotlib.pyplot as plt

def lissage(I): # I est une matrice du type MxNx3 ou MxNx4 (image couleur)
    N,M,_ = np.shape(I)

    image_lissee = np.zeros( (N,M,3),dtype=np.uint8 )

    image_lissee[0,:] = I[0,:,:,3] # première ligne
    image_lissee[-1,:] = I[-1,:,:,3] # dernière ligne
    image_lissee[:,0] = I[:,0,:,:,3] # première colonne
    image_lissee[:, -1:] = I[:, -1,:,:,3] # dernière colonne

    for i in range(1,N-1):
        for j in range(1,M-1):
            image_lissee[i,j] = [ np.mean ( I[i-1:i+2,j-1:j+2,k] ) for k in
range(3) ]

    return image_lissee

image = imageio.imread("tux.png")

for i in range(42):
    image = lissage(image)
    if i%10==1:
        print("sauvegarde après",i,"lissages")
        imageio.imwrite("tux_lisse_"+str(i)+".png",image)
```

5.2 Détection de contours

On part de l'idée que les contours dans une image correspondent à de brusques changement de luminosité.

Pour étudier les changements de luminance, on pourrait remplacer chaque pixel $I_{i,j}$ par

$$(I_{i+1,j} - I_{i,j})^2 + (I_{i,j+1} - I_{i,j})^2.$$

Ceci est l'analogie discret de la norme du vecteur gradient :

$$\left(\frac{\partial f}{\partial x}(x,y) \right)^2 + \left(\frac{\partial f}{\partial y}(x,y) \right)^2.$$

En effet,

$$f(x+h,y) = f(x,y) + h \frac{\partial f}{\partial x}(x,y) + o(h) \quad \text{donc} \quad \frac{\partial f}{\partial x}(x,y) = \frac{f(x+h,y) - f(x,y)}{h} + o_{h \rightarrow 0}(1).$$

Il y a moyen d'avoir un meilleur terme de reste. En prenant les développements limités à l'ordre 2 :

$$f(x+h,y) = f(x,y) + h \frac{\partial f}{\partial x}(x,y) + \frac{h^2}{2} \frac{\partial^2 f}{\partial x^2}(x,y) + o_{h \rightarrow 0}(h^2)$$

$$f(x-h,y) = f(x,y) - h \frac{\partial f}{\partial x}(x,y) + \frac{h^2}{2} \frac{\partial^2 f}{\partial x^2}(x,y) + o_{h \rightarrow 0}(h^2)$$



FIGURE 9 – Image originale, puis après 1, 11, 21, 31 et 41 lissages

En soustrayant, on obtient :

$$f(x+h, y) - f(x-h, y) + 2h \frac{\partial f}{\partial x}(x, y) + o_{h \rightarrow 0}(h^2) \quad \text{donc} \quad \frac{\partial f}{\partial x}(x, y) = \frac{f(x+h, y) - f(x-h, y)}{2h} + o_{h \rightarrow 0}(h).$$

Le terme de reste est donc bien meilleur ! Et c'est analogue pour la dérivée suivant y .

(En fait, on a déjà expliqué tout ça en MPSI avec le calcul numérique d'une dérivée !)

Revenons au monde discret... Pour détecter les contours, ce serait donc mieux de remplacer chaque pixel $I_{i,j}$ par

$$(I_{i+1,j} - I_{i-1,j})^2 + (I_{i,j+1} - I_{i,j-1})^2,$$

c'est-à-dire qu'on applique séparément les masques $\begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$ et $\begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$ et qu'on calcule

la norme 2 du résultat.

En fait, il est encore mieux de faire une moyenne autour du point et de considérer les masques $\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$ et $\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$ (filtre de Sobel), et c'est ce que je vous propose de faire.

Question 13. ★★ Ecrire une fonction `gradient(L)` qui prend en argument une matrice de la forme $N \times M$ contenant les luminances des pixels, et renvoyant la matrice de flottants des gradients discrets (attention aux bords !).

On fait l'hypothèse que les points du contour correspondent à un gradient élevé (en pratique, supérieur à un certain seuil).

Question 14. ★★ Ecrire une fonction `contours(G, seuil)` prenant en argument un tableau **numpy** de gradients de format $N \times M$, et renvoyant une matrice de couleurs de format $N \times M$ (niveau gris), un pixel valant 0 (noir) s'il est sur un contour, et tous les autres pixels valant 255 (blanc). Cette matrice devra être du type tableau **numpy** d'entiers sur 8 bits.

Solution:

```
import numpy as np
import imageio
import matplotlib.pyplot as plt

from luminance import *
from niveau_gris import *

def gradient(L): # L est du type N x M
    N,M = np.shape(L)
    G = np.zeros((N,M))
    G[0,:] = 0 # premières lignes
    G[-1,:] = 0 # dernières lignes
    G[:,0] = 0 # premières colonnes
    G[:, -1:] = 0 # dernières colonnes

    for i in range(1,N-1):
        for j in range(1,M-1):
```

```

        tmp_x = L[i+1,j-1]-L[i+1,j+1] +\
                2 * (L[i,j-1]-L[i,j+1]) +\
                L[i-1,j-1] - L[i-1,j+1]
        tmp_y = L[i+1,j-1]-L[i-1,j-1] +\
                2 * (L[i+1,j]-L[i-1,j]) +\
                L[i+1,j+1] - L[i-1,j+1]
        tmp = tmp_x**2+tmp_y**2
        G[i,j]=tmp
    return G

def contours(G,seuil):
    N,M=np.shape(G)
    C = 255*np.ones((N,M),dtype=np.uint8)
    for i in range(N):
        for j in range(M):
            if G[i,j]>seuil: C[i,j]=0
    return C

image = imageio.imread("caen.png")[:, :, :3]
L=niveaugris(image)
G=gradient(L)
C=contours(G,4000)

# tux 10000
# bambi, caen, mont_saint_michel 4000
# jacquard 1000

plt.imshow(C,cmap='gray')
plt.axis('off')
plt.show()

imageio.imsave("caen_contours.png",C)

```

Remarque. Il existe des variantes à cette méthode du gradient discret. On peut aussi considérer des laplaciens discrets. En fait, on pourrait consacrer un TD entier (ou un TIPE !) à la détection de contours.

6 Profondeur et stéganographie

6.1 Dégradation par perte de bits d'information

Question 15. Comment obtenir très simplement l'écriture décimale des k bits de poids faible de l'entier x représenté par un octet ? et des ℓ bits de poids fort ?

Solution: Il suffit de considérer respectivement le reste dans la division euclidienne de x par 2^k et le quotient dans la division euclidienne de x par $2^{8-\ell}$.

Pour les plus avancés d'entre vous, on peut bien sûr, faire des opérations directement sur les bits...

```

x = 187
print(x, "\t", bin(x))
y = x >> (8-5) # 5 bits de poids fort
print(y, "\t", bin(y))
z = (x >> 2) << 2 # 2 bits de poids faible mis à zéro

```

```
print(z,"\t",bin(z))
t = x-z # 2 bits de poids faible
print(t,"\t",bin(t))
```

```
187  0b10111011
23   0b10111
184  0b10111000
3    0b11
```

Question 16. ★ Tentez de retirer 1, 2, 3, etc. bits de poids faibles à chaque couleur d’une image. A partir de quand voit-on la différence à l’œil nu ?

Solution:

```
import matplotlib.pyplot as plt
import imageio
import numpy as np

image = imageio.imread("caen.png")[:, :, :3]

for n in range(8): # nombre de bits pour chaque couleur
    image_degradee = np.uint8( (2**(8-n) * (image // 2**(8-n))))
    plt.axis('off')
    plt.imshow(image_degradee)
    plt.show()
    imageio.imwrite("caen_"+str(n)+"bits.png", image_degradee)
```

6.2 Une image cachée dans une autre

Alice veut faire parvenir un message secret à Bob, sans qu’Oscar puisse le lire. Elle peut pour cela utiliser un procédé cryptographique connu de tout le monde, dont la sécurité repose sur une clé secrète.

Avec l’approche stéganographique, le secret réside dans le procédé lui-même. Ainsi, si la cryptographie est l’art du **secret**, la stéganographie est l’art de la **dissimulation**. Autrement dit, on va dissimuler des données dans d’autres données. Ici, dans des images, de façon imperceptible pour l’œil.

J’ai caché une image derrière l’image **mystere1.png** déposée sur la Dropbox.

Le principe que j’ai utilisé est le suivant : j’ai remplacé les 4 bits de poids faibles (pas importants) de l’image masquante, par les 4 bits de poids fort (suffisants pour percevoir l’image) du pixel correspondant dans l’image masquée. Autrement dit, on consent à perdre un peu d’information sur l’image masquante, pour cacher de l’information importante permettant de reconstituer une autre image.

Par exemple, si la composante verte du pixel ligne 345, colonne 1021 du fichier **chapeau.png** est

$$10100100_2 = 164_{10}$$

et que la composante verte du pixel ligne 345, colonne 1021 du fichier **lapin.png** est

$$01111101_2 = 125_{10}$$

alors si on veut cacher un lapin dans le chapeau, la composante verte du pixel ligne 345, colonne 1021 du fichier `magie.png` est

$$\mathbf{10100111}_2 = \mathbf{167}_{10}.$$

La personne qui connaît le procédé pourra alors affirmer que la composante verte du pixel ligne 345, colonne 1021 du fichier `lapin.png` caché était à peu près :

$$\mathbf{01110000}_2 = \mathbf{112}_{10}$$

Et on fait pareil pour chaque couleur et pour chaque pixel (le plus simple est de prendre le chapeau et le lapin de même taille).

On est donc bien parvenu à cacher une image à l'intérieur d'une autre !

Question 17. Retrouvez l'image cachée derrière `mystere1.png`.

Solution:

```
import numpy as np
import matplotlib.pyplot as plt
import imageio

mystere = imageio.imread("mystere.png")

imagecachee = 16*(mystere%16)
plt.axis('off')
plt.imshow(imagecachee)
plt.show()
```

Dans la jungle était caché un tigre.

6.3 Un texte caché dans une image

Mais au fait... dans les TD de cryptographie... on a vu qu'on pouvait encoder les textes avec chaque caractère stocké sur 8 bits grâce au code ASCII...

On peut donc cacher un texte dans une image. En pratique, ceci peut être utilisé pour tatouer numériquement une image (watermark) pour des copyrights.

J'ai caché un texte dans la première ligne de l'image `mystere2.png` déposée sur la Dropbox.

Ce texte est à lire de gauche et à droite (comme en langue naturelle). Chaque caractère est caché pour moitié dans la composante rouge d'un pixel, et pour moitié dans la composante verte, avec le procédé suivant.

Par exemple, si on veut cacher un `a` de code ASCII

$$\mathbf{97}_{10} = \mathbf{01100001}_2$$

dans le pixel RGB

$$(\mathbf{154}_{10}, \mathbf{76}_{10}, \mathbf{200}_{10}) = (\mathbf{10011010}_2, \mathbf{01001100}_2, \mathbf{11001000}_2)$$

alors on enverra

$$(\textcolor{red}{1001}\textcolor{red}{0110}_2, \textcolor{green}{0100}\textcolor{blue}{0001}_2, 11001000_2) = (150_{10}, 65_{10}, 200_{10})$$

- Les 4 premiers bits du code ASCII du caractère sont les 4 derniers bits de la composante rouge.
- Les 4 derniers bits du code ASCII du caractère sont les 4 derniers bits de la composante verte.

Question 18. Retrouvez le texte caché dans `mystere2.png`.

Solution:

```
import imageio
import numpy as np

mystere2 = imageio.imread("mystere2.png")
print(np.shape(mystere2))
N,M,_ = np.shape(mystere2)

texte1 = ""
for j in range(M):
    texte1 += chr(16*(mystere2[0,j,0]%16)+(mystere2[0,j,1]%16))

print(texte1[:1270])
```

```
(1400, 1400, 3)
Yellow Submarine
```

```
In the town where I was born
Lived a man who sailed to sea
And he told us of his life
In the land of submarines
```

```
So we sailed up to the sun
Till we found the sea of green
And we lived beneath the waves
In our yellow submarine
```

```
We all live in a yellow submarine
Yellow submarine, yellow submarine
We all live in a yellow submarine
Yellow submarine, yellow submarine
```

```
And our friends are all on board
Many more of them live next door
And the band begins to play
```

```
We all live in a yellow submarine
Yellow submarine, yellow submarine
We all live in a yellow submarine
Yellow submarine, yellow submarine
```

```
[Full speed ahead, Mr. Parker, full speed ahead!]
```

Full speed over here, sir!
Action station! Action station!
Aye, aye, sir, fire!
Heaven! Heaven!]

As we live a life of ease (A life of ease)
Everyone of us (Everyone of us) has all we need (Has all we
need)
Sky of blue (Sky of blue) and sea of green (Sea of green)
In our yellow (In our yellow) submarine (Submarine, ha, ha)

We all live in a yellow submarine
Yellow submarine, yellow submarine
We all live in a yellow submarine
Yellow submarine, yellow submarine
We all live in a yellow submarine
Yellow submarine, yellow submarine
We all live in a yellow submarine
Yellow submarine, yellow submarine

Derrière l'image des Beatles était cachée une chanson des Beatles.
Toute la première ligne n'a pas été utilisée... d'où les caractères insignifiants à la fin, coupés au montage.

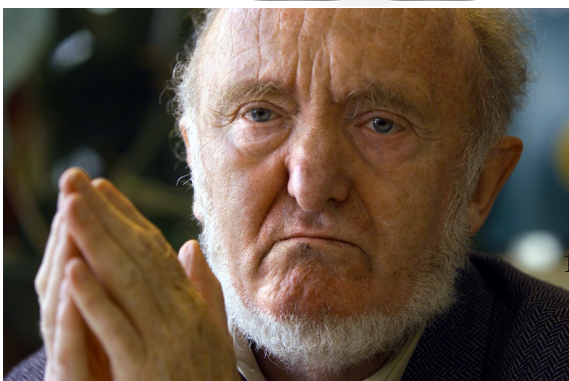
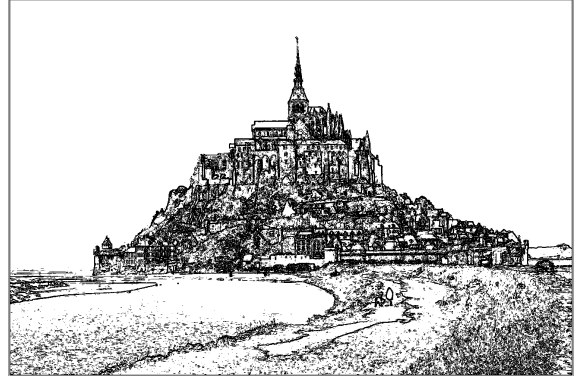


FIGURE 10 – Détection de contours

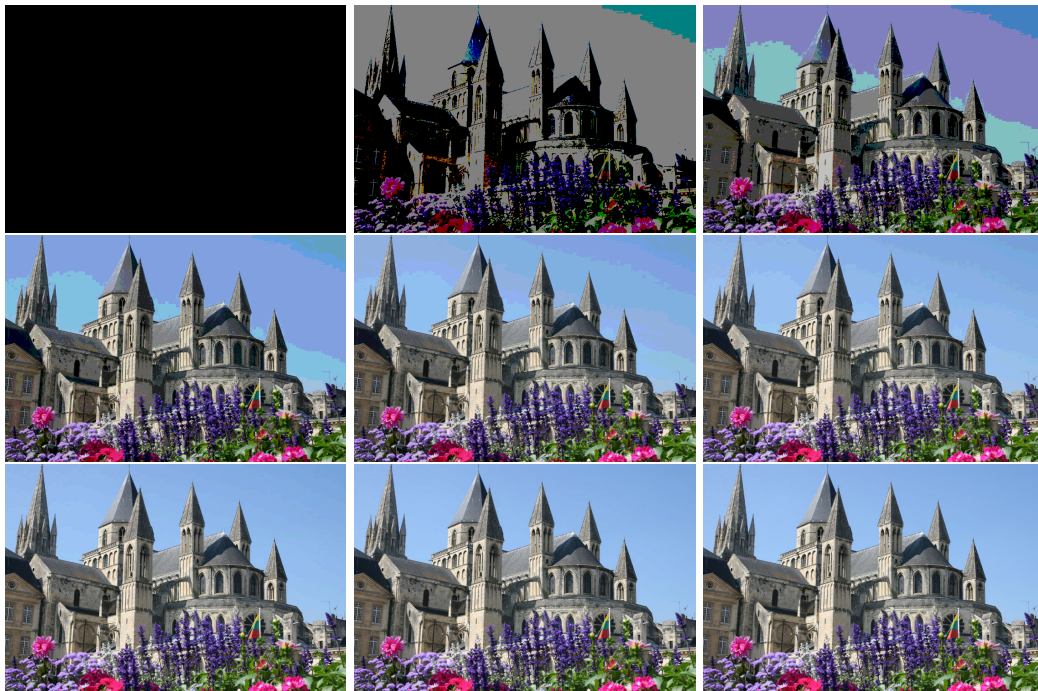


FIGURE 11 – Perte de bits

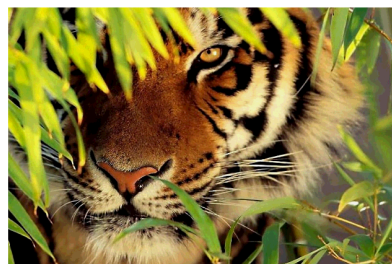


FIGURE 12 – Image cachée par stéganographie



FIGURE 13 – Image mystère 2