LYCÉE MARGUERITE YOURCENAR - MORANGIS

Stéganographie et cryptologie visuelle

Projet d'Informatique et Sciences du Numérique

Groupe : Ariane Kayvantash et Solène Hirles

Dossier personnel de Solène Hirles

2018-2019

Sommaire

Présentation du projet

Suite à nos projets personnels pour la suite de nos études, Ariane dans le traitement d'image et moi dans le cyber sécurité, on a décidé de réunir ces deux domaines pour en faire un projet qui nous permettrait de commencer à les pratiquer. On s'est donc intéresser à la cryptographie d'image et grâce à notre professeure d'Informatiques et Sciences du Numériques, Mme Loger, on a aussi trouvé une autre méthode : la stéganographie. On a donc décidé de créer un projet dans le but de cacher une image que seul le destinataire pourrait voir à l'aide des deux méthodes que sont la stéganographie et la cryptographie visuelle.

La stéganographie consiste à dissimuler une information dans une autre. Cette méthode est très vieille. En effet, elle est connue depuis l'antiquité et possède de nombreuses variantes en fonctions du support, de la méthode ou de la technologie existante. Dans notre cas, on cherche à dissimuler une image dans une autre en en créant une troisième. On cherche à l'aide de cette méthode que le destinataire qui reçoit l'image obtenue ne puisse pas supposer qu'il existe un contenue dissimulée dans celle-ci. Ainsi si quelqu'un intercepte cette image, il ne pourrait pas empêcher de transmettre l'image cachée et utilisée la même méthode afin de transmettre de fausses informations. Pour la partie stéganographie de notre projet, on va chercher à réaliser la méthode LSB (Least Significant Beat) qui consiste à remplacer les bits de poids faible de l'image contenante.

Néanmoins, la stéganographie reste une méthode assez connue. Ainsi si l'on suppose un contenu d'en dissimuler un autre grâce à cette méthode, il est facile de réussir à le récupérer. Aujourd'hui, il existe d'autres méthodes qui permettent de rendre une information incompréhensible à n'importe quelle autre personne qui n'est pas le destinataire.

La cryptologie permet de rendre une information secrète. Elle comprend la cryptographie, le codage et la cryptanalyse, le décodage. Depuis les années 1970, elle est devenu un thème de recherche suite à l'essor de l'informatique puis d'internet et des communications. La cryptologie visuelle a pour but que l'image transmisse ne doit pas pouvoir être authentifiable par l'œil humain sans avoir été au préalable décodé. Dans notre projet, nous utiliserons une méthode de cryptographie à clé secrète permettant de chiffrer et de déchiffrer à partir de la même clé. Mais pour cela, l'image devra être binarisée c'et-à-dire être en noir et blanc, sans niveau de gris et la clé sera une image de la taille de celle à cacher qui sera composé aléatoirement de pixels noirs et de pixels blancs.

Cahier des charges

Le projet se rapporte aux domaines de compétences suivants :

- **Dimension algorithmique** : Créer des algorithmes permettant de coder une image selon deux méthodes : la stéganographie et la cryptographie visuelle à clé secrète.

- **Éléments de programmation** : Pour réaliser cela, nous utilisons le langage de programmation Python, un éditeur de programme et une console qui affiche les résultats du programme exécuté dans l'éditeur. Nous avons utilisés ces modules et ces bibliothèques : PIL Images (pour travailler sur les images et les afficher), numpy (pour créer des tableaux et les convertir en images) et Tkinter (pour l'interface utilisateur).

Production finale attendue : Un algorithme qui permet de coder n'importe quelle image selon une clé générée aléatoirement et des algorithmes de codage et décodage permettant de cacher une image dans une autre.

Caractéristiques de la production finale : Le codage de la stéganographie fonctionne ainsi que l'interface. Le codage des la cryptographie fonctionne partiellement. Cependant, le décodage de la stéganographie et celui de la cryptographie ont des problèmes de fonctionnement.

Contraintes à respecter : Date butoir, travail en équipe, dossier écrit (5-10 pages)

Matériel et logiciel à mettre en œuvre : Python

Répartition des tâches

Solène	Ariane
Codage stéganographie	Interface de l'utilisateur
Algorithme de binarisation	Décodage stéganographie
Codage de la cryptographie	Décodage cryptographie

Réalisation

Stéganographie

On a vu que le principe de la stéganographie consiste à dissimuler une image dans une autre. Le codage et le décodage se font en modifiant les bits de chaque octet de chaque pixel de chaque images c'est-à-dire de l'image à cacher et de l'image contenante. Plus les images sont grandes, plus cela prendra du temps.

Principe du codage

On va travailler avec deux images de la même taille. On nommera l'image à cacher « l'image secrète » et l'image la contenant et qui sera visible « l'image conteneur ».

On travaille avec des images en couleurs donc chaque pixel est codé sur quatre octets au maximum : les trois premiers servant à coder les proportions des composants de couleur qui sont le rouge, le vert et le bleu et le quatrième code pour la transparence. Le codage de la stéganographie s'effectue en deux étapes.

- Premièrement, on met à zéro les quatre bits de poids faible (les quatre bit qui se trouvent à droite et qui possède la valeur la plus faible) de chaque octet de chaque pixel de l'image conteneur. On garde les quatre bits de poids fort de chaque octet de chaque pixel de l'image car ce sont eux qui contiennent l'essentiel des informations de l'image qui doit rester visible. Sur chaque pixel de l'image secrète, on décale vers la droite les quatre bits de poids fort de chaque octet qui deviennent alors les quatre bits de poids faible. Néanmoins, ces quatre bits contiennent l'essentiel de l'image à cacher. En déplaçant en bit de poids faible, nous cachons ces informations en les empêchant de s'exprimer.

- Dans un second temps, on possède alors une image qui ne possède plus de quartet de poids faible sur ses octets. C'est l'image conteneur. On a aussi une image qui n'a plus de quartet de poids fort sur ses octets. C'est l'image secrète. On assemble donc ces deux composants sur chaque octet en effectuant une somme entre eux. Les informations de l'image secrète se retrouvent alors dans les quatre bits de poids faibles de l'image conteneur.

⇨ *Voir illustrations dans Annexes : Figure 1*

Mise en œuvre

⇨ *Voir le code dans Annexes : Figure 2*

J'utilise la bibliothèque PIL pour travailler sur le projet. On utilise le module *Image* afin de pouvoir travailler sur les images. On commence tout d'abord par demander les images que l'utilisateur veut utiliser avec la commande *input.* On les nomme ensuite « *image_conteneur* » et « *image_secrete* ». On utilise ensuite le module Image pour ouvrir et lire les images.

Pour réaliser le codage de la stéganographie, on doit créer deux fonctions : la fonction **cache** qui a *n* pour argument et la fonction **decalage** qui possédant *n* et la variable *decale* qui est égale à 4 pour arguments.

La fonction **cache** permet de mettre à zéro le quartet de bits de poids faible mais de garder les quatre bis de poids fort. Pour cela, on effectue un ET logique entre la valeur d'un octet, *n* et le nombre binaire **0b11110000. 0b** indique à Python qu'il s'agit d'un nombre binaire. On a choisit ce nombre car il permet de mettre les quatre bit de poids faible à zéro.

La fonction **decalage** permet de faire passer les bits de poids fort en bits de poids faible. Pour cela, on doit effectuer un décalage vers la droite. Cela est possible grâce au symbole **>>**. Ainsi, on décale les bits de l'octet n de la valeur de la variable *decale* vers la droite. Or *decale* est défini pour la valeur 4 donc on décale l'octet n de quatre bits vers la droite. On a donc fait passer les quatre bits de poids fort en quatre bits de poids faible.

On crée ensuite le corps principal du programme. Pour cela, on fait une boucle SI car on ne peut cacher une image dans une autre que si elles sont de même taille. On récupère la taille de chaque image avec l'instruction *nom_image.size* et on les compare. Si elles ont la même taille le programme se poursuit sinon il affiche qu'elles doivent être de même taille. On crée deux variables *l* et *h* qui prennent les valeurs de la taille d'une des deux images (n'importe laquelle étant donné qu'elles ont la même taille). On crée un nouvelle variable qui contiendra la nouvelle image qu'on va créer. Pour cela, on déclare la création d'une nouvelle image avec l'instruction *Image.news('RGB', (l, h)).* Dans les parenthèses, on indique d'abord le type de l'image, ici 'RGB' qui indique qu'il s'agit d'une image en couleur. Puis, on indique la taille souhaité de l'image. Comme on veut afficher l'image conteneur, on met les deux variables *l* et *h* qui contiennent les valeurs de sa taille. On parcourt alors chaque ligne des images avec une première boucle for et dans celle-ci, on parcourt chaque colonne des images avec une autre boucle for. On crée trois nouvelles variables : *rouge1, vert1,* et *bleu1*. Chacune sera attribuée à un composant de couleur. On utilise l'instruction *nom_image.getpixel((coordonnées du pixel))* pour récupérer les octets des trois composants de couleur du pixel que l'on affecte aux trois variables. On fait d'abord cela sur l'image conteneur. On applique ensuite chacune à leur tour, ces variables dans la fonction **cache** afin de seulement conserver les quatre bit de poids fort de l'image conteneur. Ensuite, on crée à nouveau trois variables : *rouge2, vert2* et *bleu2* auxquelles on affecte les octets des trois composants de couleurs du pixel de l'image secrète récupérés avec la même instruction. On applique ces variables, chacune à leur tour dans la fonction **decalage** afin de faire passer le quartet de poids fort en quartet de poids faible. On applique ensuite l'instruction *putpixel* qui permet de modifier la couleur d'un pixel sur l'image finale, la nouvelle image que l'on crée. Pour chaque pixel, on additionne les deux octets de chaque composant de couleur de deux images. Par exemple, pour un pixel de coordonnées (x,y) de l'image finale, chacun de ces octet de composants de couleurs est divisible en deux : les quatre bits de poids fort qui correspondent au quatre bits de poids fort de l'octet du même composant de couleur du pixel de même coordonnées de l'image conteneur et les quatre bits de poids faible qui correspondent aux quatre bits de poids fort de l'octet du même composant de couleur du pixel de même coordonnées de l'image secrète. Pour finir, on sort des deux boucles for et on sauvegarde l'image avant de l'afficher.

Algorithme de binarisation

La réalisation de cet algorithme était nécessaire pour la cryptographie car on souhaite générer une clé en noir et blanc. Or, il est plus efficace de dissimuler une image binarisée dans une clé en noir et blanc qu'une image en couleur.

Principe de l'algorithme

Binariser une image, c'est créer une nouvelle image faite uniquement avec des pixels noirs et des pixels blancs. Les pixels sont regroupés entre eux selon des critères définis. En effet, la binarisation d'une image revient à regrouper les pixels en seulement deux groupes : noirs et blancs. Pour cela, il faut définir un seuil en dessous duquel les pixels deviennent blancs et au-dessus, ils deviennent noirs.

Mise en œuvre

⇨ *Voir le code dans Annexes : Figure 3*

Pour réaliser cela, j'ai travaillé avec les bibliothèques numpy et PIL dont j'ai principalement utilisé le module *Image.*

J'ai d'abord commencé par créer une fonction **seuil** qui a pour argument *x.* Cette fonction permet de séparer les pixels en deux groupes et de transformer un groupe en pixel noir et l'autre en pixel blanc. On crée une boucle if. On veut que si la valeur d'un pixel est inférieure à 129, la fonction le transforme en un pixel blanc. Pour cela, on utilise l'instruction *return 0* où 0 correspond à la valeur d'un pixel noir. Par contre, si le pixel est supérieur ou égale à 129, la fonction renvoie la valeur 255 qui correspond à un pixel blanc.

Je me suis ensuite occupée du corps principal du programme. Pour cela, j'ai créé une variable qui ouvrait et lisait l'image avec l'instruction *Image.open('nom_image')*. J'ai crée une liste qui a récupéré les valeurs des pixels de l'image grâce à l'instruction *.getdata().* J'ai récupéré la taille de l'image que j'ai affectée à deux nouvelles variables. J'ai ensuite crée une liste *listetrans* dans laquelle j'ai appliqué la fonction **seuil** sur tout les éléments de la liste des valeurs des pixels, obtenue précédemment. Ensuite, j'ai crée une nouvelle image de type *« L »* (image en noir et blanc dont chaque pixel est codé sur un octet seulement) et de même taille que l'image sur lequel le programme a été effectué. J'ai ensuite converti les valeurs de la liste *listetrans* en pixel sur la nouvelle image avec l'instruction *.putdata(listetrans).* J'ai ensuite terminé par sauvegarder la nouvelle image. Cet algorithme ne fonctionne que sur les images en niveau de gris. Nous l'utiliserons donc pour binariser la clé et devrons créer un autre algorithme de binarisation pour les images en couleurs.

Cryptographie

Principe du codage

Le codage de la cryptographie s'effectue sur une image binarisée, c'est-à-dire en noir et blanc. Nous allons donc utiliser un algorithme de binarisation. Puis le chiffrement de cette image se fera en trois étapes.

- Premièrement, on crée un tableau de la même taille que l'image à cacher contenant des entiers entre 0 et 256, 256 étant exclus. Ainsi pour chaque pixel de l'image, un entier aléatoire lui est associé dans le tableau.

- Deuxièmement, on crée la clé qui doit être en noir et blanc. Pour cela, on transforme le tableau en une image puis on la binaise grâce à l'algorithme de binarisation. On obtient alors la clé. Celle-ci doit rester secrète et être transmise par un autre biais que l'image.

- Troisièmement, on effectue un OU exclusif entre chaque composant du pixel de la clé et du pixel de l'image à cacher. On obtient une nouvelle image qui a toujours la même taille que l'image d'origine.

Mise en œuvre

⇨ *Voir le code dans Annexes : Figure 4*

On travaille avec PIL Image et les modules *numpy, math et random*.

J'ai ensuite créé deux fonctions : la fonction **seuil** et la fonction **noirblanc.** La fonction **seuil** permettra de séparer les composants de couleur du pixel en deux groupes. C'est la même que celle de l'algorithme de binarisation. La fonction **noirblanc** permet de comparer les groupes des composants de couleur du pixel. Avec une boucle *if*, on créer un pixel noir si les trois composantes ont 0 comme valeur sinon on créer un pixel blanc.

Comme pour la stéganographie, on récupère l'image à cacher avec l'instruction *input* et on l'ouvre et on la lit avec PIL Image. On récupère la taille de l'image que l'on affecte à deux variables. Puis on crée une nouvelle image de la même taille et avec le même mode que l'image secrète. On parcourt chaque pixel de l'image avec deux boucles *for* une pour les lignes et une pour les colonnes. On récupère la valeur des trois composantes du pixel dans une liste grâce à l'instruction *nom_image.getpixel(coordonees_du_pixel).* On créer une variable p qui correspond au résultat de la fonction **noirblanc** dans laquelle les trois variables sont les résultats de la fonction seuil sur les trois composants de couleur du pixel obtenu précédemment autrement dit sur les trois éléments de la liste créer. Ensuite, on modifie le pixel de même coordonnées dans la nouvelle image avec les valeurs de p. Cela est possible grâce à l'instruction *.putpixel().* On sort des boucles *for* puis on sauvegarde l'image qui est maintenant binarisée. Pour finir, j'ai récupérer la liste de pixels de l'image obtenue grâce à l'instruction *.getdata().*

Je me suis ensuite occupée du corps principal du programme qui nécessite la génération de la clé, l'application du OU exclusif et l'obtention de l'image finale. Tout d'abord, on commence par générer la clé. Pour créer un tableau d'entiers aléatoire compris entre 0 et 255 inclus, on utilise *numpy* pour créer le tableau et *random.randint* pour générer des entiers

aléatoires. Au final, on a utilisé cette instruction *numpy.random.randint(256, size= nom_image.size)* que l'on a affecté à une variable. On crée donc un tableau de la taille de l'image possédant des entiers généré aléatoirement compris entre 0 et 256, 256 étant exclus. On doit ensuite convertir le tableau en image. Cela nécessite l'utilisation de l'instruction *Image.fromarray(),* dans laquelle on a placé la variable affectée au tableau entre les parenthèses. A partir de la, on a obtenu une image en couleur. Or on veut que la clé soit en noir et blanc. On va donc binariser cette image à l'aide du corps principal de l'algorithme de binarisation que l'on a repris en entier hormis la première ligne qui consistait à ouvrir et lire l'image (car cela est déjà fait dans ce programme). On l'a adapté sur cet algorithme là en modifiant les noms des variables.

On doit ensuite réaliser un OU exclusif entre chaque composant du pixel clé et du pixel de l'image secrète. On a commencé par créer une liste vide qui sera rempli des nouvelles valeurs des pixels par la suite. On crée une variable i qu'on initialise. On souhaite comparer les éléments qui sont au même rang dans la liste des pixels de la clé et dans celle des pixels de l'image à cacher. Pour cela, on crée une boucle *while* qui sera utilisé tant que i sera inférieur à la longueur de la liste des pixels de l'image secrète. On ajoute une bouche if. En effet, on souhaite que si les deux éléments sont égaux alors on crée un pixel noir dans la liste vide sinon on créé un pixel blanc dans cette même liste. Pour créer un pixel blanc ou noir dans la liste vide, on remplit celle si avec la valeur d'un pixel blanc soit 255 ou avec les valeurs d'un pixel noir soit 0. On remplit la liste vide avec l'instruction *nom_liste.append(valeur).* On sort de la boucle if et on augmente la valeur de i de 1 en faisant *i=i+.* On sort de la boucle *while* et on créer une nouvelle image de type *« L »* et de même taille que l'image secrète. On souhaite remplacer les pixels cette nouvelle image par les pixels dont on a les valeurs dans la liste que l'on vient de remplir. On utilise l'instruction nom*_image.putdata(liste)* pour effectuer cela. Enfin, on sauvegarde la nouvelle image créée qui est l'image finale.

Les difficultés rencontrées

En travaillant sur ce projet, j'ai rencontré de nombreuses difficultés.

Tout d'abord, sur ma première version du codage de la stéganographie, j'ai souhaité utilisé le module imageio. Or celui-ci nécessitait des installations et cela était trop compliqué. J'ai donc dû chercher un autre module « module PIL » permettant de travailler sur les images et réalisant les mêmes instructions que celui-ci. Néanmoins, son application sur le programme d'origine était assez compliquée. J'ai donc crée un nouveau programme de codage de stéganographie utilisant principalement PIL Images.

La partie sur laquelle j'ai rencontré le plus de difficultés, c'est la cryptographie. En effet, au départ, je souhaitais créer une clé assez complexe qui aurait été générée à partir d'un tableau aléatoire contenant aléatoirement les entiers 0 et 1. Je voulais associer au chiffre 1 un pixel noir suivi d'un pixel blanc et l'inverse pour le chiffre 0.

Comme la date butoir approchait, j'ai décidé alors de simplifier la clé en générant un tableau avec des entiers aléatoires compris entre 0 et 255 inclus. Ensuite, je devais convertir ce tableau en une image pour le binariser. Cela a nécessité de longues recherches.

La deuxième grande difficulté de cette partie a été de réaliser la fonction XOR, qui ne fonctionne toujours pas. Dans un premier temps, il a fallu effectuer beaucoup de recherches afin de comprendre le principe de la fonction. Une fois que j'ai compris ce qu'elle devait faire. J'ai dû chercher comment elle pouvait accomplir cela. Premièrement, il fallait trouver comment parcourir deux listes en même temps en étant toujours au même rang sur les deux. Après des recherches et des tests, j'ai trouvé une instruction qui me semble capable de faire cela.

Bilan

Je suis très contente d'avoir choisi la spécialité d'Informatique et Sciences du Numériques car j'ai pu approfondir mes connaissance en informatique et découvrir plus profondément la voie dans laquelle je souhaite m'engager. En effet, j'ai pu découvrir plus profondément la stéganographie et la cryptographie et apprendre à la coder. J'ai aussi appris à travailler en groupe. J'ai trouvé cela positif car cela nous a permis d'échanger entre nous, de nous entraider dans les recherches et les difficultés rencontrées. On a également pu partager nos idées. J'ai aussi dû à être autonome, productive et à respecter un cahier des charges ainsi qu'une date butoir. Je pense que cela est un bon départ pour mes études car je devrais très souvent travailler en équipe et être encore plus autonome. De plus, comme je souhaite travailler dans le cyber défense, ce projet m'a permis d'acquérir des bases qui me serviront pour mes études.

Annexes

Figure 1

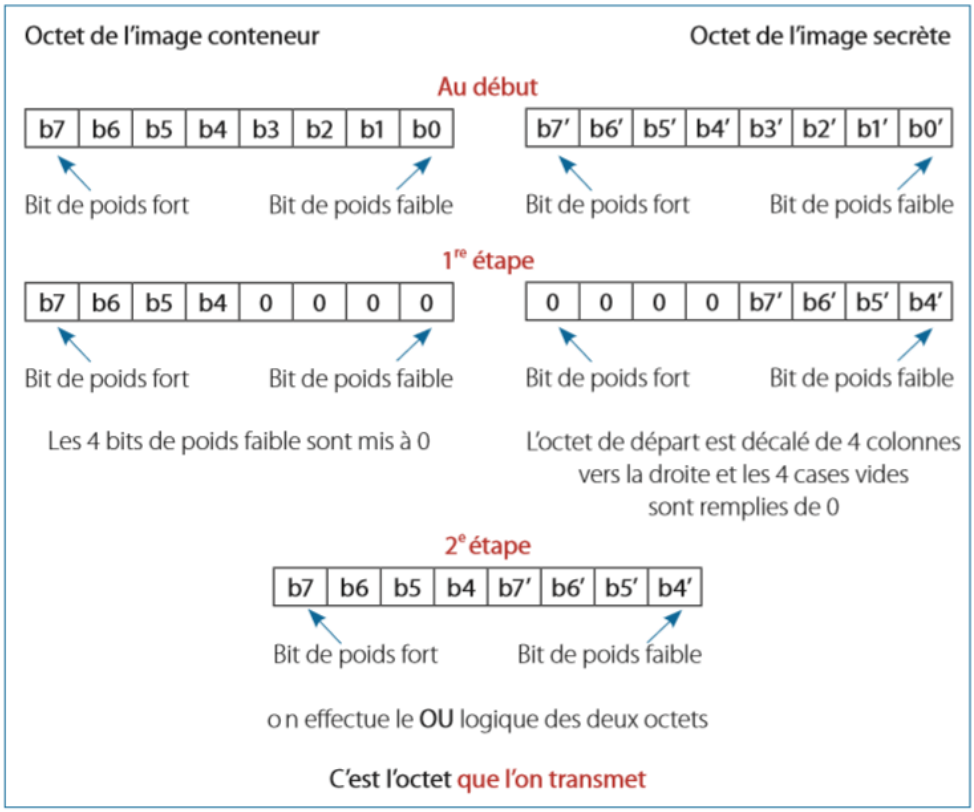

Figure 2

```python
from PIL import Image

image_A=input("copier le nom de votre image secrète ainsi que l'extention")
image_B=input("copier le nom de votre image conteneure ainsi que l'extention ")
image_secrete= Image.open(image_A) #On lit les images grâce au module PIL Image
image_conteneur= Image.open(image_B)

#Les fonctions
def cache (n):
    """cache les bits de poids faibles pour seulement laisser les bits de poids forts visibles"""
    return n & 0b11110000 #On effectue un ET logique entre la valeur d'un octet correspondant à une composante de couleur des
    # pixels et un nombre binaire indiqué sur python par 0b qui permet de mettre à zéro les quatres bits de poids faibles

def decalage (n,decale=4):
    """ cette fonction a pour but de faire passer des bits de poids forts en bits de poids faibles"""
    return n >> decale #On effectue un décalage de 4 bits vers la droite dans l'octet, ainsi les quatres bits de poids forts
    # deviennent quatres bits de poids faibles

#Le corps principal du programme
if image_secrete.size==image_conteneur.size: #On verifie que les deux images ont la même taille
    l,h=image_conteneur.size # On récupère la longueur et la hauteur de l'image visible
    image_finale= Image.new('RGB', (l,h)) #On créer une nouvelle image en couleur et de la même taille que les images précédentes
```

```python
    for x in range (l): #On parcourt chaque ligne
        for y in range (h): # on parcourt chaque colonne
            rouge1, vert1, bleu1= image_conteneur.getpixel((x,y)) #On récupère les trois composants de couleurs des pixel de
            # l'image qui restera visible
            rouge1, vert1, bleu1= cache(rouge1), cache(vert1), cache(bleu1) # On garde uniquement les quatres bits de poids forts
            # de chaque composant de couleur du pixel afin que l'image conteneur reste visible mais qu'on puisse cacher l'image secréte
            rouge2, vert2, bleu2= image_secrete.getpixel((x,y)) #On récupère les trois compsants de couleurs des pixels de l'image
            # à cacher
            rouge2, vert2, bleu2= decalage(rouge2), decalage(vert2), decalage(bleu2) #On conserve les quatres bits de poids forts qu'on
            # décale en bits de poids faibles
            image_finale.putpixel((x,y),(rouge1+rouge2, vert1+vert2, bleu1+bleu2)) #putpixel permet de modifier la couleur d'un pixel
            #On combine les 2 octets de chaque composant des deux images. Ainsi, on a sur l'octet final les quatres bits de poids forts
            # qui sont ceux de l'image visible et les quatres bits de poids faibles sont les quatres bits de poids forts de l'image
            # cachée

    image_finale.save('image_fianle.png') #On sauveagarde la nouvelle image créée
    image_finale.show() #On l'affiche
else:
    print("Les images doient être de la même taille")
```

Figure 3

```python
# Créé par hirles, le 04/04/2019 en Python 3.2

#but: passer les images en noir et blanc
from numpy import*
from PIL import Image

def seuil(x): #fonction qui sépare les pixels en deux groupes
    if x < 128:
        return 0 #transformation en pixel noir
    else:
        return 255 #transformation en pixel blanc

#Image secrète
im = Image.open("image_secrete.png")
liste_secrete = im.getdata() #création de la liste des pixels
a,b=im.size #on prend la taille de l'image

listetrans= [seuil(x) for x in liste_secrete]#on applique la fonction sur la liste
result = Image.new("L",(a,b)) #création de la nouvelle image
result.putdata(listetrans)
result.save("image secrete.png")
```

Figure 4

```python
#Fonction pour coder une image dans une image (cryptographie)
def code_image():
    import numpy, random
    from PIL import Image
    import math

    def seuil(x):
        """Fonction qui permet de séparer les pixels en deux groupes"""
        if x<128:
            return 0 #transformation en pixel noir
        else:
            return 255 #transformation en pixel blanc

    def noirblanc(r,g,b):
        """Fonction qui permet de créer des pixels noirs et des pixels blancs"""
        if r==0 and g==0 and b==0:
            return(0,0,0) #pixel noir
        else:
            return(255,255,255) #pixel blanc

    #Corps principal
    image_A= input("copier l'url de votre image secrète avec l'extention")
    image_secrete = Image.open(image_A) #On lit l'image

    #Binarisation
    #liste_secrete = image_secrete.getdata() #création de la liste des pixels


    a,b=image_secrete.size #on prend la taille de l'image
    imgF = Image.new(image_secrete.mode,image_secrete.size)
    for i in range(b):
        for j in range(a):
            pixel = image_secrete.getpixel((j,i)) # récupération du pixel

            p = noirblanc(seuil(pixel[0]), seuil(pixel[1]), seuil(pixel[2]))

            # composition de la nouvelle image

            imgF.putpixel((j,i), p)
    imgF.save("image secrete.png")
    liste_secrete=imgF.getdata()


    #Génération de la clé
    tableau_aleatoire = numpy.random.randint(256, size =(a,b))# une tableau de la taille de l'image avec des entier entre 0 et 256, 2r6 exclus.
    cle=Image.fromarray(tableau_aleatoire) #on convertit le tableau en une image, qui est la clé
    #On cherche à transformer la clé en une image en noir et blanc
    liste_cle = cle.getdata() #création de la liste des pixels


    listetrans=[seuil(x) for x in liste_cle] #on applique la fonction sur la liste

    result_cle = Image.new("L",(a,b)) #création de la nouvelle image
    result_cle.putdata(listetrans)
    result_cle.save("clé.png") #On sauvegarde la clé dans un fichier afin qu'elle puisse être envoyer pour le décodage

    liste_codee=[]
    i=0

    while i< len(liste_secrete):

        if liste_secrete[i][0] == listetrans[i]: #On compare les éléments des deux listes
            liste_codee.append(0) # On ajoute la valeur d'un pixel noir dans la liste des pixels de l'image finale
        else:
            liste_codee.append(255) # On ajoute la valeur d'un pixel blanc dans la liste des pixels de l'image finale
        i+=1
    imgcodee =  Image.new("L",(a,b))
    imgcodee.putdata(liste_codee)
    imgcodee.save("image_codee.png")
```