TP 9 - Images codées

Le but de ce TP est de manipuler des images couleurs et de décoder les messages cachées dans ces images.

On rappelle que pour pouvoir utiliser des images : il faut tout d'abord importer les modules `Image` et `numpy` afin de pouvoir effectivement ouvrir et traiter les images :

```
from PIL import Image
import numpy as np  #Tableaux à 3 dimensions hauteur x largeur x profondeur.
```

Ensuite pour ouvrir et afficher une image présente dans le répertoire, on utilise les commandes suivantes :

```
image = Image.open("nom.png") #Lecture du fichier.
image.show()  #Affiche l'image.
tab = np.array(image)  #Création d'un tableau contenant l'image.
nvImage = Image.fromarray(tab) #Création de l'image représentée par le tableau.
```

On rappelle également les commandes suivantes utilisables sur les tableaux `numpy` : `image.shape` renvoie les dimensions de l'image et `= np.zeros((n,m),dtype=np.uint8)` crée un tableau `numpy` de taille $n \times m$ contenant des 0.

1 Déchiffrement d'images

Un des membres des services secrets a envoyé deux images apparemment chiffrées. Vous avez pu les récupérer. Vous pensez qu'en combinant ces deux images, vous pourrez en obtenir une autre. En particulier, il est probable que l'image résultat soit une photo indiquant à quel endroit vous pourrez rencontrer l'agent double de l'autre camp.

Méthode de déchiffrement On dispose de deux images de mêmes dimensions, dont chaque pixel est codé en RVB sur trois octets (un octet rouge, un octet vert, un octet bleu). L'objectif est de combiner ces deux images pour obtenir une image résultat.

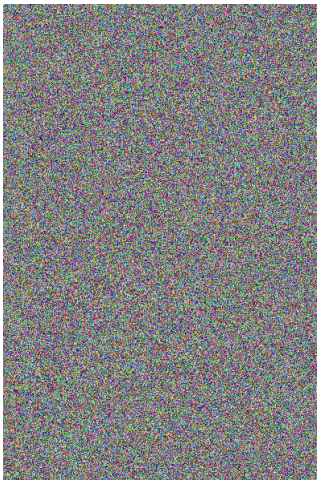


Image secrète n°1

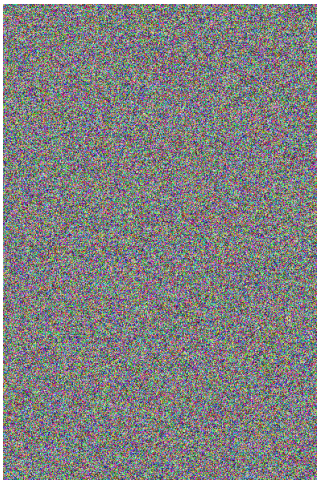


Image secrète n°2

L'image résultat, qui contient le message en clair, est une image en niveaux de gris, de mêmes dimensions que les deux images d'entrée. Chaque pixel de l'image résultat est codé sur un octet. Pour obtenir le pixel (i, j) de l'image résultat, on utilise uniquement les pixels (i, j) des deux images d'entrée comme suit :

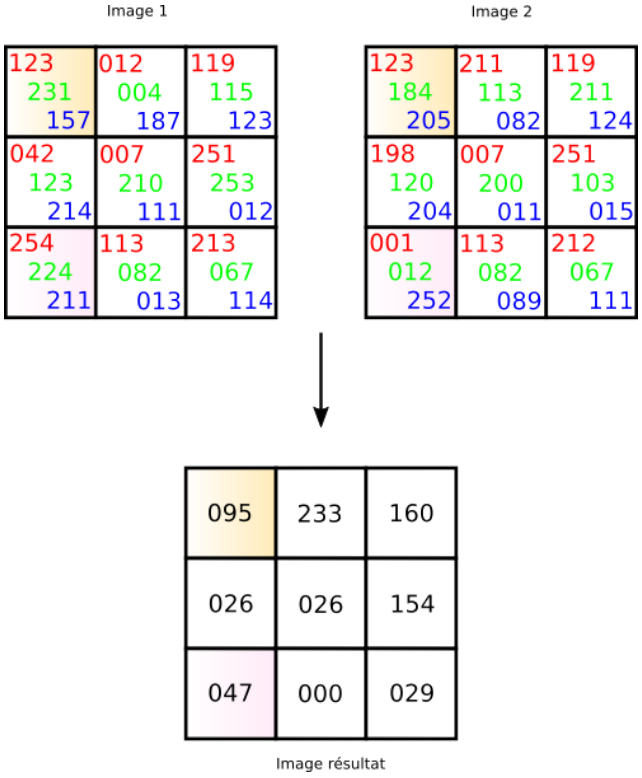
- si les valeurs de la composante rouge des deux pixels (i, j) d'entrée sont identiques, alors l'information est codée dans le plan vert c'est-à-dire que le pixel (i, j) de l'image résultat sera obtenu en faisant le "ou exclusif" des deux octets codant les niveaux de vert des pixels (i, j) des deux images d'entrées
- sinon les valeurs de la composante rouge des deux pixels (i, j) d'entrée sont différentes, alors l'information est codée dans le plan bleu c'est-à-dire que le pixel (i, j) de l'image résultat sera obtenu en faisant le "ou exclusif" des deux octets codant les niveaux de bleu des pixels (i, j) des deux images d'entrées.

L'opérateur XOR est l'opérateur logique "ou exclusif" dont la table de vérité est la suivante :

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

On remarque qu'elle vaut 1 uniquement si l'une des entrées vaut 1, pas les deux. En python, il existe une fonction permettant de réaliser ce "ou exclusif". Elle est disponible dans le module `operator` et s'appelle `xor` : `from operator import xor`.

Exemple Si les deux images de départ (format 3×3) sont celles en haut de la figure, alors on obtient par l'algorithme précédent la troisième image, située en bas :



Pour la case sur fond orange, les composantes rouges sont égales (123). L'information est donc codée sur la composante verte, comme le $\texttt{xor}(231, 184) = 95$, le pixel de l'image résultat correspondant vaut 095.

Pour la case sur fond rose, les composantes rouges sont différentes (254 et 001). Donc l'information est stockée dans les composantes bleues, comme le $\texttt{xor}(211, 252) = 47$, le pixel de l'image résultat correspondant vaut 047.

Question 1. Pour découvrir le lieu du rendez-vous, écrire une fonction `dechiffrement` prenant en arguments les deux noms de fichiers d'images à décodées et renvoyant l'image résultat déchiffrée grâce à la méthode de déchiffrement expliquée ci-dessus.

Quel est le lieu du rendez-vous ?

2 La stéganographie ou l'art de cacher des informations dans les images

Comment peut-on cacher une information dans une image ? Pour détailler l'opération nous travaillerons sur une image en niveau de gris. La couleur de chaque pixel est codée sur un octet et peut donc prendre 256 nuances différentes. Une image peut être vue comme un tableau dont chaque case représente un pixel.

Un texte caché de longueur $l \leq 255$ (c'est-à-dire composé de moins de 255 caractères, espaces compris) l'est de la façon suivante :

- la longueur l du message (correspondant au nombre de caractères) est codée sur 8 bits (c'est pour ça qu'on limite le message à 255 caractères mais on pourrait utiliser plusieurs octets si nécessaire) ;
- les bits de poids faible des huit premiers octets de l'image sont modifiés pour y cacher l (le premier bit étant celui de poids fort) ;
- le texte à cacher est découpé en caractères, eux-même traduits en paquets de 8 bits ;
- les bits de poids faible des 8 octets suivants sont modifiés pour intégrer le premier caractère en lui-même ;
- et ainsi de suite pour tous les octets du message (on suppose bien sûr que l'image est assez grande).

Exemple de construction

On souhaite cacher le mot "mot" dans une image de taille 4×8 (c'est suffisant ici). Notre mot fait 3 caractères. Ce qui donne sur un octet la valeur 00000011. Les codes ASCCI des lettres "m", "o" et "t" sont respectivement 109, 111 et 116 (rappelez-vous les fonctions `ord` et `chr`) ce qui donne respectivement $(01101101)_2$, $(01101111)_2$ et $(01110100)_2$. Sur une image contenant des pixels quelconques, le changement de la valeur de la nuance de gris donne quelque chose qui ressemble à ceci (les octets à gauche sont aléatoires) :

59	33	153	128	36	200	9	147
55	175	198	32	132	53	79	240
190	30	71	81	178	81	71	129
103	26	75	247	117	92	250	86

58	32	152	128	36	200	9	147
54	175	199	32	133	53	78	241
190	31	71	80	179	81	71	129
102	27	75	247	116	93	250	86

La première ligne est obtenue en modifiant tous les bits de poids faibles (correspondant à 2^0) des 8 octets de la première ligne de l'image. Les six premiers 59, 33, 153, 128, 36 et 200 voient leur bit de poids faible changés en 0 (si c'était déjà 0, pas de modification) et les deux derniers voient leur bit de poids faible changés en 1 (si il n'était pas déjà égal à 1). On obtient donc une nouvelle ligne dont les 6 premiers octets terminent par 0 (sont pairs) et les deux derniers terminent par 1 (sont impaires).

Vous devez avoir bien compris cet exemple avant de continuer !

Décoder un message Les deux questions qui suivent permettent de décoder un message caché dans une image selon la description ci-dessus. Les images seront toujours vues comme des tableaux `numpy` mais nous allons procéder à un petit changement afin de faciliter le traitement. Une image ayant une hauteur et une largeur, le tableau en résultant a aussi une hauteur et une largeur. Ce qui nous impose de faire attention aux fins de ligne lorsque nous traitons les octets. Pour éviter cela, il est possible de modifier la géométrie du tableau en utilisant la propriété *shape* (voir exemple de code ci-dessous).

```
#T est un tableau issu d'une image
ligne,colonne=T.shape #On récupère les dimensions du tableau.
T.shape=(1,ligne*colonne) #On modifie la géométrie du tableau.
T=T[0,:] #Le tableau n'a plus qu'une seule dimension.
#C'est à dire qu'il ne reste plus qu'une ligne.
#Tous les octets s'indice comme ceci T[i], i étant un indice quelconque.
#Pour retrouver les dimensions d'origine, il faut de nouveau modifier shape.
T.shape=(ligne,colonne)
```

Question 2. Écrire une fonction `decoderOctet` prenant en paramètre un tableau et l'indice d'une case de ce tableau notée `pos`. Elle retourne un entier compris entre 0 et 255 dont chaque bit est constitué des bits de poids faible des 8 cases consécutives du tableau depuis la position `pos`. Si on reprend l'exemple ci-dessus, `T` étant le tableau de droite, en appelant la fonction `decoderOctet(T,0)` on travaille depuis l'indice 0 (contenant le bit de poids fort) jusqu'à l'indice 7 (contenant le bit de poids faible). En testant la parité de chaque octet du tableau, on arrive à reconstruire l'octet $(00000011)_2$ et on renverra 3.

Question 3. En utilisant la fonction précédente, écrire une fonction `decoderMessage` prenant en paramètre une image et retournant le message décodé. Appliquer sur l'image *fete.png*.

Coder un message Dans la suite, on cherche à mettre au point les fonctions permettant de cacher un message.

Question 4. Écrire une fonction `valBit` prenant en paramètre une position et un entier. Elle retourne la valeur du bit à la position `pos`, c'est à dire 0 ou 1. Par exemple, avec une variable `A` valant 33, l'appel de la fonction `valBit(5,A)` retourne 1 car le 5-ième bit vaut 1 (le bit de poids faible a la position 0 et le bit de poids fort la position 7).

Question 5. Écrire une fonction `coderOctet` prenant en paramètre un tableau `T`, une position `pos` et un entier `octet`. Le tableau est issu d'une image (toujours un tableau numpy comme nous le faisons depuis le début), la position indique une case du tableau et l'entier est l'information à coder dans le tableau depuis la position `pos`. Cet entier est toujours compris entre 0 et 255. La fonction retourne le tableau modifié comme expliqué ci-dessus, c'est à dire que les cases d'indice `pos` à `pos+7` inclus ont leur bit de poids faible modifié en fonction de la valeur de l'entier passé en paramètre (reprendre l'exemple).

Question 6. Écrire une fonction `cacherMessage` prenant en paramètre une image et un message texte, et retournant une nouvelle image contenant le message caché. A vous de tester sur le thème que vous voulez avec le message que vous voulez.

3 Stéganographie ou l'art de cacher des images dans une image

Dans cette partie on souhaite cacher une image couleur dans... une autre image couleur. Qui pourrait penser que la première image est cachée dans la deuxième image ?

Image originale

Image originale

Résultat de l'incrustation de l'image de la crevette dans celle de la baleine.
On remarque une perte de qualité.

Aussi surprenant que ce le soit, il n'est pas bien difficile de réaliser ce travail. L'utilisation des opérateurs binaires bit à bit va grandement nous faciliter la tâche. Le principe est le suivant :

- l'image qui sera insérée sera plus petite que son image hôte ;
- toutes les composantes de couleur RVB de chaque pixel sont codées sur 8 bits (de 0 à 255) ;
- pour chaque pixel de l'image final et pour chaque composante de couleur, la valeur est le résultat d'une manipulation binaire telle que les 4 bits de poids fort sont issus du pixel ayant les même coordonnées de la première image, et les 4 bits de poids faibles sont les 4 bits de poids forts issus du pixel ayant les même coordonnées de la deuxième image. Par exemple, pour le pixel de coordonnées $(0, 0)$, c'est à dire celui qui est en haut à gauche, en ne prenant que la composante bleue :
  - on suppose que la valeur vaut 153 pour la première image, soit 10011001 en binaire ;
  - on suppose que la valeur vaut 200 pour la deuxième image, soit 11001000 en binaire ;
  - en agençant les bits des octets de la manière indiquée, cela donne 10011100 soit 160 en décimal. La valeur obtenue est très proche de la valeur de départ 153, les modifications sur l'image de départ sont donc minimes, voir imperceptibles. On ne verra pas l'incrustation (sinon il faut modifier le nombre de bits qu'on conserve). D'un autre côté, sur la valeur 200 on ne conserve que les 4 bits de poids forts, soit 11000000 ce qui donne 192. Encore une fois, celle valeur est très proche de la valeur de départ, on pourra donc reconstruire une image très proche de celle qui a été incrustée.

Principe des opérateurs binaires Nous n'allons utiliser les opérateurs `or` et `and` comme nous le faisons lors de tests logiques. Il existe d'autres opérateurs binaires qui travaillent sur tous les bits d'un coup et qui ne renvoie pas de booléen mais le résultat de l'opération. Pour tester ces différents opérateurs et bien comprendre leur fonctionnement, nous utiliserons directement la notation binaire sur des entiers. Ainsi, la valeur 23 peut s'écrire `0b00010101` en python. Le préfixe `0b` permet d'indiquer à Python que le nombre est au format binaire naturel..

Nous allons utiliser les opérateurs suivants :

- `|` (OU logique bit à bit) : il renvoie l'opération d'un OU logique réalisée bit à bit entre deux ensembles de bits de même taille. Par exemple, `0b0010 | 0b0001` renvoie `0b0011`.
- `&` (ET logique bit à bit) : il renvoie l'opération d'un ET logique réalisée bit à bit entre deux ensembles de bits de même taille. Par exemple, `0b0111 & 0b0010` renvoie `0b0010`.
- `»` (opérateur de décalage à droite) : il renvoie l'opération d'un décalage de tous les bits vers la droite. Ceux qui sortent sont perdus, ceux qui rentrent valent 0. Par exemple, `0b1110 >> 2` renvoie `0b0011`.

- « (opérateur de décalage à gauche) : il renvoie l'opération d'un décalage de tous les bits vers la gauche. Ceux qui sortent sont perdus, ceux qui rentrent valent 0. Par exemple, `0b1110 << 1` renvoie `0b11100`.

Tests dans la console Anticipez les résultats des opérations suivantes avant de les tester dans la console.

Opérateur	Opérations à tester sur des entiers de 4 bits		
OU logique `|`	`0b0001 | 0b0100`	`0b1100 | 0b0011`	`0b1101 | 0b1100`
ET logique `&`	`0b0001 & 0b0111`	`0b1100 & 0b0110`	`0b1000 & 0b1110`
décalage droite `>>`	`0b0110 >> 1`	`0b0111 >> 2`	`0b1000 >> 4`
décalage gauche `<<`	`0b0110 << 1`	`0b1110 << 2`	`0b1001 << 4`

Extraire une image cachée

Question 7. En utilisant uniquement les opérateurs vu précédemment, indiquer la suite d'opérations nécessaires permettant d'obtenir deux entiers sur 8 bits dont les 4 bits de poids fort du premier sont issus d'un autre entier noté n et les 4 bits de poids faibles sont à 0. Les 4 bits de poids fort du deuxième entier sont les 4 bits de poids faible de n, ses 4 bits de poids faible valant 0. Cela peut se résumer avec les éléments suivants : depuis `n=0b01101001` , on veut obtenir `0b01100000` et `0b10010000`.

Question 8. L'image de la baleine incrustée de l'image de la crevette vous est fourni dans le répertoire. L'incrustation suit exactement ce qui a été décrit avant. Écrire le code de la fonction `extraireImage` prenant en paramètre le nom d'une image et affichant le résultat de l'extraction avec 4 bits de poids faible. L'image incrustée étant plus petite que l'image hôte, on appliquera le traitement sur toute l'image hôte. Ce qui permettra de voir le résultat dans tous les cas.

Question 9. Parmi les images fournies pour le TD, il y en a une qui représente Astérix et Obélix. Une autre image de ces deux personnages se cache à l'intérieur. Pour ne pas voir l'incrustation, seul les deux premiers bits de poids fort de chaque pixel de l'image à incruster ont été gardés pour devenir les bits de poids faible. Écrire le code de la fonction `extraireImage2` (modification de la fonction précédente) prenant en paramètre le nom d'une image et affichant le résultat de l'extraction avec seulement 2 Bits de poids faible. Que rencontrent Astérix et Obélix dans la forêt ?

Incruster une image

Question 10. En utilisant uniquement les opérateurs vu précédemment, indiquer la suite d'opérations nécessaires permettant d'obtenir un entier sur 8 bits dont les 4 bits de poids fort sont issus d'un deuxième entier et les 4 bits de poids faible sont les 4 bits de poids forts d'un troisième entier. Cela peut se résumer avec les éléments suivants : `0b01100111` avec `0b10001001` doit donner `0b01101000` .

Question 11. En utilisant deux images de votre choix, écrire le code de la fonction `cacherImage` prenant en paramètre deux noms d'image et sauvegardant une nouvelle image ressemblant à la première incrustée de la deuxième. S l'incrustation se devine, modifier le nombre de bits poids fort à garder et ajuster vos fonctions.