Stéganographie

La stéganographie est l'art de la dissimulation : l'objectif est de faire passer inaperçu un message dans un autre message.
Par exemple, une image est dissimulée dans la photographie ci-dessous sans qu'elle soit détectable facilement à l'œil.

1. Objectif et méthode

Dans ce TD, on va essayer de cacher une image **secret.bmp** à l'intérieur d'une autre image **public.bmp**.
Une image au format **.bmp** est stockée sous la forme d'un tableau à 3 dimensions :

- coordonnée x du pixel ;
- coordonnée y du pixel ;
- couleur (rouge, vert ou bleu).

La valeur stockée dans chaque case du tableau correspond au niveau de la couleur voulu. Elle est stockée sur 8 bits donc est un entier compris entre 0 et 255.

Pour dissimuler l'image **secret.bmp** à l'intérieur d'une autre image **public.bmp** nous allons utiliser le fait que pour chacune des couleurs, le codage entre 0 et 255 n'est peut être pas obligatoire. En effet, pour chaque entier codé sur 8 bits, les 4 bits de poids fort (les plus à gauche) donnent quasiment toute l'information, les autres servant à apporter des nuances.
Ainsi, en ne prenant pas en compte les 4 bits de poids faible, l'information est codée de 00000000 (0 en base 10) à 11110000 (240 en base 10) au lieu de l'être de 00000000 (0 en base 10) à 11111111 (255 en base 10). En réalisant cette opération, on pert seulement environ 5 % de l'information.

La pire des altérations est représentée sur la figure ci-contre où est représenté un blanc pur (R=255,G=255,B=255) à gauche, et la couleur obtenue en ne conservant que les 4 bits de poids fort à 1 (R=240, G=240, B=240) à droite.

*Dans ce TP, on va donc tronquer l'information de toutes les valeurs de couleurs de l'image **public.bmp**. Les bits de poids faibles ainsi libérés serviront à contenir les bits de poids forts des informations de couleur de l'image **secret.bmp**.*

2. Manipulation d'une image

Après s'être assuré d'être dans le bon répertoire, on peut alors charger l'image en mémoire et l'afficher. On utilisera les fonctions d'affichage et de manipulation des images en utilisant les fonctions `pyplot` de la librairie `matplotlib` :

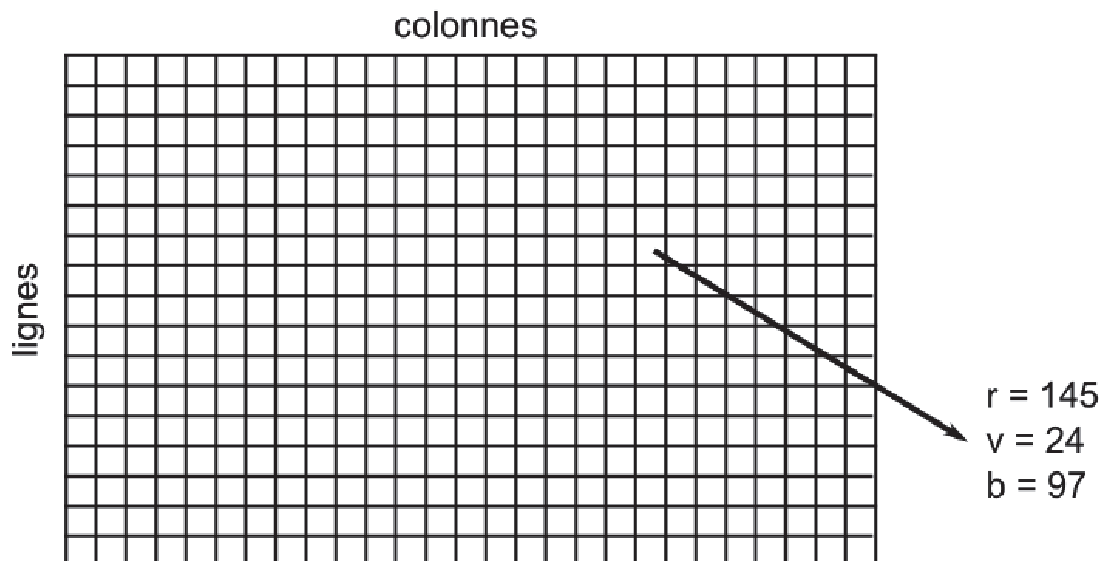
Python

```
import matplotlib.pyplot as plt

#Chargement de l'image public.bmp dans la variable image_public
image_public=plt.imread("public.bmp")

#Affichage de l'image
plt.imshow(image_public)
plt.show()
```

L'objet `image_public` est en fait un tableau à trois dimensions qui contient une information de couleur pour chaque pixel :



Pour trouver les caractéristiques de l'image, on applique la méthode `.shape` à l'objet `image_public`. Cette méthode renvoie un *nuplet* (une sorte de liste) qui contient les dimensions du tableau :

Python

```
image_public.shape
(542, 764, 3)
```

On apprend donc que l'image contient 542 lignes et 764 colonnes soit 542×764 pixels et qu'à chaque pixel (couple (ligne,colonne)) sont attachées trois valeurs : les taux de rouge, de vert et de bleu. Chaque taux est un nombre codé sur 8 bit et allant donc de 0 à 255.

Rq Comme toujours en Python, on fera attention aux numéros des lignes et des colonnes : il y a ici 542 lignes numérotées de 0 à 541.

On peut accéder aux niveaux de couleurs de chaque pixel en plaçant les coordonnées entre crochets :

```python
#Determination du niveau de rouge du pixel du coin
#superieur gauche:
image_public[0,0,0]
141

#Determination du niveau de bleu d'un pixel :
image_public[84,282,2]
244
```

Python

3. Suppression des bits de poids faible

3.1. Masque binaire

Dans un premier temps, on va passer les quatre bits de poids faible (à droite) à la valeur 0. Pour cela, on peut utiliser l'opérateur & qui permet de réaliser l'opération logique "et" bit à bit. La table de vérité de l'opérateur "et" est :

A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

```python
bin (0b1&0b1)
0b1
bin (0b10&0b11)
0b10

#Il n' est pas necessaire de travailler avec des binaires :
0b10&0b11
2
2&3
2
```

Python

1. Comment utiliser l'opérateur & pour ne conserver que les 4 bits de poids forts d'un octet (les autres prennent la valeur 0) ?

2. Créer une fonction `poids_fort` transformant un octet (entré en base 10) de sorte à passer ses 4 bits de poids faibles à 0. La fonction doit retourner un octet (en base 10).
Par exemple la fonction appliquée à 255 (qui s'écrit 11111111 en binaire) doit renvoyer 240 (qui s'écrit 11110000 en binaire).

3.2. Troncature d'une image complète

3. En utilisant la méthode `.shape` (qui renvoie un nuplet qui s'utilise comme une liste) et la fonction `poids_fort`, écrire une fonction `image_fort` permettant de ne conserver que les 4 bits de poids fort pour chaque niveau de couleur de chaque pixel.
Pour vérifier que la troncature n'altère pas trop le rendu visuel de l'image, on peut sauvegarder l'image tronquée sur le disque et visualiser côte-à-côte les images originales et tronquées :

```python
image_public_trunc=image_fort(image_public)
plt.imsave("public_trunc .png", image_public_trunc)
```

Python

4. Des bits de poids forts aux bits de poids faibles

4.1. Décalage vers la droite

L'opération `A » n` permet de décaler de n bits vers la droite les bits du nombre A.

```python
bin (0b1101>>1)
0b110

bin (0b1101>>2)
0b11

bin (0b1101>>3)
Ob1

#Encore une fois, inutile de ne travailler qu'avec des binaires :
0b1101>>2
3

13>>2
3
```

Python

4. Créer une fonction `fort_vers_faible` qui permet de décaler les 4 bits de poids fort sur les 4 bits de poids faibles d'un octet.

4.2. Décalage de l'image secrète

5. En utilisant la méthode `.shape` et la fonction `fort_vers_faible`, écrire une fonction `image_faible` permettant de décaler les 4 bits de poids fort sur les 4 bits de poids faibles pour chaque niveau de couleur de chaque pixel.

6. Appliquer cette fonction à l'image **secret.bmp** enregistrée sur votre disque. Pour cela, on l'importera en Python sous le nom `image_secret`. L'image finale obtenue sera enregistrée sur le disque sous le nom **secret_deca.png**. On pourra vérifier cette fois-ci que le décalage des bits de poids faibles altère sensiblement l'image secrète.

5. Concaténation des deux images

On dispose de deux nombres binaires 0bxxxx0000 et 0byyyy et ou souhaite obtenir le binaire 0bxxxxyyyy.

7. Quelle opération mathématique simple permet d'obtenir le résultat voulu ?

8. Créer une fonction `concatenation` qui permette d'obtenir une seule image à partir des deux images (publique et secrète) tronquées.

9. Appliquer cette fonctions aux images `public_trunc` et `secret_deca`. Enregistrer le résultat sur le disque sous le nom **image_finale.png**.

Observez le résultat de votre travail !

Rq

On constate alors que, sur les à-plats de couleur de l'image publique, on distingue l'image secrète par transparence.

6. Récupération de l'image secrète

10. Écrire une fonction `decryptage`, utilisant un décalage à gauche `A « n`, permettant de récupérer l'image secrète cachée.
Tester cette fonction sur **image_finale.bmp**. L'image obtenue sera enregistrée sous le nom **image_decrypte.png**.

Il ne vous reste qu'à tester cette fonction sur l'image enregistrée à votre nom.

7. Pour aller plus loin

7.1. Optimisation

Bien que fonctionnel, l'algorithme proposé n'est pas optimisé car il impose de parcourir une fois entièrement chaque image. Il est possible d'améliorer la rapidité d'exécution :

- Toutes les opérations (troncature, décalage et concaténation) peuvent être réalisée dans une seule boucle for au lieu d'avoir une boucle pour chaque traitement ;
- Il est inutile de parcourir toute l'image publique si l'image secrète est plus petite.

7.2. Dissimulation

Le camouflage ici réalisé est très sommaire car l'image secrète peut très facilement être retrouvée par différence entre l'image publique et l'image finale puis par augmentation du contraste.
Pour rendre la récupération de l'image secrète plus difficile, on peut par exemple changer l'ordre des pixels de l'image secrète lors du décalage des bits de poids forts. La nature de ce changement d'ordre peut être :

- fixée ;
- basée sur une clé de chiffrage/déchiffrage.