Devoir libre d'Informatique 2

- Le DL sera rendu sous forme informatique (programme Python), par mail, à M. Demarais. Le fichier est à envoyer au plus tard le dimanche 2 janvier à minuit.
- LE NOM DU FICHIER AURA LE FORMAT SUIVANT : nom_prenom_DL2.py.
- Inutile de recopier les uns sur les autres, ce DL ne sera pas noté ! Le but, c'est de s'entrainer ! Si vraiment vous avez une passion pour le recopiage, alors vous pourrez recopier le corrigé qui vous sera fourni ultérieurement... !! Au moins, il n'y aura pas d'erreurs !
- Commenter régulièrement votre programme afin d'expliquer ce que fait ce dernier. Un programme est fait pour être relu, par d'autres personnes. Les commentaires sont donc très utiles.
- Utiliser les noms de variables imposés. Si vous introduisez d'autres variables, utiliser des noms explicites, en lien avec le problème posé.
- Pour plus de facilité, utiliser une console IPython (beaucoup d'images à visualiser).
- A chaque fois que vous affichez une image, pensez à mettre un titre. L'objectif principal de ce DL est de connaitre le principe de codage des images et de les manipuler. Dans la partie II), on modifiera l'apparence d'une image en agissant sur ses couleurs. Dans la partie III), on déterminera une méthode pour cacher une image dans une autre image... Et on pourra chercher à visualiser une image cachée dans une autre image par la méthode étudiée précédemment... Les parties II) et III) sont indépendantes. Pour les parties I), II) et III) 2), on travaillera uniquement avec l'image « masque.png ».

1 Préambule

1.1 Représentation informatique des images

Les images avec l'extension png « portable network graphics » sont enregistrées sous forme matricielle avec différents formats. Le format retenu pour ce DL est uint8 qui code sur 8 bits chaque couleur. Chaque image est stockée sous forme d'un tableau à 3 composantes :

```
array([[[254, 224, 190, 255],
        [253, 211, 174, 255],
        [253, 201, 160, 255],
        ...,
        [255, 238, 174, 255],
        [255, 241, 166, 255],
        [255, 253, 176, 255]],

       ...,

       [[253, 234, 180, 255],
        [253, 225, 151, 255],
        [255, 213, 110, 255],
        ...,
        [254, 248, 194, 255],
        [219, 192, 141, 255],
        [231, 193, 144, 255]]], dtype=uint8)
```

- Les deux premières composantes du tableau sont les coordonnées x et y des pixels.

- La troisième composante contient un tableau de 4 valeurs (chacune de ces valeurs étant codée sur un octet = 8 bits) :

1. les niveaux des trois couleurs RGB (Red, Green, Blue),
2. ainsi que la transparence (pas toujours présente et non utilisée dans ce DL, toujours de valeur 255 ici).

Pour chaque pixel, chaque couleur est donc représentée par un entier compris entre 0 et 255 (codage sur 8 bits).

Exemple :

	R	G	B	Couleur obtenue
[0, 0, 0, 255]	0	0	0	Noir
[255, 255, 255, 255]	255	255	255	Blanc
[255, 0, 0, 255]	255	0	0	Rouge
[0, 255, 0, 255]	0	255	0	Vert
[0, 0, 255, 255]	0	0	255	Bleu

1.2 Ouverture et lecture d'une image

On utilise dans ce DL les bibliothèques matplotlib.pyplot, numpy et matplotlib.image (pour l'ouverture et l'écriture d'images).

- En tout premier lieu, créer un dossier « DL2 » dans lequel vous mettrez tous les fichiers nécessaires à ce DL (notamment les images fournies et votre script).
- Importer comme suit les bibliothèques :

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

- On propose d'utiliser la fonction lecture_image() suivante :

```
def lecture_image(nom_fichier):
    """ lit une image stockée au format.png dans le fichier\
        nom_fichier et retourne l'image codée au format uint8 """
    img = mpimg.imread(nom_fichier)  # lecture de l'image
    img = (img * 255).round().astype(np.uint8)
    # reformatage de l'image : multiplication par 255, arrondi, passage au type
    # uint8
    return img
```

Copier cette fonction au début de votre script de manière à pouvoir l'utiliser par la suite.

1.3 Affichage d'une image

- Tester ce morceau de programme :

```
img = lecture_image('masque.png')    # lecture de l'image
plt.figure()                         # crée une fenêtre de tracé vide
plt.title('image masque')            # permet de mettre un titre
plt.imshow(img)                      # affichage de l'image
plt.show()                           # affichage de la fenêtre
```

La variable img contient alors l'image sous forme matricielle.

- -Dans la console interactive, taper l'instruction : img et retrouver les éléments décrits ci-dessus.
- Pour avoir accès aux couleurs du pixel (20, 40), tapez dans l'éditeur :

```
print(img[20, 40, :])              # accès au pixel (20, 40)
print(img[20, 40, 1])              # accès à la couleur verte du pixel (20, 40)
```

1.4 Taille d'une image

La méthode shape donne les dimensions de l'image en pixels. La profondeur est le nombre d'éléments par pixel (3 couleurs et une transparence) : (largeur, hauteur, profondeur) = img.shape

Tester cette instruction dans l'éditeur, en tapant :

```
print(img.shape)
```

Donner les dimensions de l'image en commentaire dans votre programme.

1.5 Sauvegarde d'une image

L'instruction imsave permet de sauvegarder une image dans un fichier. Tester l'instruction suivante dans l'éditeur :

```
mpimg.imsave('sauvegarde_masque.png', img)
```

Vérifier que le fichier « sauvegarde_masque.png » a bien été créé dans le dossier « DL2 ».

2 Modification de l'apparence d'une image en agissant sur ses couleurs

1) Image rouge : Ecrire une fonction rouge ayant pour argument la forme matricielle de l'image img et qui renvoie la forme matricielle de l'image monochrome rouge correspondante.

Indication : pour obtenir une image monochrome rouge : on annule les composantes bleues et vertes de tous les pixels.

Tester votre fonction en affichant votre image rouge. Vous pouvez alors sauvegarder votre image rouge sous le nom « rouge.png ».

IMPORTANT : Vérifier, à l'issue de cette question, que la forme matricielle de l'image de base img n'a pas été transformée. Si toutefois c'était le cas, modifier votre fonction de manière à créer une copie indépendante de la forme matricielle de l'image utilisée img au début des instructions dans la fonction. On rappelle que, pour réaliser une copie d'un tableau M, l'opération M_bis = M ne suffit pas. On propose d'utiliser ici l'instruction suivante pour réaliser une copie indépendante de la forme matricielle d'une image : img_bis = np.copy(img)
Bien penser à toujours réaliser des copies de vos images au début des instructions pour chaque fonction dans l'ensemble du DL.

2) Image monochrome : Ecrire une fonction monochrome ayant pour arguments la forme matricielle de l'image img et une chaîne de caractère c (qui peut prendre les valeurs 'R', 'G', 'B' ou 'grey'), et qui renvoie une copie de la forme matricielle de l'image monochrome de la couleur demandée. Quand c prend la valeur 'grey', on retournera l'image en niveaux de gris obtenue comme expliqué ci-dessous.

Une image grise est une image pour laquelle chaque pixel a les mêmes valeurs de couleurs (R, G et B). Par exemple, [100, 100, 100, 255] est un pixel gris. Pour convertir une image couleur en niveaux de gris, il faut que pour chaque pixel, chaque couleur prenne la valeur moyenne des valeurs initiales.

Exemple : [100, 120, 20, 255] devient [80, 80, 80, 255] Tester votre fonction en affichant différentes images monochromes.

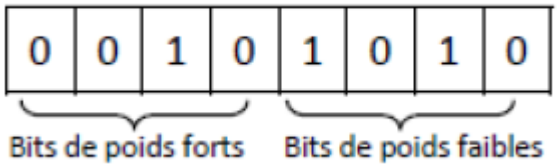
3 Camouflage d'une image

Dans la partie 3), on utilisera deux images ayant la même dimension (même nombre de pixels). On cachera une première image appelée « secret.png » (destinée à un seul interlocuteur) dans une autre image servant de masque appelée « masque.png ». On camouflera l'image « secret.png » sur les bits de poids faibles de l'image « masque.png » (voir 1) Principe du camouflage). Dans la partie 4), on cherchera à afficher une photo cachée dans une autre photo... (dé-camouflage).

3.1 Principe du camouflage

Nous avons vu précédemment que chaque couleur est représentée par un entier compris entre 0 et 255, ce qui permet d'obtenir plus de 16 millions de couleurs différentes. L'idée de départ est que pour chacun de ces entiers, un codage entre 0 et 255 n'est peut-être pas nécessaire. En effet, pour chaque entier codé sur 8 bits, les 4 bits de poids forts donnent quasiment toute l'information, les autres servant à apporter des nuances.

Exemple :



On va alors tronquer l'information de chacune de ces valeurs et ne garder que l'information principale de chaque image. L'information principale de l'image à cacher sera alors dissimulée sur les bits de poids faibles de l'image masque.

3.2 Travail préliminaire

Afficher la valeur de la couleur d'un pixel en décimal puis en binaire. Pour la valeur rouge du pixel (60, 40) de l'image img, tapez dans l'éditeur :

```
print(img[60, 40, 0])
print(bin(img[60, 40, 0]))
```

Vérifier que le codage de la couleur se fait bien sur 8 bits. Pour la valeur rouge du pixel (20, 40) de l'image img, tapez dans l'éditeur :

```
print(img[20, 40, 0])
print(bin(img[20, 40, 0]))
```

Remarque : si le code binaire obtenu débute par des zéros, les zéros situés à gauche du premier 1 sont automatiquement tronqués, ce qui explique dans certains cas la visualisation sur moins de 8 bits.

Nous pourrons utiliser dans ce devoir les opérateurs booléens suivants :

Opération	Description
x & y	Opération « et » sur les bits de x et y
x << y	Décalage à gauche de y bits sur x
x >> y	Décalage à droite de y bits sur x

3.2.1 Opération « et » sur les bits de x et y (x & y) :

Cette instruction effectue un « et logique » sur chacun des bits pris un à un.

Exemple :

ET LOGIQUE « & »			ET « bit à bit »	
0&0	0		1010 & 0111	0010
0&1	0		1111 & 0110	0110
1&0	0		1111 & 0001	0001
1&1	1		0000 & 0110	0000

- Tester dans l'éditeur les instructions suivantes (img contient une image sous forme matricielle) :

```
a = img[60, 40, 0]
print('a = ', a)
print('bin(a) = ', bin(a))
b = a & 0b11110000
print('a & 0b11110000 = ', b)
print('bin(a & 0b11110000) = ', bin(b))
c = a & 0b00001111
print('a & 0b00001111) = ', c)
print('bin(a & 0b00001111) = ', bin(c))
```

- -Copier le résultat de ces tests en commentaire dans votre programme. Indiquer ce que font les instructions « a & 0b11110000 » et « a & 0b00001111 ».

3.2.2 Opération « décalage de bit » (<< ou >>)

Cette opération effectue un décalage de bits vers la gauche ou vers la droite.

- Tester dans l'éditeur les instructions suivantes (img contient une image sous forme matricielle) :

```
a = img[60, 40, 0]
print('a = ', a)
print('bin(a) = ', bin(a))
b = a >> 3
print('bin(a >> 3) = ', bin(b))
c = a << 5
print('bin(a << 5) = ', bin(c))
```

- - Copier le résultat de ces tests en commentaire dans votre programme. Indiquer ce que font les instructions « a >> 3 » et « a << 5 ». 3)

3.3 Camouflage

L'opération de camouflage consiste à placer les 4 bits de poids forts de l'image « secret.png » à la place des 4 bits de poids faibles de l'image « masque.png ». Avant de pouvoir camoufler l'image « secret.png » dans l'image « masque.png », on propose au préalable de « libérer » les 4 bits de poids faibles de l'image « masque.png ».

3) Ecrire une fonction reduit_4bits, ayant pour argument la forme matricielle de l'image à traiter img, qui renvoie une copie de la forme matricielle de cette image dont les 4 bits de poids faibles de chaque couleur de chaque pixel ont été remplacés par des 0. On utilisera l'une des opérations définies ci-dessus.

Tester la fonction avec la forme matricielle de « masque.png ».

Afficher l'image de base et l'image modifiée. Voit-on une différence ?

4) Ecrire une fonction cache_image(im_masque, im_secret) ayant pour arguments les formes matricielles de l'image masque et de l'image à cacher.

Cette fonction retourne une image sous forme matricielle, dans laquelle les 4 premiers bits sont les bits de poids forts de l'image masque et les 4 derniers bits sont les bits de poids forts de l'image secret.

Vous utiliserez votre fonction reduit_4bits, ainsi que les opérations décrites ci-dessus.

Tester votre fonction en camouflant « secret.png » dans « masque.png », et afficher l'image obtenue.

Vous pouvez sauvegarder votre image sous le nom « secret_dans_masque.png ».

L'image masque est-elle modifiée visuellement ?

Vérifier que les valeurs RGB ont pourtant bien été modifiées.

3.4 Dé-camouflage :

5) Ecrire une fonction trouver_image ayant pour argument la forme matricielle de l'image à traiter img, et qui retourne la forme matricielle de l'image cachée par le protocole précédent.

Vérifier que vous arrivez à retrouver l'image « secret » cachée précédemment, en affichant l'image correspondante.

Vous pouvez sauvegarder votre image sous le nom « extraction_secret.png ».

Cette image a-t-elle été modifiée visuellement ? Trouver l'image cachée dans l'image « matheux.png ». On la sauvegardera sous le nom « image_trouvee.png ». Commenter !