TP 1 - ILLUSTRATIONS ON IMAGE ENCRYPTION AND STEGANOGRAPHY

HTI

Digital Content Protection

Marilou Grignoli

Andrea Miens

Mise en Place :

Ce TP a pour objectif de mettre en place différentes méthodes d'encodage et décodage d'image afin de voir lesquelles sont exploitables et leurs faiblesses. Pour réaliser l'ensemble de ces méthodes d'encryptement, nous coderons en python en utilisant différentes bibliothèques :

- numpy
- opencv (appelé cv2 en python)
- PIL
- random

Example 1 : CAESAR cipher applied to images – Lena test image

La première étape est de charger l'image de lena en nuance de gris en une matrice (les valeurs seront donc comprises entre 0 et 255). Pour ce faire, on a réalisé la fonction chargement :

```
path = "../TP-1/TP-HTI/lena.png"

def chargement(path):
    img = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
    matim = np.array(img)
    return matim
```

Cette première méthode d'encodage consiste à appliquer la méthode du code César, c'est-à-dire faire un décalage d'une valeur constante à l'ensemble des valeurs de luminance des pixels de l'image.

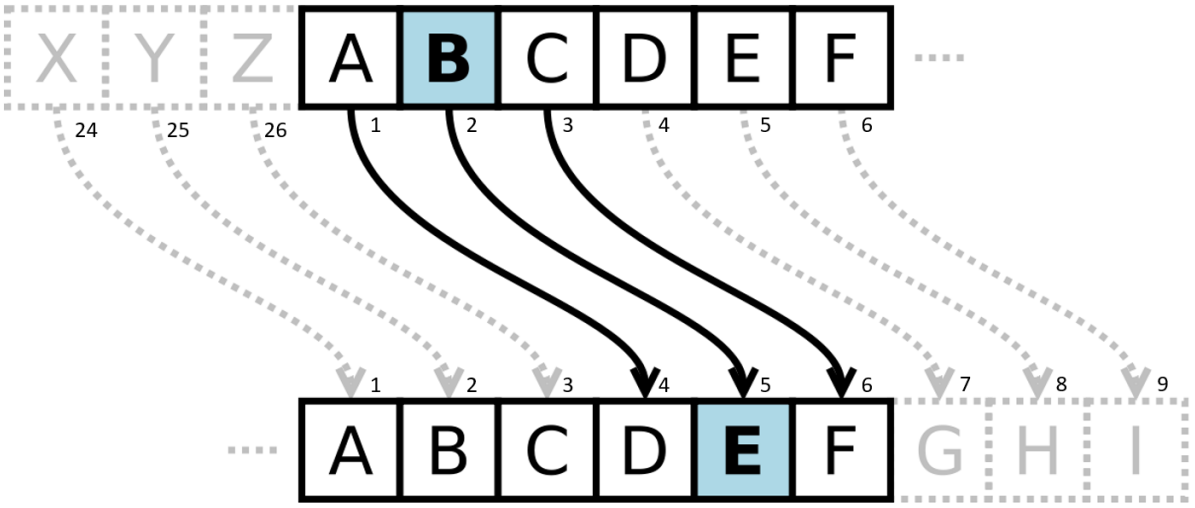


Figure 1 : Idée générale du code César, image modifiée à partir de celle de Wikipédia

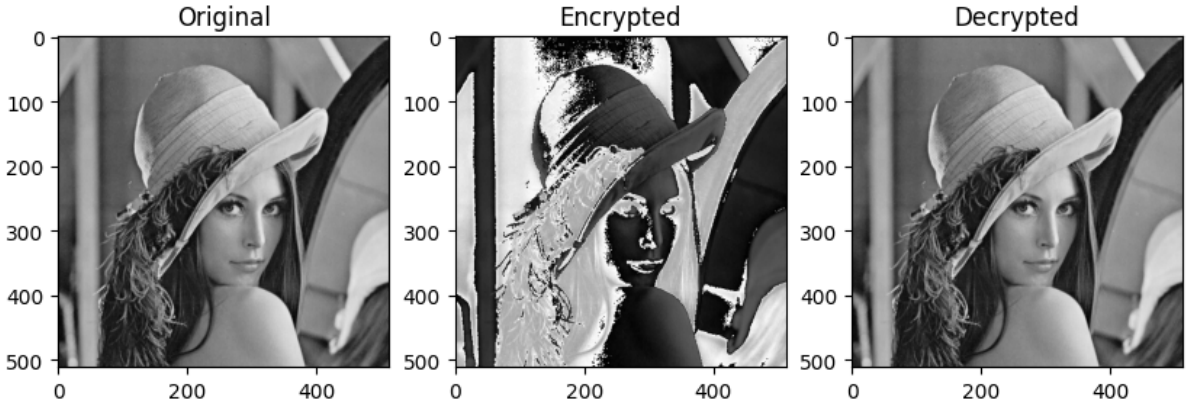
En appliquant cette idée aux valeurs de l'image, on obtient les fonctions d'encodage et décodage suivantes :

```
def encryption_cypher(img, dec):
    for x in range(0, np.size(img, 0)):
        for y in range(0, np.size(img, 1)):
            img[x][y] = (img[x][y] + dec)%256
    return img
```

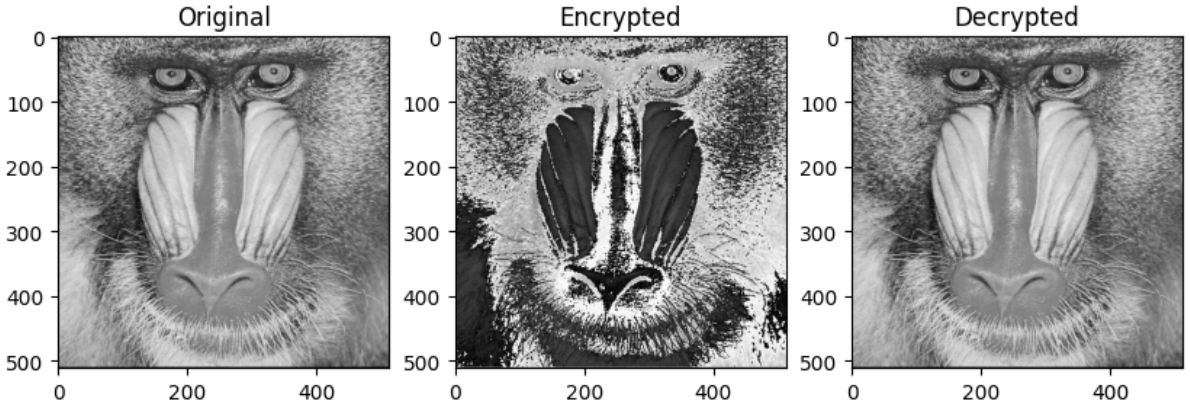
```
def decryption_cypher(img, dec):
    for x in range(0, np.size(img, 0)):
        for y in range(0, np.size(img, 1)):
            img[x][y] = (img[x][y] - dec)%256
    return img
```

Ces deux fonctions prennent en entrée la matrice de l'image et un entier qui est la valeur de décalage que l'on souhaite appliquer.

Voici le résultat obtenu pour lena et baboon avec pour valeur de décalage 128 :



Example 2: CAESAR cipher applied to images – Baboon test image



Analyse :

Dans le cas des deux exemples, on peut très clairement reconnaître l'image qui a été encryptée. Il y a simplement une saturation mais cela n'empêche pas de reconnaître l'image. Le Caesar cypher est plus adapté pour un texte comme c'est un simple changement de caractère. Pour les images, changer du même décalage chaque pixel ne suffit pas à protéger le contenu de l'image. Même pour un texte il est facile de le décrypter.

Example 3: Simple substitution cipher applied to images – Lena test image

A partir de maintenant, nous allons nous intéresser à un type d'encodage à clé. Toujours en considérant en entrée une matrice des coefficients de l'image et d'une clef, nous allons procéder à une substitution. La clef est un vecteur de 256 valeurs, toutes différentes et comprises entre 0 et 255. Pour générer notre image encodée, on associe à

chaque pixel une nouvelle valeur. Cette valeur est celle dans le vecteur clé à la position de la valeur initiale du pixel.

Exemple : (J'épargne au lecteur mes talents de dessinateurs)

Supposons un vecteur clé : [0 254 34 6 … 1]

J'utilise cette clé pour coder mon image, imaginons que le pixel que je regarde ait la valeur 4, alors je remplace sa valeur par la valeur à la quatrième position dans le vecteur, dans notre exemple 6.

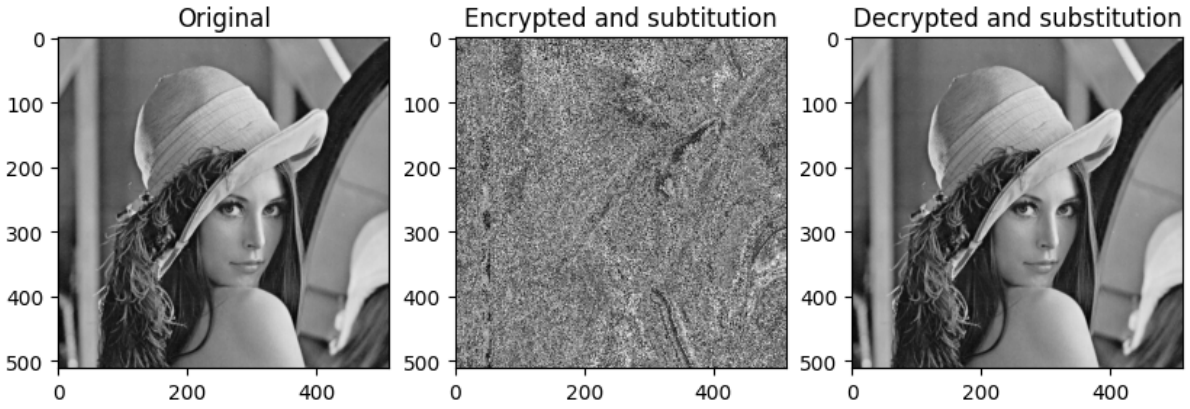
Ce qui nous donne la fonction de génération de clé et de l'image encodée suivantes :

```python
def simplesubstitution(img, key):
    for x in range(0, np.size(img, 0)):
        for y in range(0, np.size(img, 1)):
            img[x][y] = key[img[x][y]]
    return img
```

```python
def genkey():
    key = np.arange(256)
    random.shuffle(key)
    return key
```

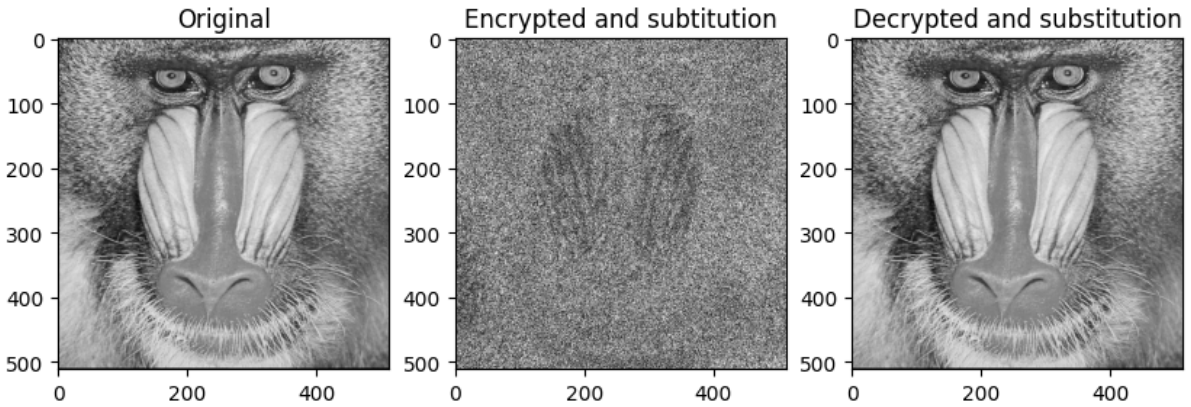
Pour la fonction de décryptement, il suffit de réaliser le même processus. C'est-à-dire qu'avec la clé qui a servi à coder le message, on peut générer une nouvelle clé pour décrypter. Par rapport à l'exemple précédent, on cherche donc une clé qui associe la valeur 4 pour un pixel de valeur 6 afin de récupérer la valeur initiale. On remarque qu'on peut donc générer cette nouvelle clé de manière efficace et ainsi décoder l'image comme ci-dessous :

```python
def decryptsimplesubstitution(img, key):
    key_uncrypt = np.zeros(256)
    for x in range(0, 255):
        key_uncrypt[key[x]] = x
    simplesubstitution(img, key_uncrypt)
    return img
```



Grignoli Marilou
Miens Andréa

Example 4: Simple substitution cipher applied to images – Baboon test image

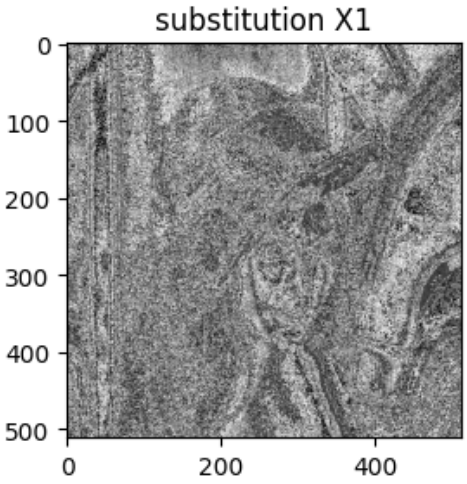


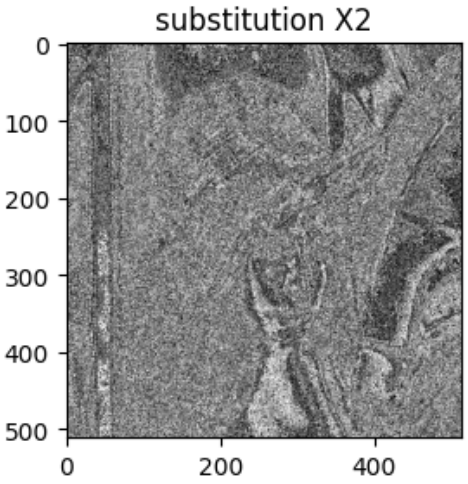
Analyse :

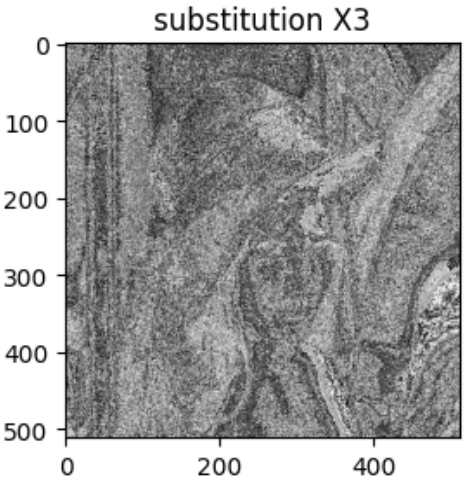
Avec la substitution simple, on obtient un meilleur niveau de confidentialité pour les deux images que dans la première partie. En revanche, on peut encore retrouver les traits marquants des deux images. On observe une différence entre l'impact de la substitution entre lena et baboon : on peut l'expliquer par la fourrure de baboon plus présente, pleine de textures alors que lena présente des zones plus unies. Ces zones très texturées vont ressortir comme plein de points gris qui perdent la structure que possédait initialement l'image. Cette hypothèse semble confirmée avec l'image de lena car la plume de son chapeau n'est plus reconnaissable et il s'agit de la zone la plus texturée alors que la partie haute à droite de son chapeau est très lumineuse et est reconnaissable.

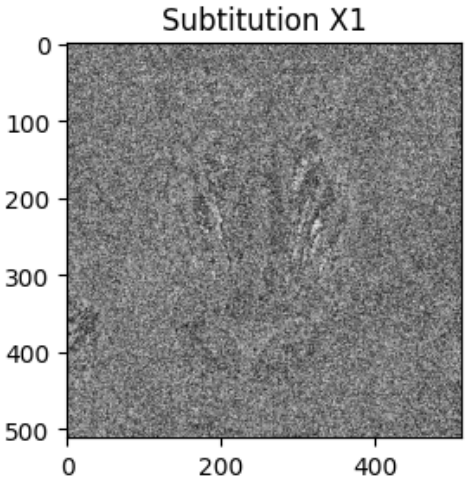
On a essayé d'appliquer la substitution avec la même clé et en changeant de clé à chaque fois.

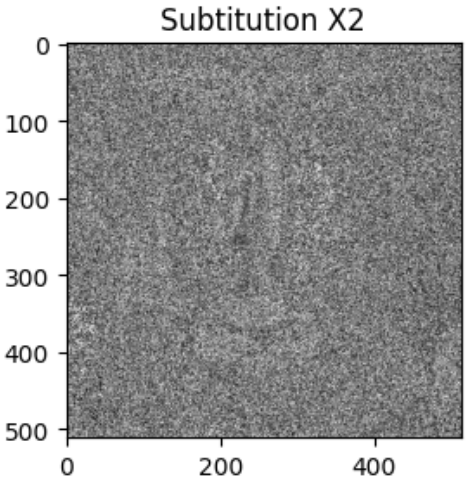
Avec la même clé :

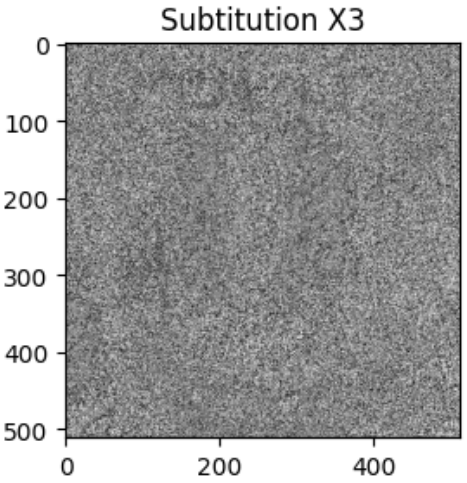




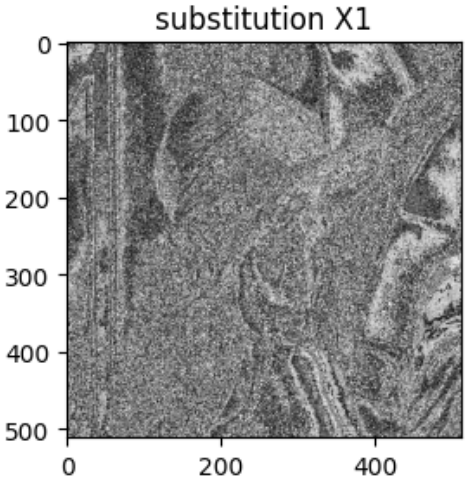


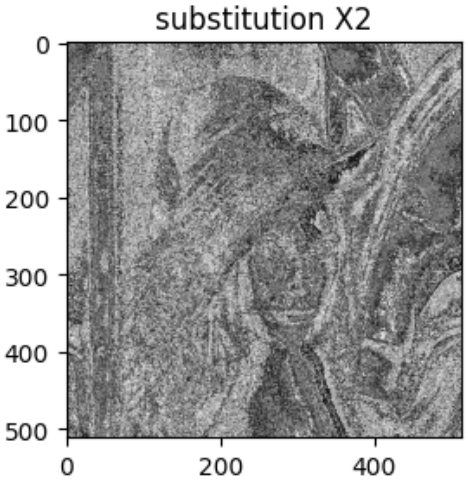


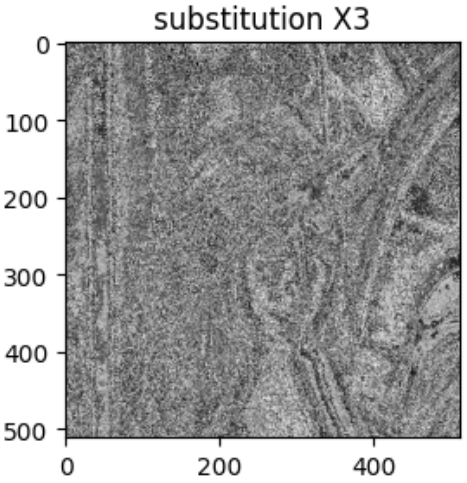


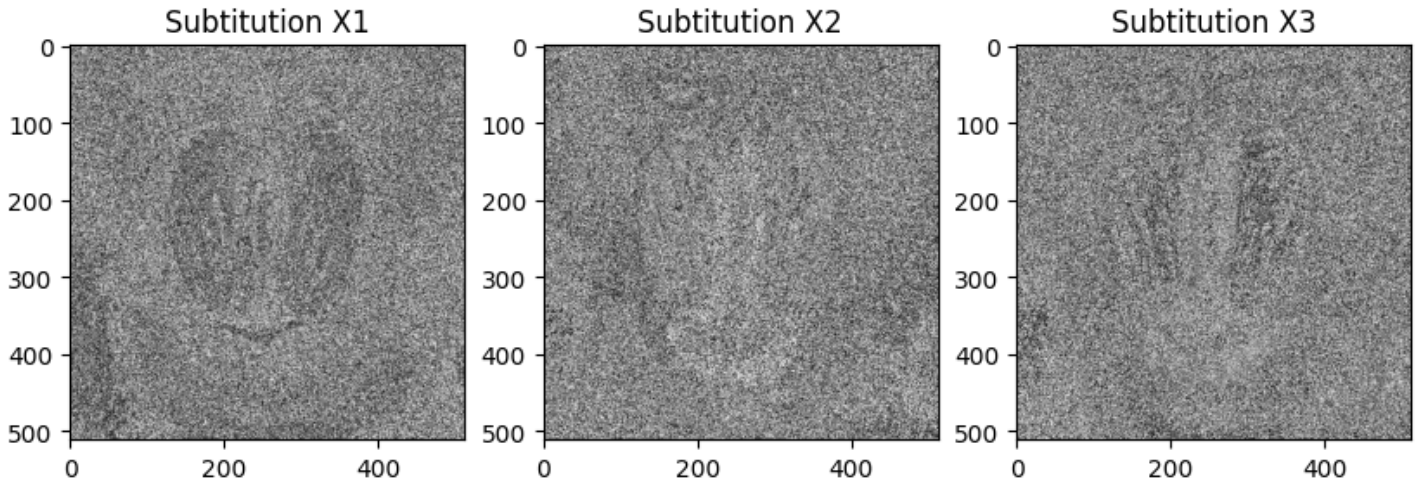


Avec des clés différentes :









On observe donc que l'itération de la substitution n'a pas d'impact sur la protection de l'image, et cela même avec des clés différentes. En effet la substitution n'utilise qu'une clé de 256 valeurs, donc il y a plus de chance pour que les pixels des zones uniformes se retrouvent avec des valeurs peu éloignées.

Example 5: Simple transposition cipher applied to images – Lena test image

A partir de maintenant, on va considérer que chaque pixel va avoir un unique pixel comme image par notre fonction d'encodage. En considérant l'unicité des coordonnées de chaque pixel, on peut générer une clé contenant autant de valeurs que de pixel dans l'image et jouer sur ces coordonnées.
La génération de la clé est la fonction suivante :

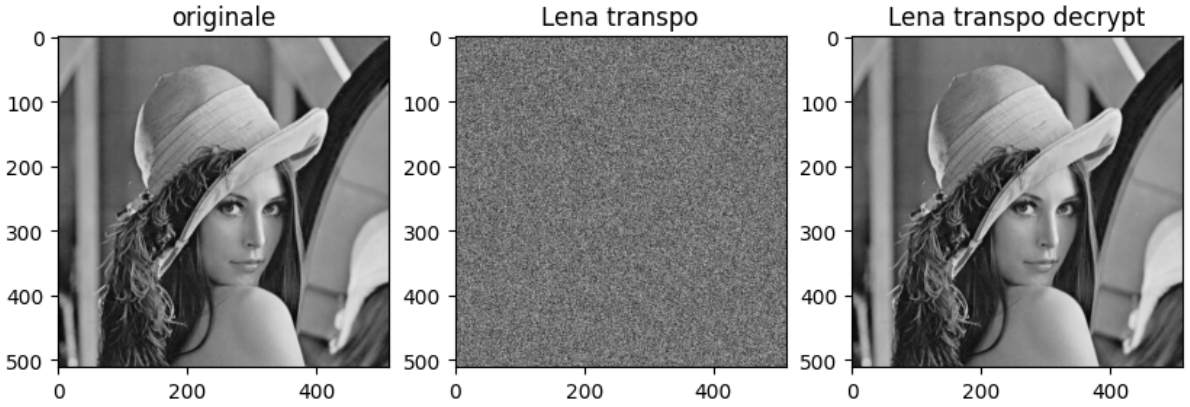
```
def genkeysquare(matim):
    key = np.arange(np.size(matim, 0)*np.size(matim, 1))
    random.shuffle(key)
    return key
```

Là encore, les valeurs étant uniques on peut par une division euclidienne obtenir un couple de coordonnées comme dans la fonction d'encodage suivante :

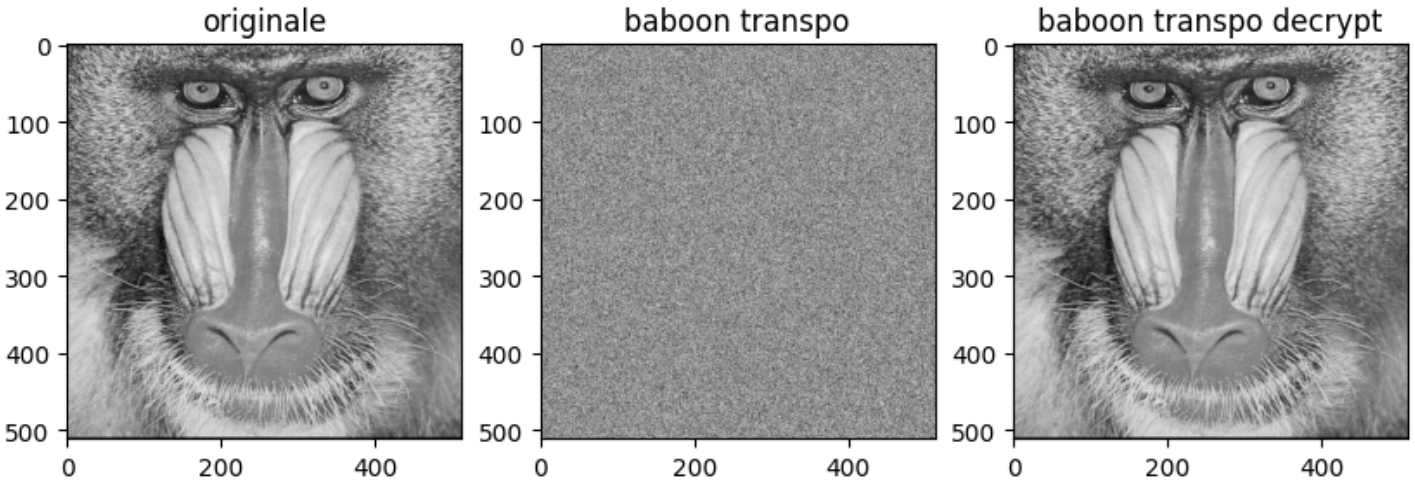
```python
def transposition(img, key):
    lenght = np.size(img, 0)
    width = np.size(img, 1)
    img_new = np.zeros((lenght, width))
    for x in range(0, lenght*width):
        xold = x%lenght
        yold = x//lenght
        xnew = key[x]%lenght
        ynew = key[x]//lenght

        img_new[xnew][ynew] = img[xold, yold]
    return img_new
```

A partir de la même logique que la simple substitution, on peut décoder facilement avec cette fonction :

```python
def decrypttransposition(img, key):
    lenght = np.size(img, 0)
    width = np.size(img, 1)
    img_new = np.zeros((lenght, width))
    for x in range(0, lenght * width):
        xold = key[x] % lenght
        yold = key[x] // lenght
        xnew = x % lenght
        ynew = x // lenght

        img_new[xnew][ynew] = img[xold, yold]

    return img_new
```



Example 6: Simple transposition cipher applied to images – Baboon test image



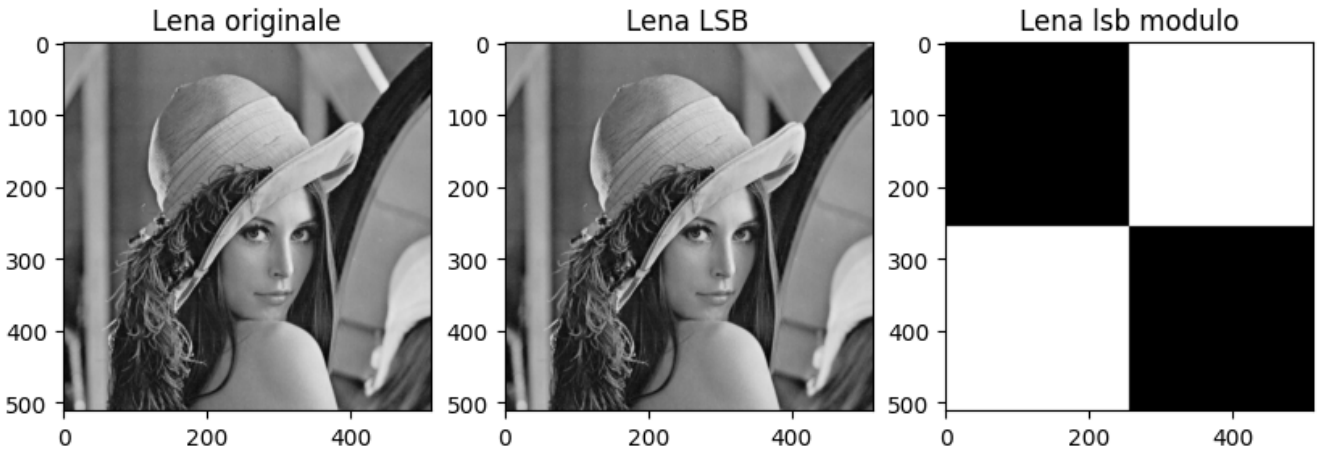
On observe que les deux images ne sont pas reconnaissables une fois qu'on leur applique le simple transposition cypher.

Dans une image, la transposition est très pratique et marche bien pour cacher le contenu de l'image. Dans un texte en revanche on garde la même fréquence d'apparition pour une lettre donc

Example 7: LSB (lowest significant bit) steganography – visual impact on Lena test image

Nous passons maintenant à la mise en place de la méthode de LSB (ie. lowest significant bit). L'idée est de choisir la valeur du bit le moins significatif en fonction de son emplacement dans l'image. Le code ci-dessous va nous permettre découper notre image en 4 zones, les zones en haut à gauche et en bas à droite verront leur LSB mis à 0 et les deux autres zones à 1. Ceci va donc forcer les zones à 0 d'avoir une valeur de luminance paire et les autres zones une valeur impaire. Ce code nous permet de réaliser les différentes zones en même temps pour éviter trop de boucles et de tests.

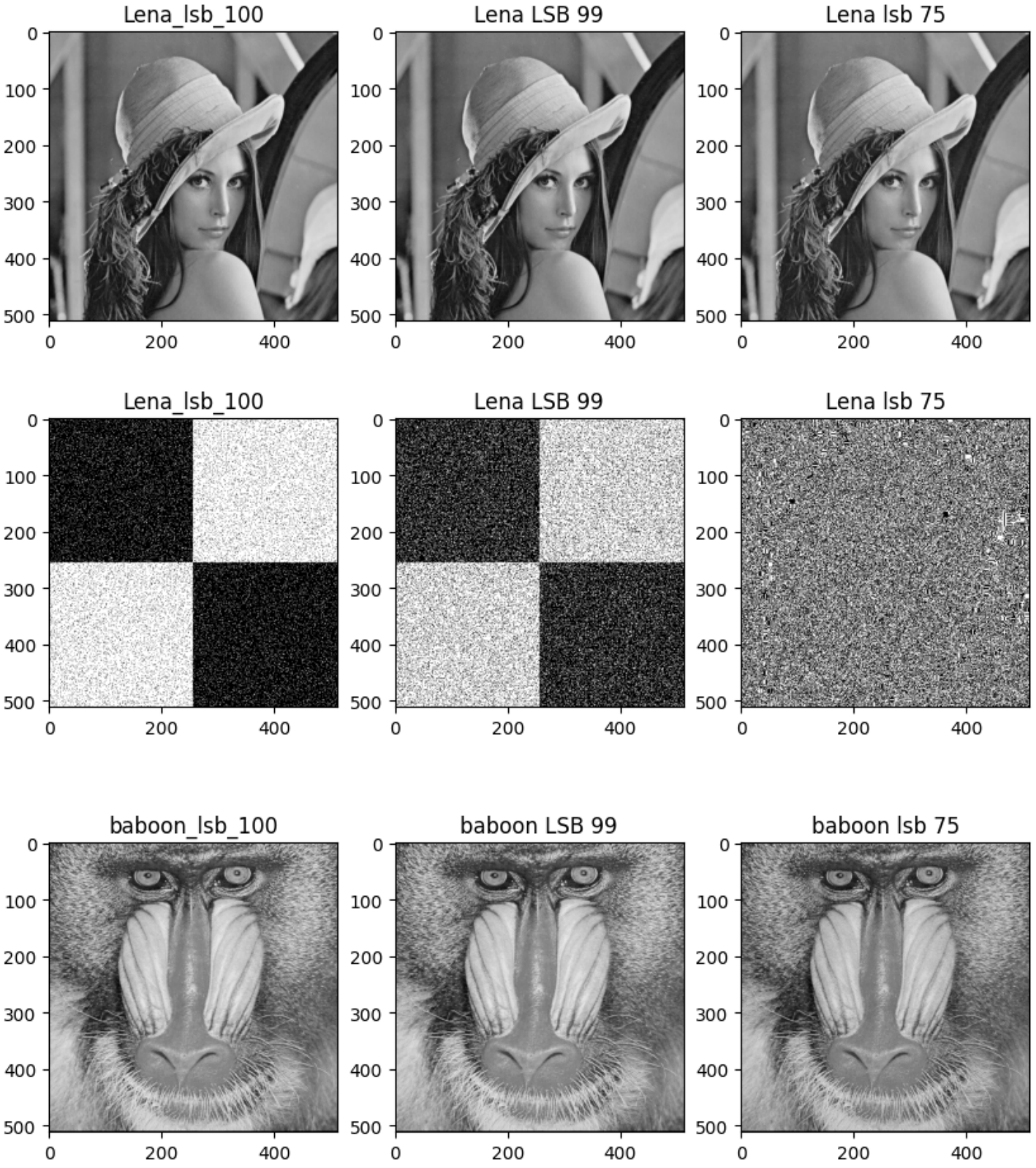
```python
def lsb_conversion(img):
    lenght = np.size(img, 0)
    width = np.size(img, 1)
    medium_l = lenght//2
    medium_w = width//2
    img_new = np.zeros((lenght, width))
    for x in range(0, lenght):
        if x <= medium_l:
            for y in range(0, medium_w):
                img_new[x][y] = img[x][y] - img[x][y]%2
                img_new[x][medium_w+y] = img[x][medium_w+y] - img[x][medium_w+y]%2 + 1
        else:
             for y in range(0, medium_w):
                 img_new[x][y] = img[x][y] - img[x][y] % 2 + 1
                 img_new[x][medium_w+y] = img[x][medium_w+y] - img[x][medium_w+y] % 2
    return img_new
```

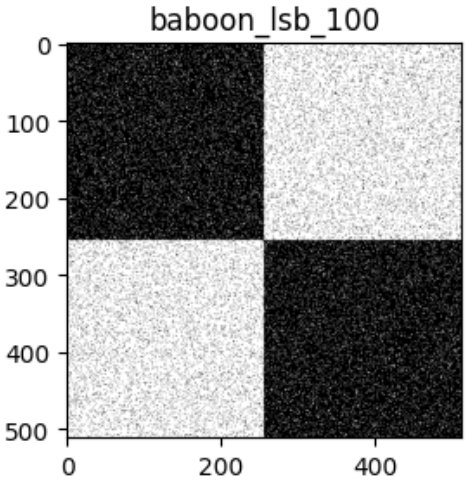


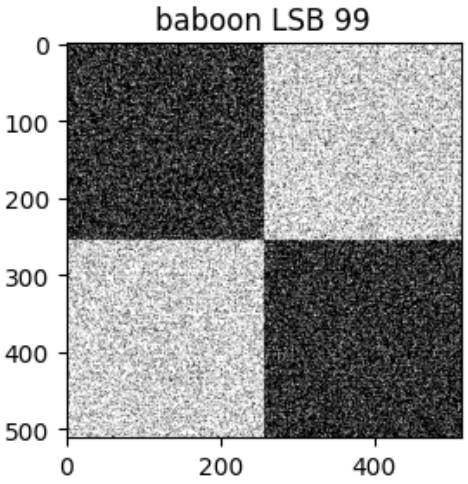
On observe qu'à l'œil nu il est impossible de voir la différence entre lena et lena LSB. Pour vérifier que nous avons bien appliquer le lsb à notre image on

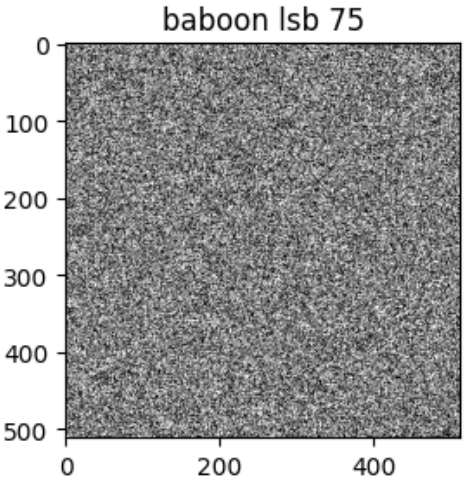
fait un modulo 2 et on peut voir les zones dont le lsb a été mis à 1 et celles où il a été mis à 0.

Example 8: LSB (lowest significant bit) steganography – resilience against jpg compression for the Lena test image

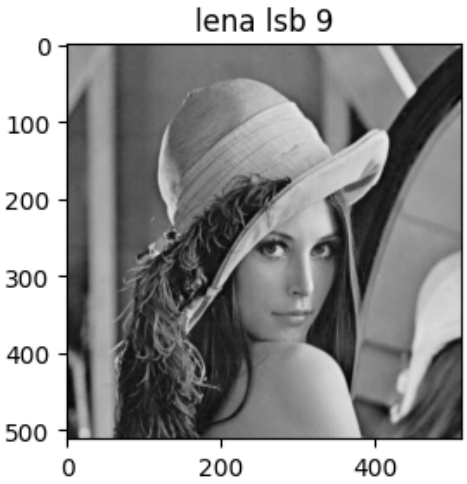


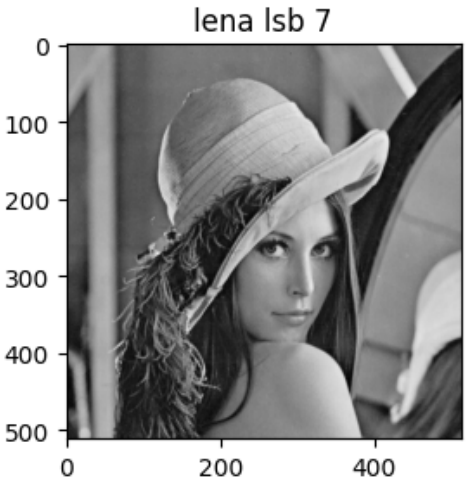


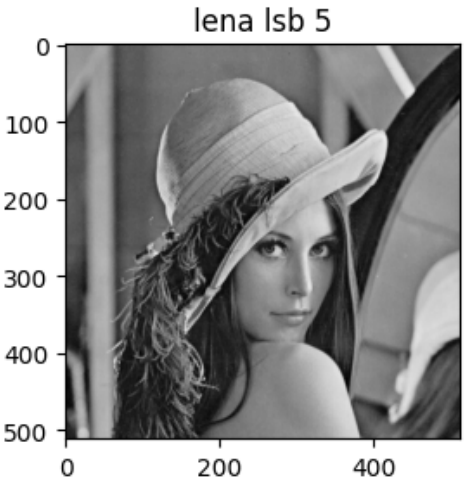


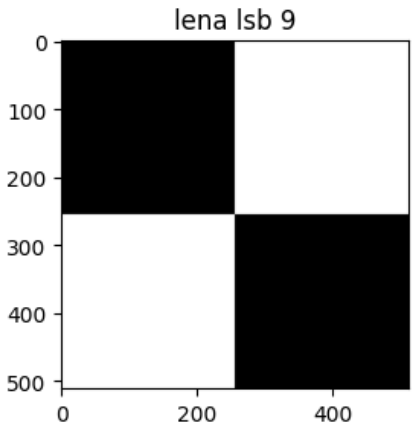


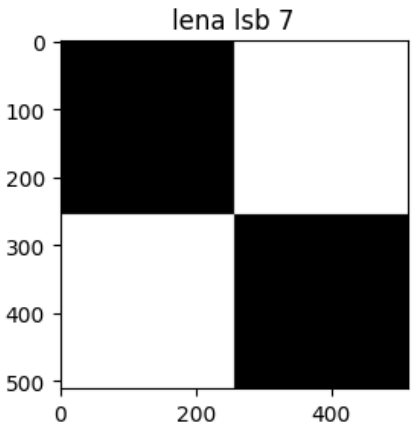
Pour le format JPEG : Le bit de moindre poids n'est pas une technique robuste face au format JPEG. On observe une disparition progressive de notre damier au fur et à mesure que le niveau de qualité diminue. En dessous d'une certaine qualité de compression, on ne peut plus reconnaître le damier que nous avions initialement ajouté.

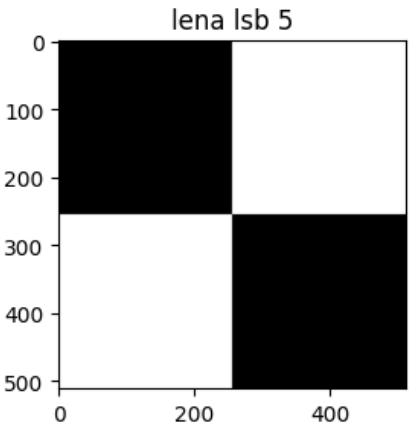








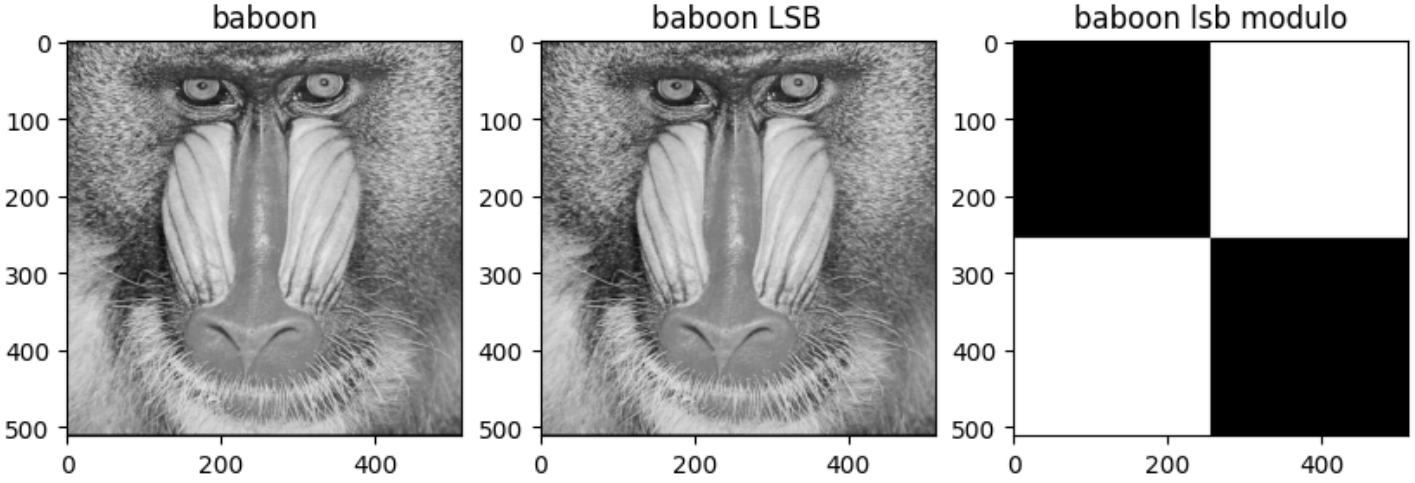




Pour le format PNG : PNG est un algorithme de compression sans perte. À un niveau de compression de 9, il est très rapide avec un fichier de petite taille, mais la taille du fichier reste assez grande. À un niveau de compression de 5, le fichier est compressé, mais le temps de calcul est long et nécessite suffisamment de puissance de calcul.

Conclusion de l'exemple :
Bien que visuellement, le format .jpg impacte peu notre vision de l'image comme c'est une méthode avec perte, on perd de l'information et elle impacte le bit de poids faible. C'est pourquoi cela impacte la reconstruction du damier. En compressant notre image avec le format PNG, qui est une méthode sans perte, on peut récupérer notre damier sans aucun bruit parasite.

Example 9: LSB (lowest significant bit) steganography – visual impact and resilience against jpg compression for the Baboon test image



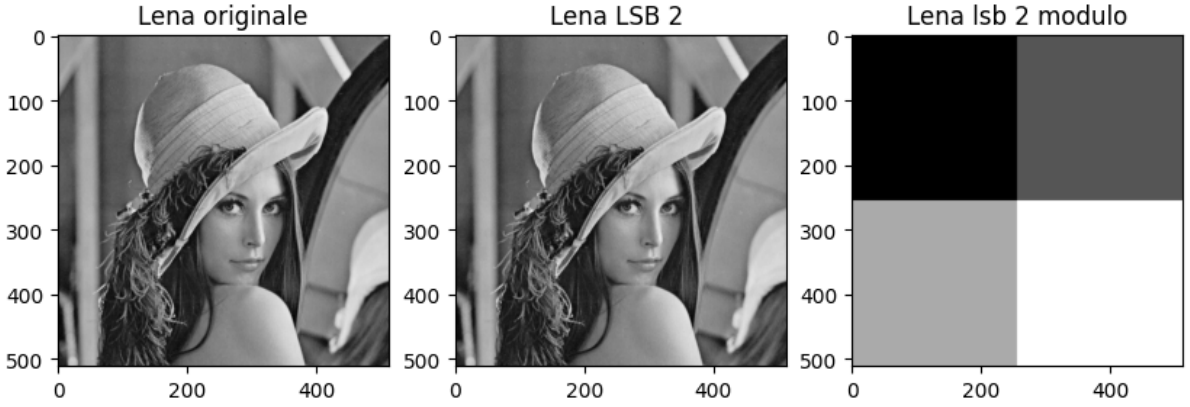
Example 10: 2bit - LSB (lowest significant bit) steganography – visual impact and resilience against jpg compression

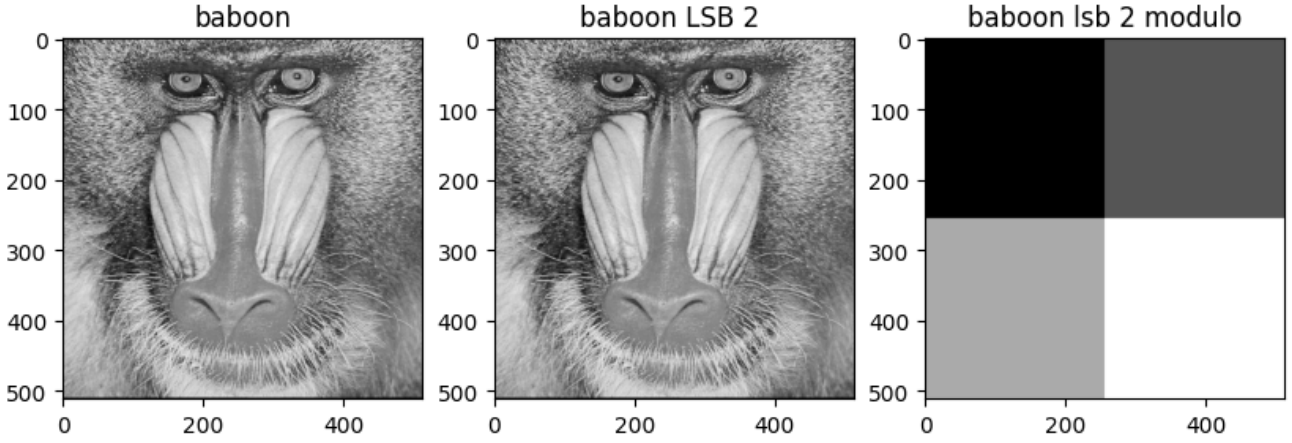
Avec la même idée que dans le cas LSB, on va jouer sur la valeur des 2 bits de poids faibles. On va encore découper notre image encore une fois en 4 zones. Dans chaque zone, on a :

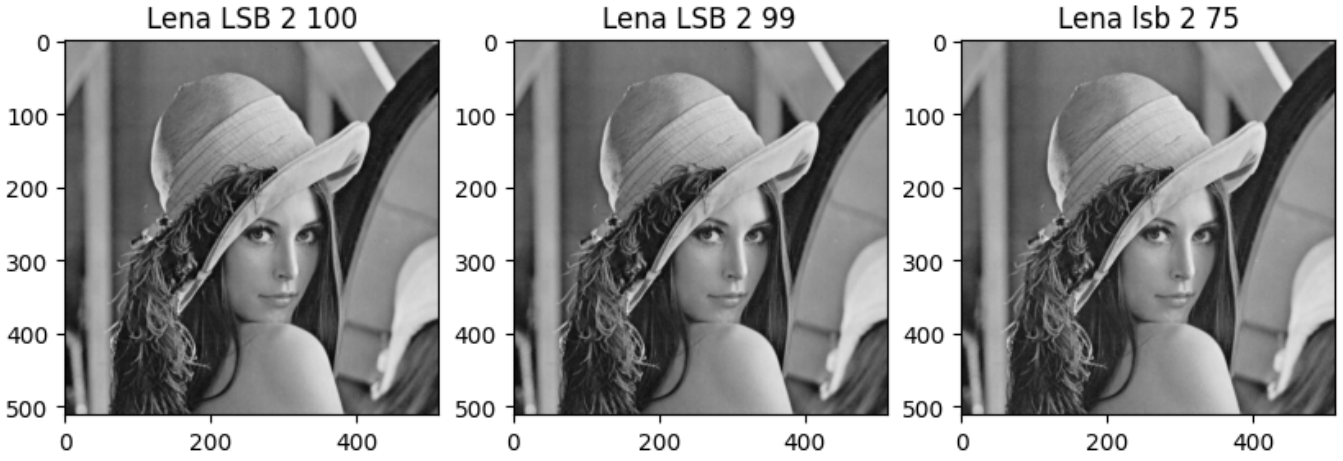
- zone en haut à gauche : les deux bits de poids faibles à 0
- zone en haut à droite : on aura 01 (ie. le bit de poids le plus faible à 1 et l'autre à 0)
- zone en bas à droite : on aura 10 (ie. le bit de poids le plus faible à 0 et l'autre à 1)

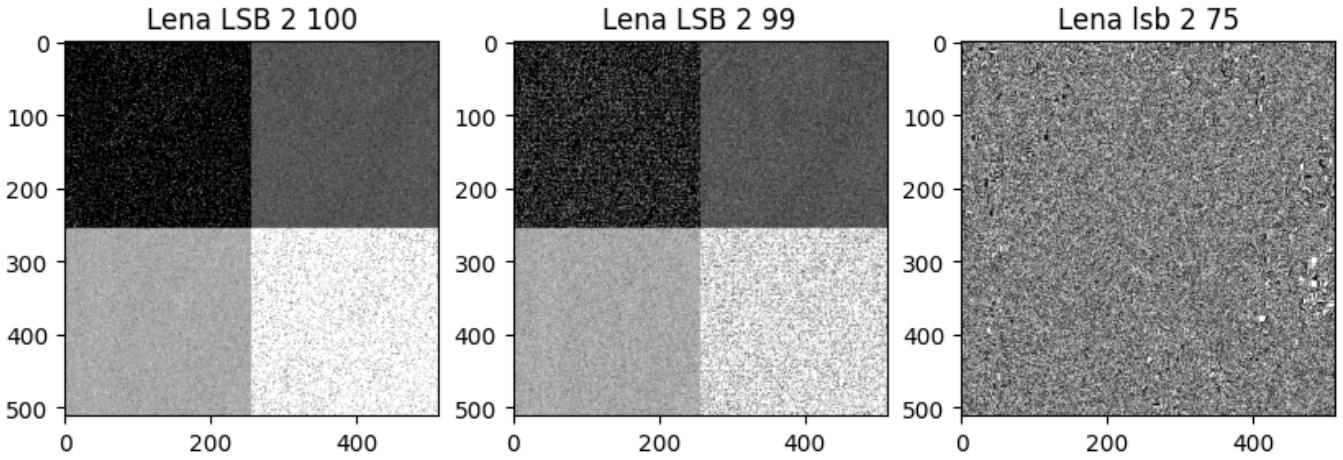
- zone en bas à gauche : on aura 11 (ie. les deux derniers bits prendront la valeur 1)

```python
def lsb2_conversion(img):
    lenght = np.size(img, 0)
    width = np.size(img, 1)
    medium_l = lenght // 2
    medium_w = width // 2
    img_new = np.zeros((lenght, width))
    for x in range(0, lenght):
        if x <= medium_l:
            for y in range(0, medium_w):
                img_new[x][y] = img[x][y] - img[x][y] % 4
                img_new[x][medium_w+y] = img[x][medium_w+y] - img[x][medium_w+y]%4 + 1
        else:
            for y in range(0, medium_w):
                img_new[x][y] = img[x][y] - img[x][y] % 4 + 2
                img_new[x][medium_w+y] = img[x][medium_w+y] - img[x][medium_w+y] % 4 + 3
    return img_new
```






Lena LSB 2 100
Lena LSB 2 99
Lena lsb 2 75
0
100
200
300
400
500
0
200
400


Lena LSB 2 100
Lena LSB 2 99
Lena lsb 2 75
0
100
200
300
400
500
0
200
400

Conclusion Générale

Au cours de ce TP nous avons implémenté et observé différentes méthodes de codage d'image. Voici les points de conclusion :

1) Le Caesar cipher n'apporte pas une bonne protection du contenu de l'image. En aucun cas il nous est impossible de ne pas reconnaître l'image.
2) Avec la substitution, notre image n'est encore une fois pas du tout protégée.
3) Si on itère la substitution, cela ne change pas beaucoup l'image encodée qui reste reconnaissable. Malgré cette répétition et des clés différentes, on peut toujours voir apparaître des traits propres à l'image, elle laisse donc des informations à notre image.
4) La différence entre un texte et une image en stéganographie réside dans le fait que pour un texte, une perte d'information provoque une illisibilité du message alors que pour une image on peut quand même reconnaître la forme globale insérée. En effet, dans un texte assez long il est impossible de deviner le sens alors que pour une image on peut toujours reconnaître partiellement l'image.
5) La méthode LSB ne permet pas de cacher suffisamment un message. Si les compressions appliquées aux images sont sans perte alors notre message restera intact. Malheureusement, dès l'or que l'on applique des compressions avec perte comme le JPEG notre message s'en trouve impacté et peut devenir illisible.