Candidat N°21234

AGBETI-MESSAN Epe-Nsin Merveilles

TIPE :
La cryptographie des images

Encadré par : Professeur ESSANHAJI Abdelhak

Plan de Présentation

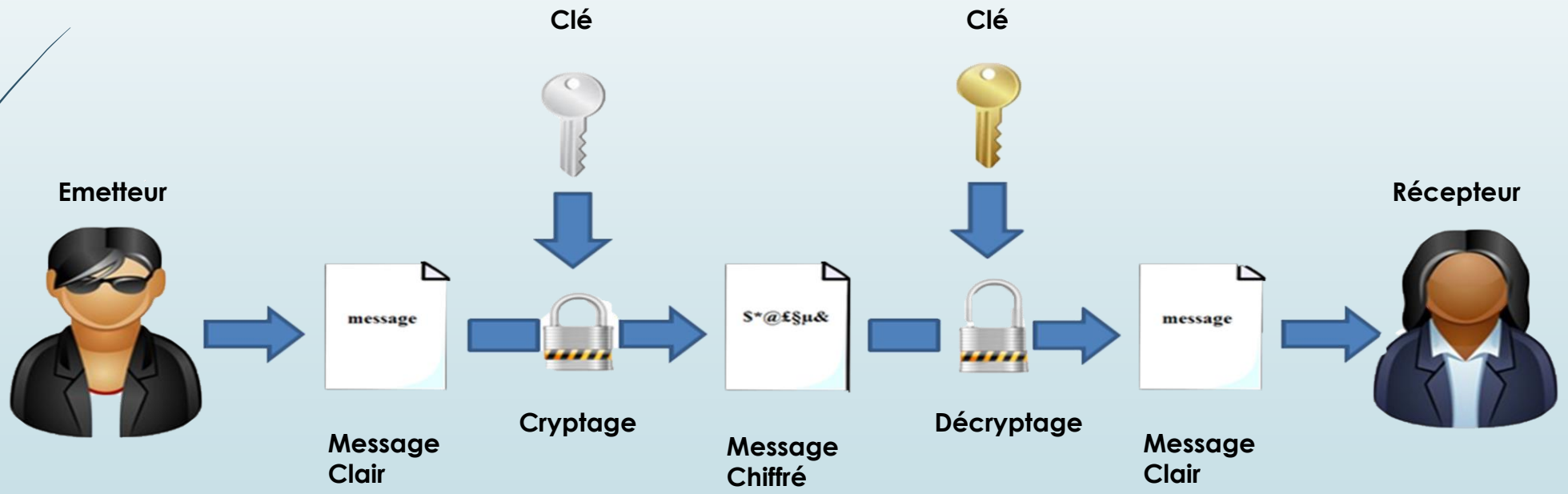
- Introduction
- Quelques Concepts de base en Cryptographie
- L'Image Numérique
- Présentation de deux permutations classiques de plan pour rendre une image inintelligible
- Présentation d'une nouvelle permutation créée grâce à une suite chaotique
- Analyse des résultats et comparaisons
- Conclusion

Introduction

Nécessité d'établir ainsi une protection vis-à-vis de la lisibilité à travers des canaux non sécurisés. Ceci constituera donc l'objectif de notre travail.

Quelques Concepts de base en Cryptographie

Cryptage, Transformation et Clé



Quelques Concepts de base en Cryptographie

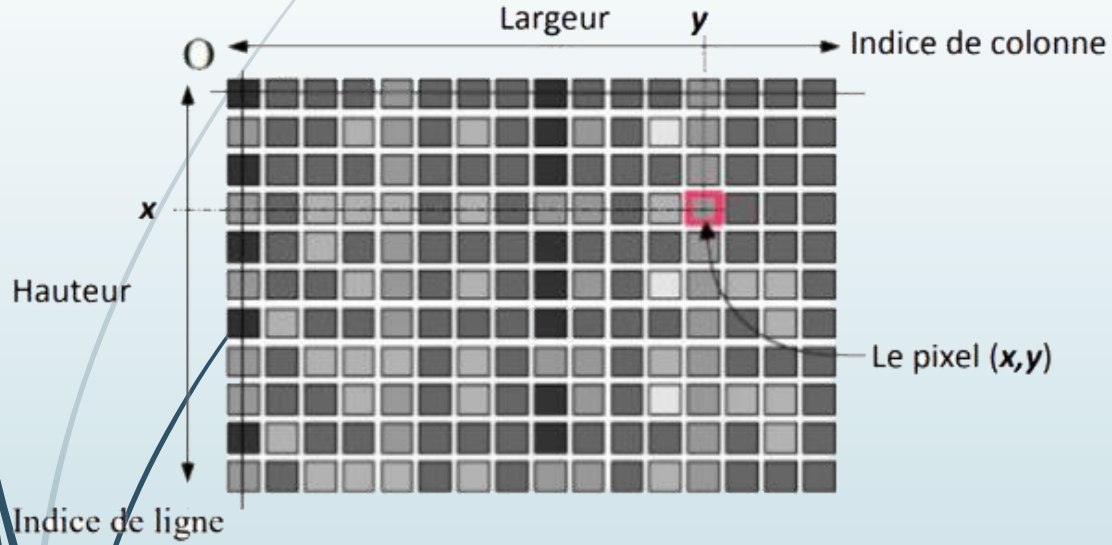
Chiffrement symétrique et asymétrique

- **Le chiffrement symétrique** : Utilise une seule clé pour chiffrer et déchiffrer les données. Vous devez partager cette clé avec le destinataire.
- **Le chiffrement asymétrique** : Nécessite deux clés pour fonctionner, une clé publique doit être rendue publique afin de chiffrer les donnée et une autre, secrète, utilisée pour décrypter les données.

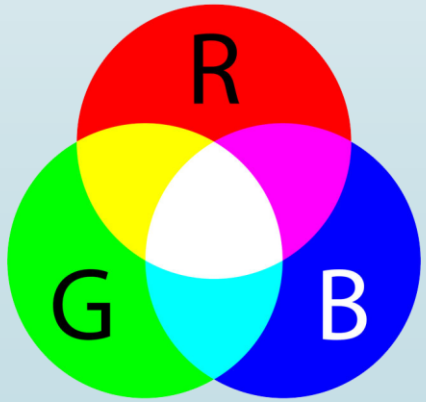
La cryptographie rassemble les principes, les moyens et les méthodes de transformation des données. Dans la nouvelle méthode proposée, nous allons nous intéresser au chiffrement symétrique des images.

L'Image Numérique

Qu'est ce qu'une image numérique et représentation d'une image numérique



Les images numériques sont constituées d'un ensemble de pixels juxtaposés en lignes et en colonnes formant ainsi une matrice. Le pixel est le plus petit élément que l'on peut trouver dans une image.



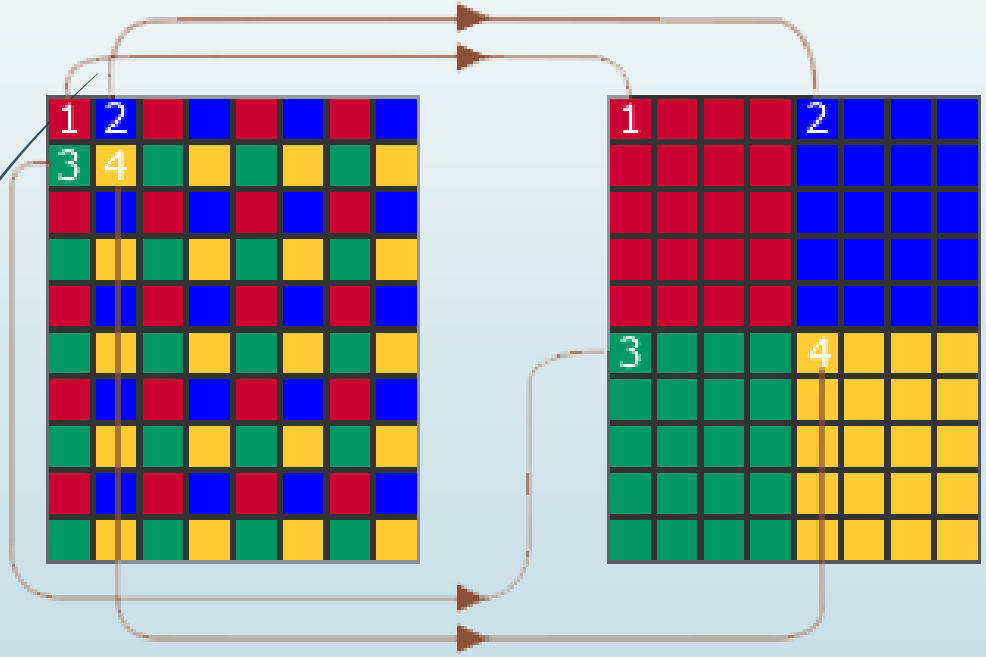
Présentation de deux permutations classiques de plan pour rendre une image inintelligible

Nous allons donc voir l'utilisation des permutations du plan afin de brouiller une image.

- La transformation du **photomaton** qui consiste à réduire l'image en la dupliquant en 4.
- La transformation du **boulanger** qui consiste à étirer l'image puis à la diviser en deux et ensuite à les racoler.

Présentation de deux permutations classiques de plan pour rendre une image inintelligible

Permutation du Photomaton



Pour une image de taille **m*n** avec n et m pair, la nouvelle position (X, Y) peuvent être calculer de cette façon en fonction de l'ancienne (x, y):

$$X = \begin{cases} \frac{x}{2} \; si \; x \; pair \\ E\left(\frac{x}{2}\right) + \frac{m}{2} \; si \; x \; est \; impair \end{cases}$$

$$Y = \begin{cases} \frac{y}{2} \; si \; y \; est \; pair \\ E\left(\frac{y}{2}\right) + \frac{n}{2} \; si \; y \; est \; impair \end{cases}$$

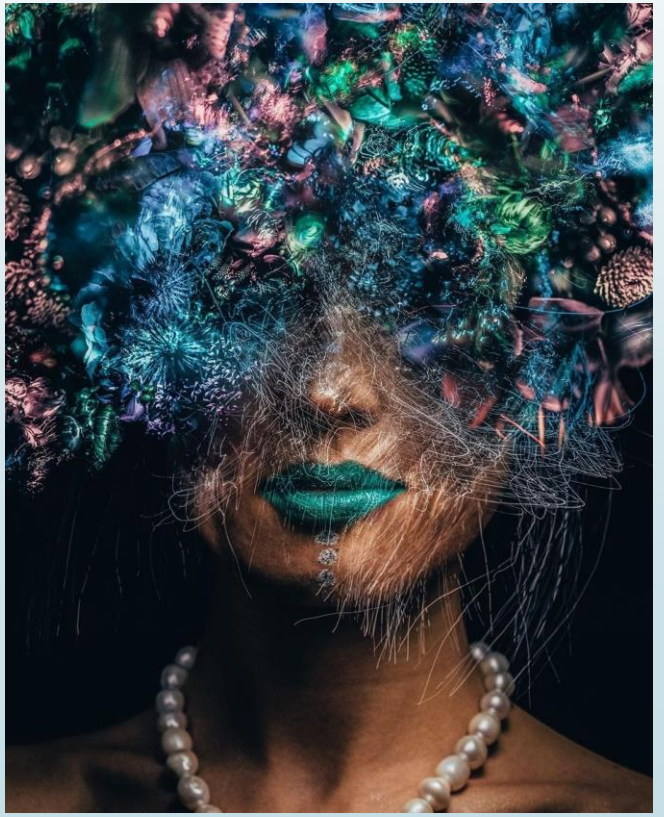
Présentation de deux permutations classiques de plan pour rendre une image inintelligible

Permutation du Photomaton

Code python du Photomaton

```python
def photomaton(x, hauteur, y, largeur):
    """ Calcul des nouvelles positions grace au principe
    du photomaton """
    if x%2 == 0:
        x = x//2
    else:
        x = x//2 + hauteur//2
    if y%2 == 0:
        y = y//2
    else:
        y = y//2 + largeur//2
    return x, y

```

Image Originale



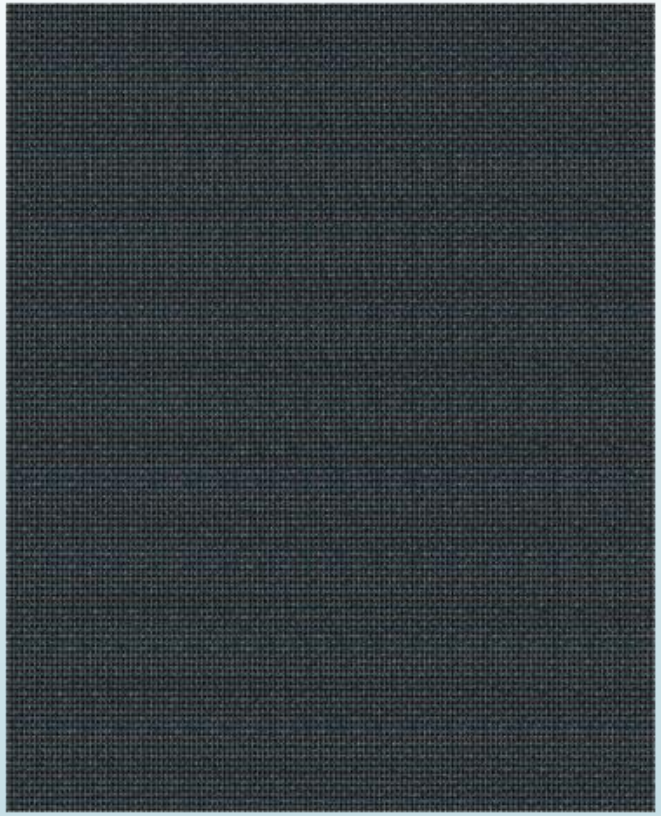
Présentation de deux permutations classiques de plan pour rendre une image inintelligible

Résultats de la Permutation au Photomaton

IMAGE APRES 1 PERMUTATION

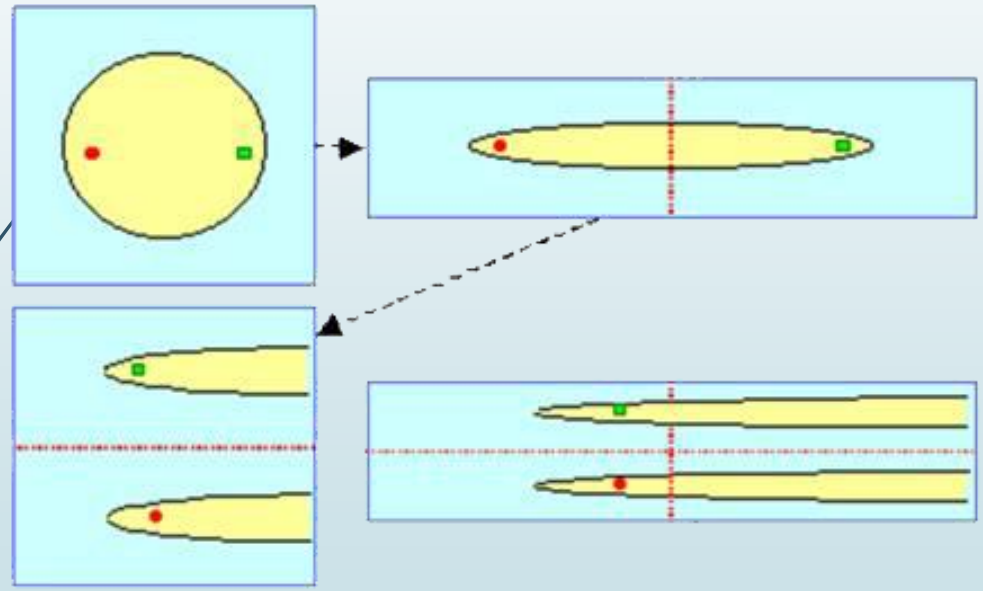
IMAGE APRES 3 PERMUTATIONS

IMAGE APRES 7 PERMUTATIONS



Présentation de deux permutations classiques de plan pour rendre une image inintelligible

Permutation du Boulanger



Pour une image de taille **m*n** avec m et n pair, la nouvelle position (X, Y) peuvent être calculer par ces deux étapes en fonction de l'ancienne (x, y):

$$(x1, y1) = \begin{cases} (\frac{x}{2}, 2 * y) \; si \; x \; pair \\ (E\left(\frac{x}{2}\right), 2 * y + 1) \; si \; x \; est \; impair \end{cases}$$

$$(X, Y) = \begin{cases} (x1, y1) \; si \; y1 < n \\ (m - 1 - x1, 2n - 1 - y1) \; si \; y1 \geq n \end{cases}$$

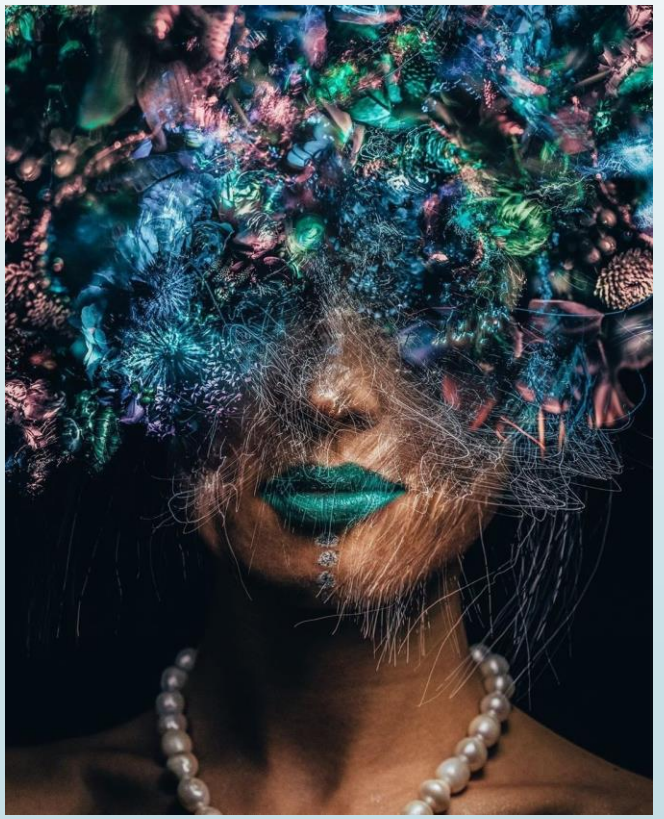
Présentation de deux permutations classiques de plan pour rendre une image inintelligible

Permutation du Photomaton

Code python du Boulanger

```python
def boulanger(x, hauteur, y, largeur):
    """ Calcul des nouvelles positions grace au principe
    du boulanger """
    if x%2 == 0:
        x, y = x//2, 2*y
    else:
        x, y = x//2, 2*y+1
    if y < largeur:
        return x, y
    else:
        return hauteur - 1 - x, 2*largeur - 1 - y

```

Image Originale



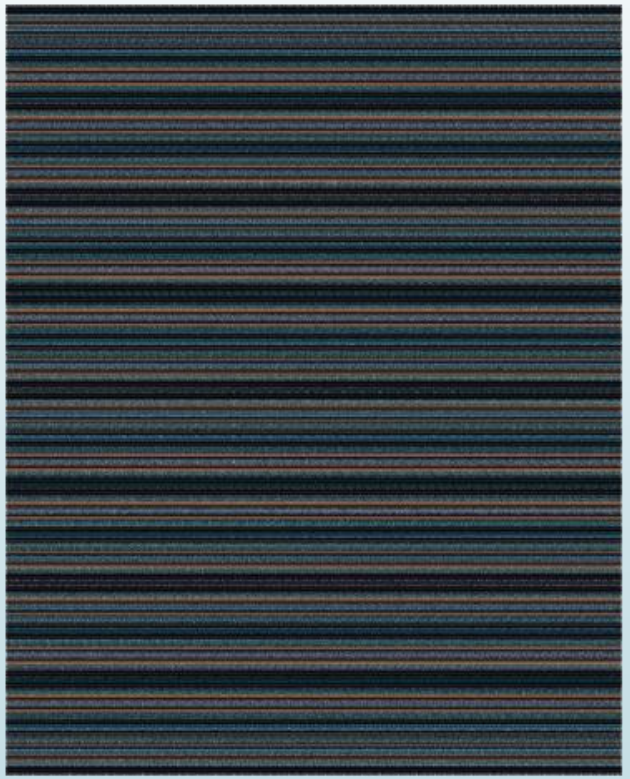
Présentation de deux permutations classiques de plan pour rendre une image inintelligible

Résultats de la Permutation au Boulanger

IMAGE APRES 1 PERMUTATION

IMAGE APRES 3 PERMUTATIONS

IMAGE APRES 7 PERMUTATIONS



Présentation d'une nouvelle permutation créée grâce à une suite chaotique

Suite Chaotique et Permutation

- Les deux méthodes précédentes ont pour point faibles d'avoir une taille pair et de redonner l'image originale après plusieurs permutations en cas d'égalité des dimensions.
- Nous allons donc créer une nouvelle permutation grâce à une suite logistique à comportement chaotique.

La suite de Robert May

$X_{n+1} = u*Xn(1-Xn)$

Cette suite donne un comportement chaotique pour $u \geq 3.57$. On prendra donc comme **clé de cryptage**, le couple (u, X_0) avec $u \geq 3.57$.

```python
def suiteRobertMay(u, x0, n):
    #Creer n valeurs aleatoires a partir de la valeur 6
    for i in range(5):
        x0 = u*x0*(1-x0)
    listeValeurs = []
    for i in range(1, n+1):
        listeValeurs.append(x0)
        x0 = u*x0*(1-x0)
    return listeValeurs

def creationDePermutation(u, x0, n):
    #Creer une permutation de n valeurs a partir de la suite
    listeValeurs = suiteRobertMay(u, x0, n)
    listeTriee = sorted(listeValeurs)
    permutation = [listeValeurs.index(i) for i in listeTriee]
    return permutation

```

Présentation d'une nouvelle permutation créée grâce à une suite chaotique

Suite Chaotique, Permutation et Sensibilité

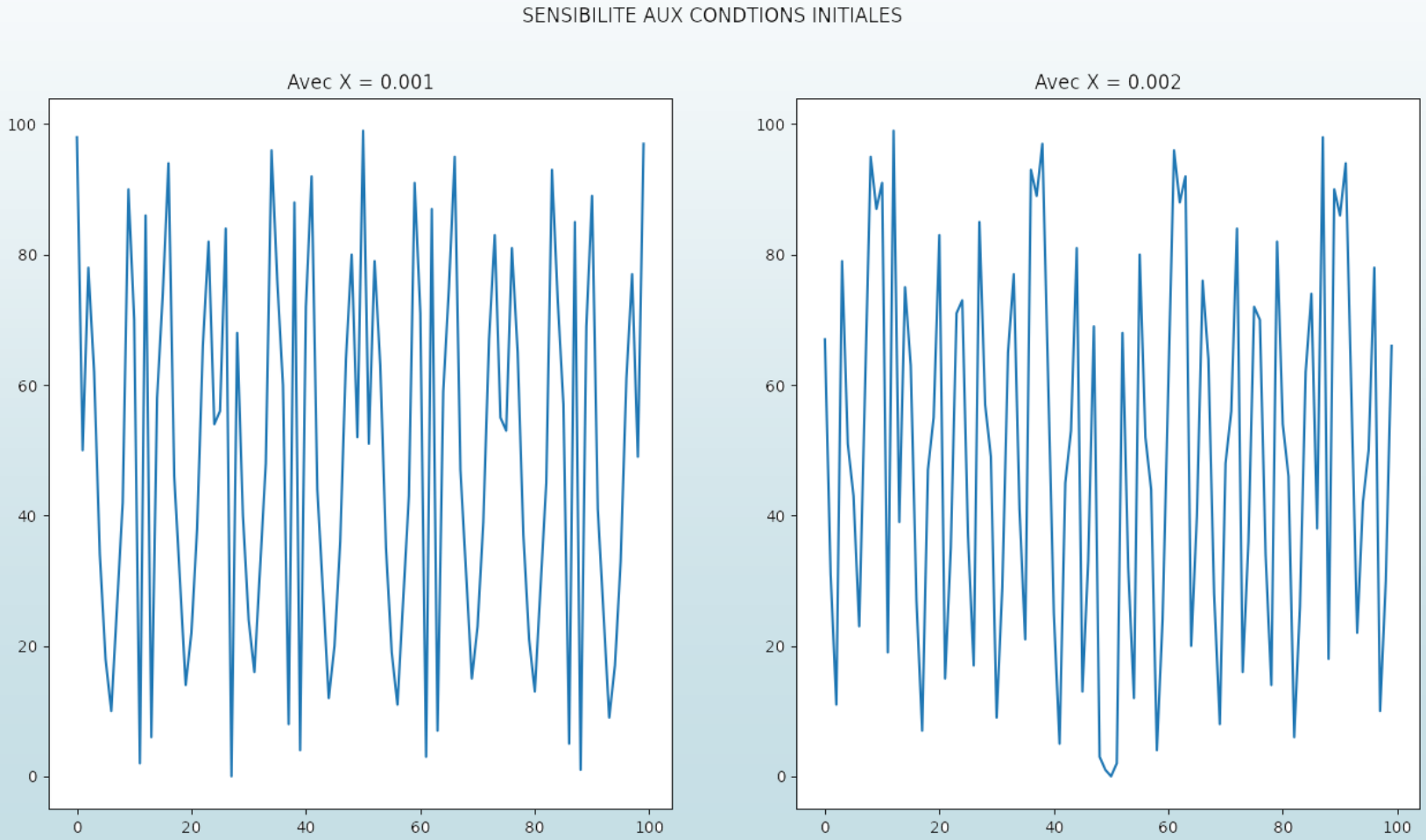
- Principe de création de la permutation :
  - Créer la liste L1 de n valeurs aléatoires
  - Créer une copie L2 de la liste L1 et la trier par ordre croissant
  - Créer une nouvelle liste L3 qui contient les positions des éléments de la liste L1 dans la liste L2. Il s'agit donc de la permutation.
- Cette permutation créer avec la suite Robert May est très sensible aux conditions initiales. Ce qui la rend meilleure pour un bon chiffrement. Exemple avec

$$u = 3.58$$
$$x0 = 0.001$$
$$x1 = 0.002$$

```python
def verificationSensibilite(u, x0, x1, n):
    #Sensibilite aux conditions initiales
    abcisses = [i for i in range(n)]
    permutation1 = creationDePermutation(u, x0, n)
    permutation2 = creationDePermutation(u, x1, n)

    plt.subplot(1, 2, 1)
    plt.plot(abcisses, permutation1)
    plt.title("Avec X = {}".format(x0))

    plt.subplot(1, 2, 2)
    plt.plot(abcisses, permutation2)
    plt.title("Avec X = {}".format(x1))

    plt.suptitle("SENSIBILITE AUX CONDTIONS INITIALES")
    plt.show()
```

Présentation d'une nouvelle permutation créée grâce à une suite chaotique

Résultat du test de sensibilité



Présentation d'une nouvelle permutation créée grâce à une suite chaotique

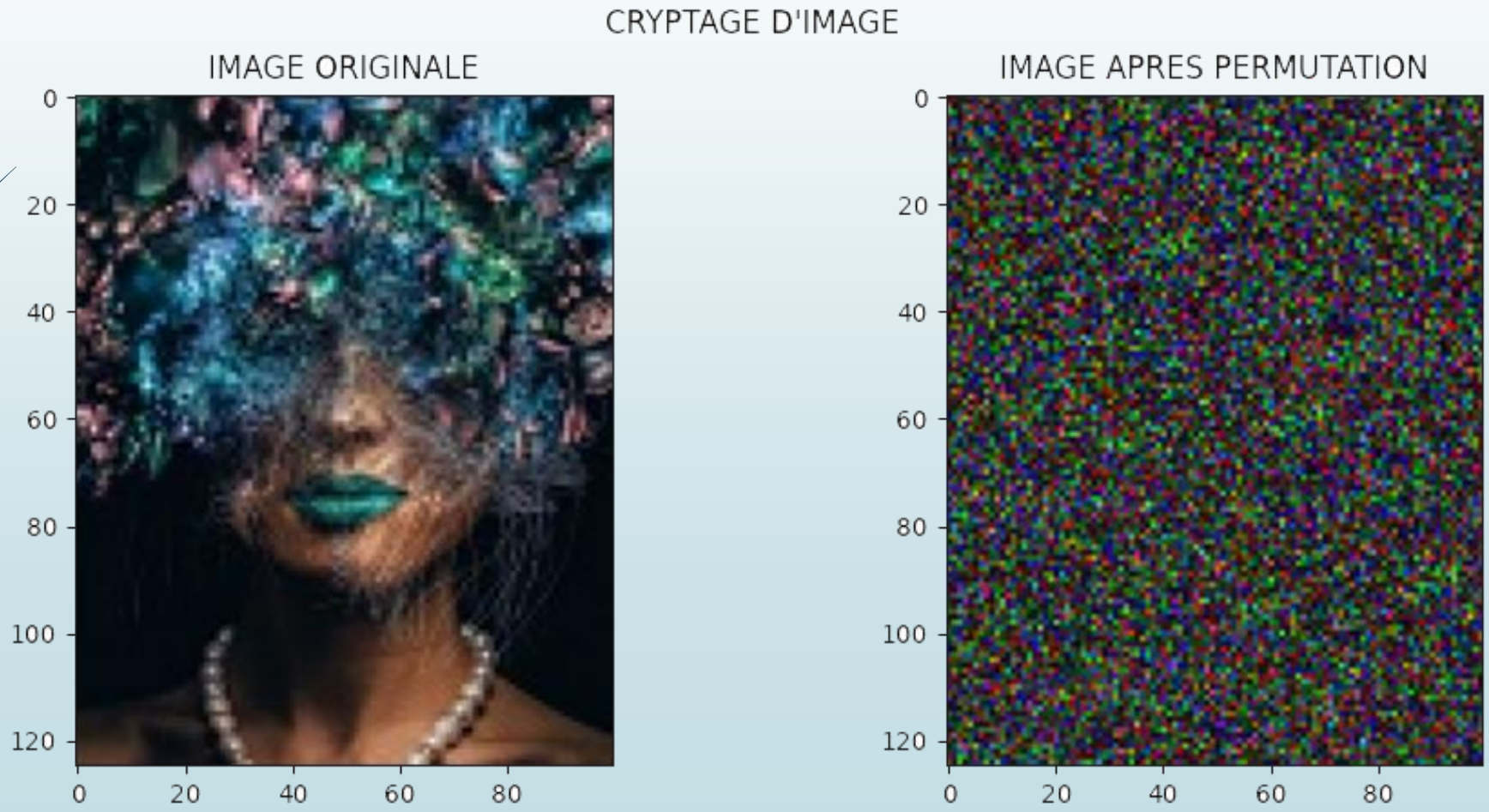
Permutation de toutes les valeurs de pixels de l'image

- Pour une image de format **m*n**, chaque pixel contient 3 valeurs RGB, il faut donc une permutation de **m*n*3** valeurs.
- Et calculer ensuite les positions (x, y, z) de chaque valeur de pixel.
- La nouvelle image est renvoyée par la fonction.

```python
def calculPositionToutImage(nombre, largeur):
    #Calcul x, y, z a partir d'une valeur "nombre"
    x = nombre // (largeur * 3)
    nombre = nombre - largeur * 3 * x
    y = nombre // 3
    z = nombre % 3
    return x, y, z

def cryptageToutImage(image, u, x0):
    #Retourne l'image apres une permuation des valeurs des pixels
    nouvelleImage = zeros_like(image)

    hauteur = image.shape[0]
    largeur = image.shape[1]

    f = creationDePermutation(u, x0, hauteur*largeur*3)

    for nombre in range(hauteur*largeur*3):
        x, y, z = calculPositionToutImage(nombre, largeur)
        nouveauX, nouveauY, nouveauZ = calculPositionToutImage(f[nombre], largeur)
        nouvelleImage[x][y][z] = image[nouveauX][nouveauY][nouveauZ]

    return nouvelleImage

```

Présentation d'une nouvelle permutation créée grâce à une suite chaotique

Résultat sur une image



Présentation d'une nouvelle permutation créée grâce à une suite chaotique

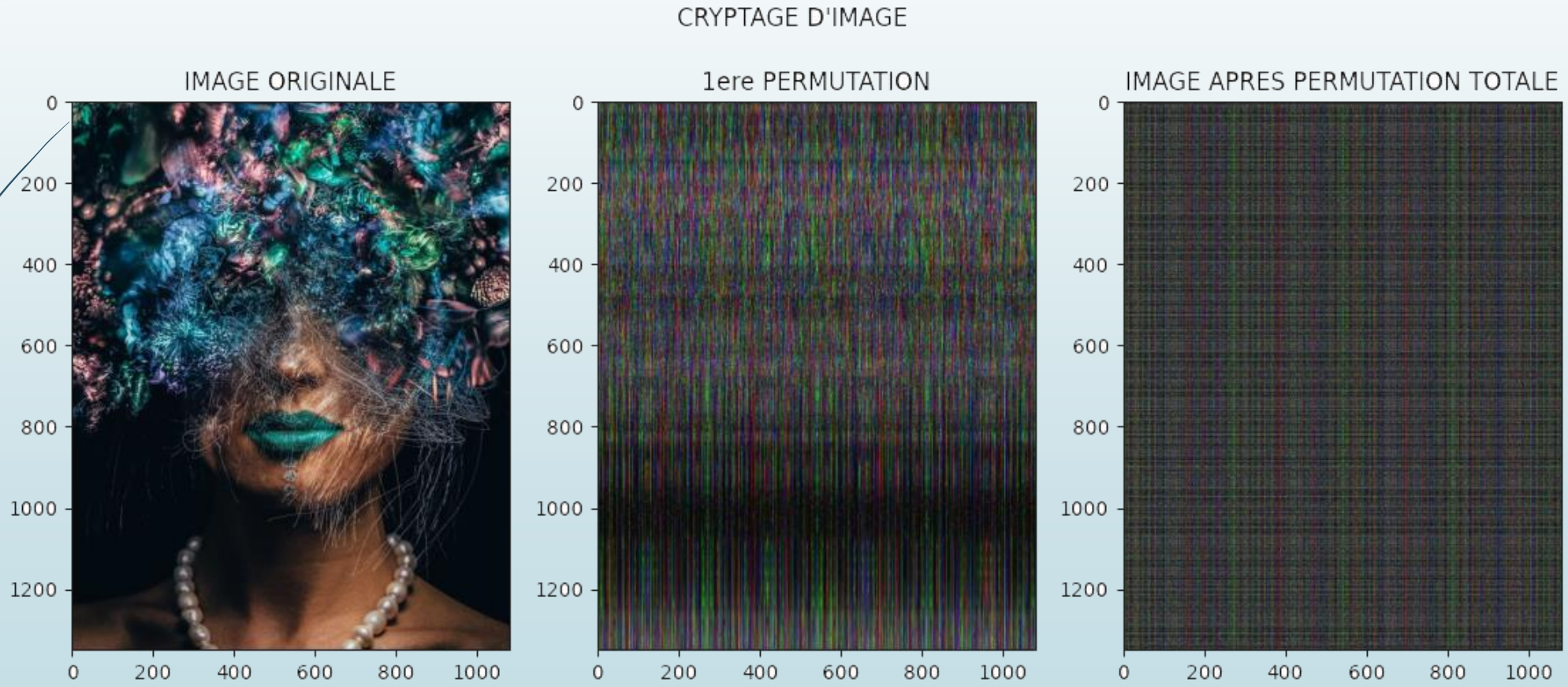
Réduction de la complexité de la permutation

- La permutation sur tous les pixels de l'image prends un grand cout en temps pour des valeurs de pixels de plus en plus grande (confère partie Analyse et Comparaisons).
- Pour y remédier, nous allons donc faire pour une image de taille **m*n** :
  - Une permutation sur chaque ligne **n*3**
  - Ensuite une permutation des lignes **m**

```python
def calculPositionSurLigne(nombre):
    #Calcul y et z des positions sur une  ligne
    y = nombre // 3
    z = nombre % 3
    return y, z

def cryptageSurLigne(image, u, x0):
    #Renvoyer deux images apres permutation sur ligne et ensuite des lignes
    nouvelleImage = zeros_like(image)

    hauteur = image.shape[0]
    largeur = image.shape[1]

    f = creationDePermutation(u, x0, largeur*3)

    for ligne in range(hauteur):
        for nombre in range(largeur*3):
            y, z = calculPositionSurLigne(nombre)
            nouveauY, nouveauZ = calculPositionSurLigne(f[nombre])
            nouvelleImage[ligne][y][z] = image[ligne][nouveauY][nouveauZ]

    imageFinale = zeros_like(image)

    f = creationDePermutation(u, x0, hauteur)

    for ligne in range(hauteur):
        imageFinale[ligne] = nouvelleImage[f[ligne]]

    return nouvelleImage, imageFinale

```

Présentation d'une nouvelle permutation créée grâce à une suite chaotique

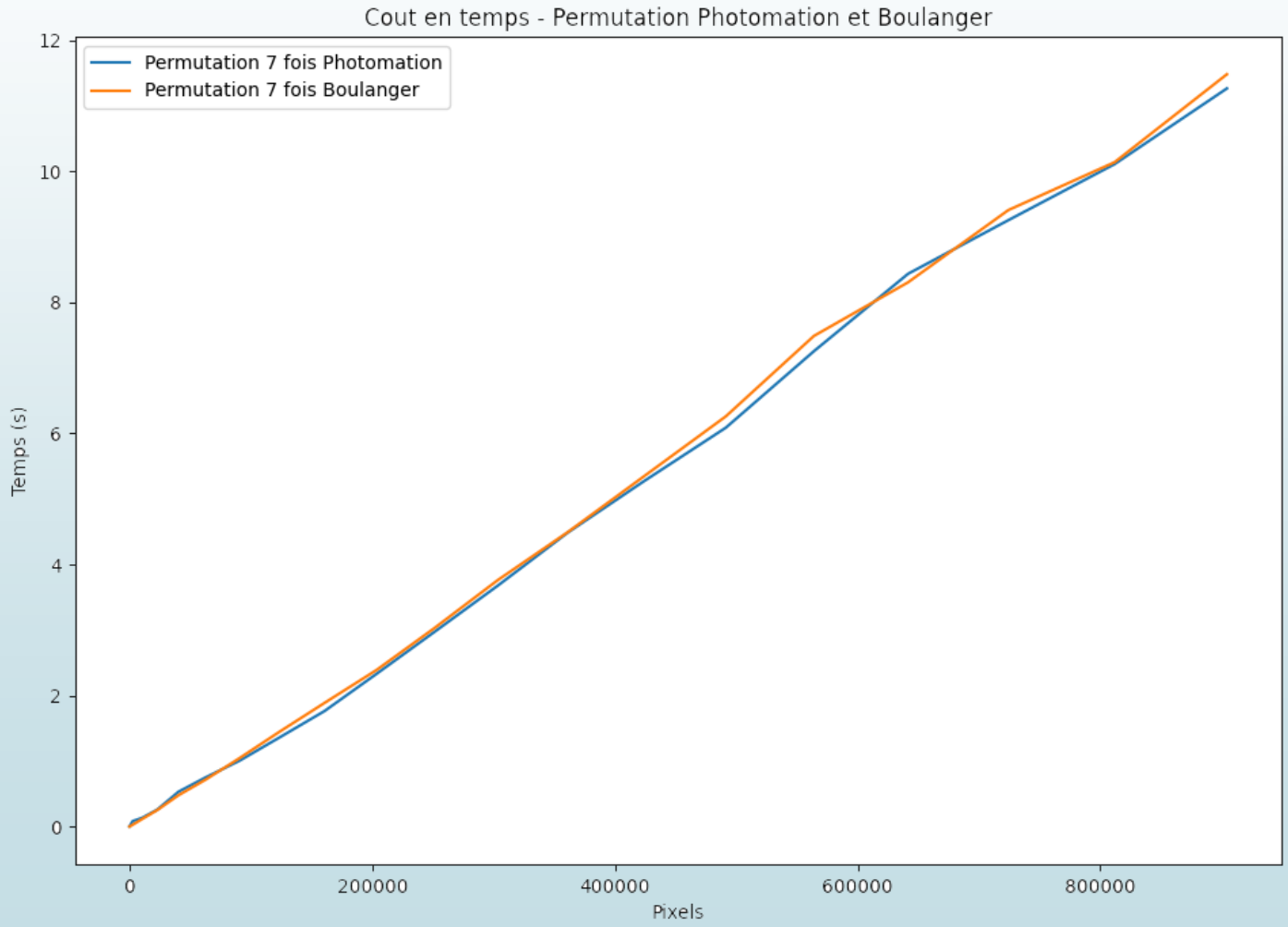
Résultats sur une image



Analyse des résultats et comparaisons

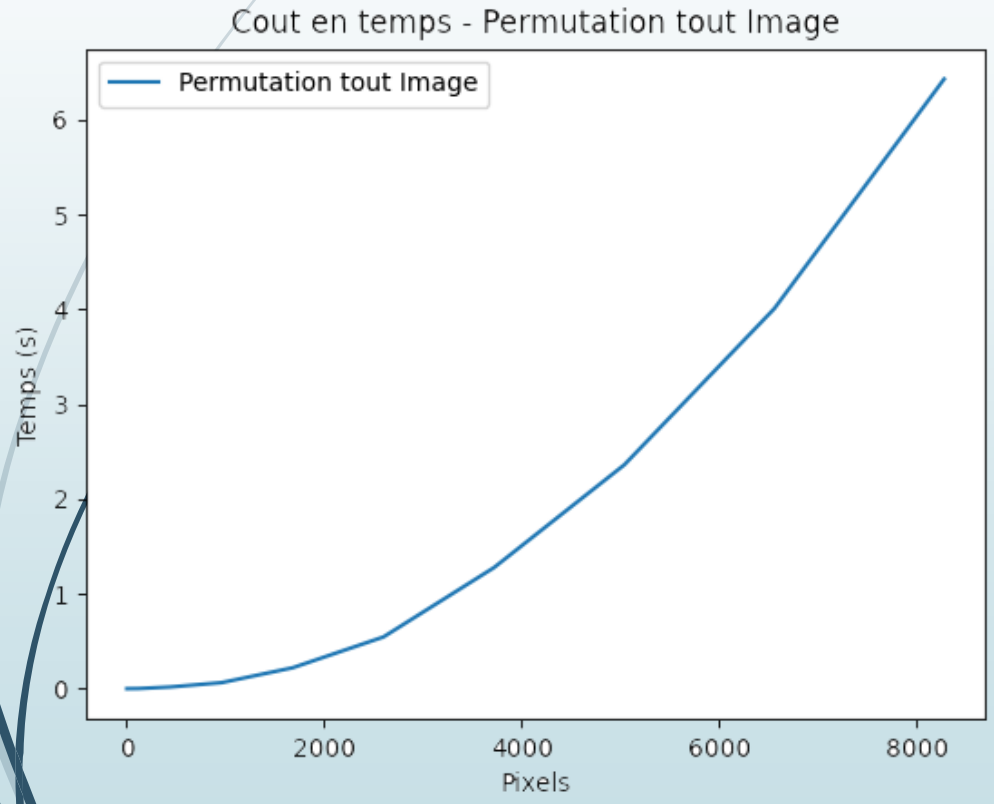
Comparaison Photomaton et Boulanger

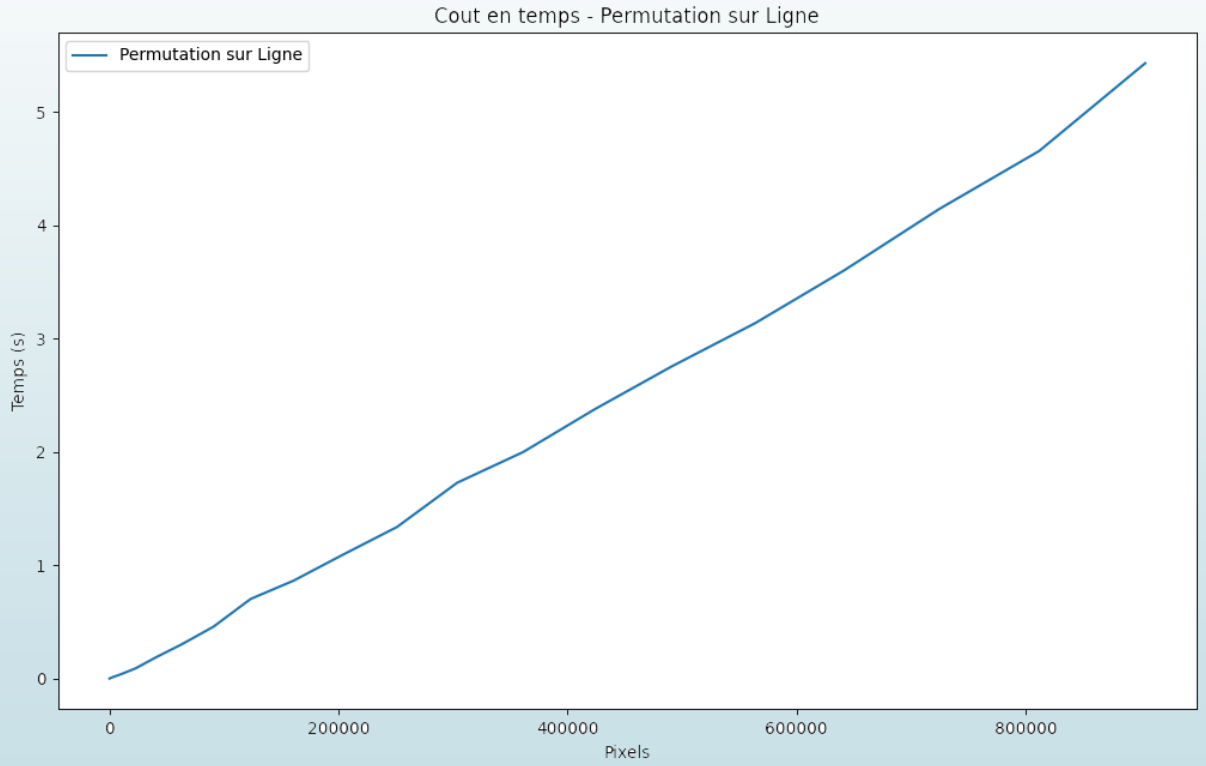
- Comparaison du coût en temps en fonction du nombre de pixel du **Photomaton** appliqué **7 fois** par rapport au **Boulanger** appliqué **7 fois**.



Analyse des résultats et comparaisons

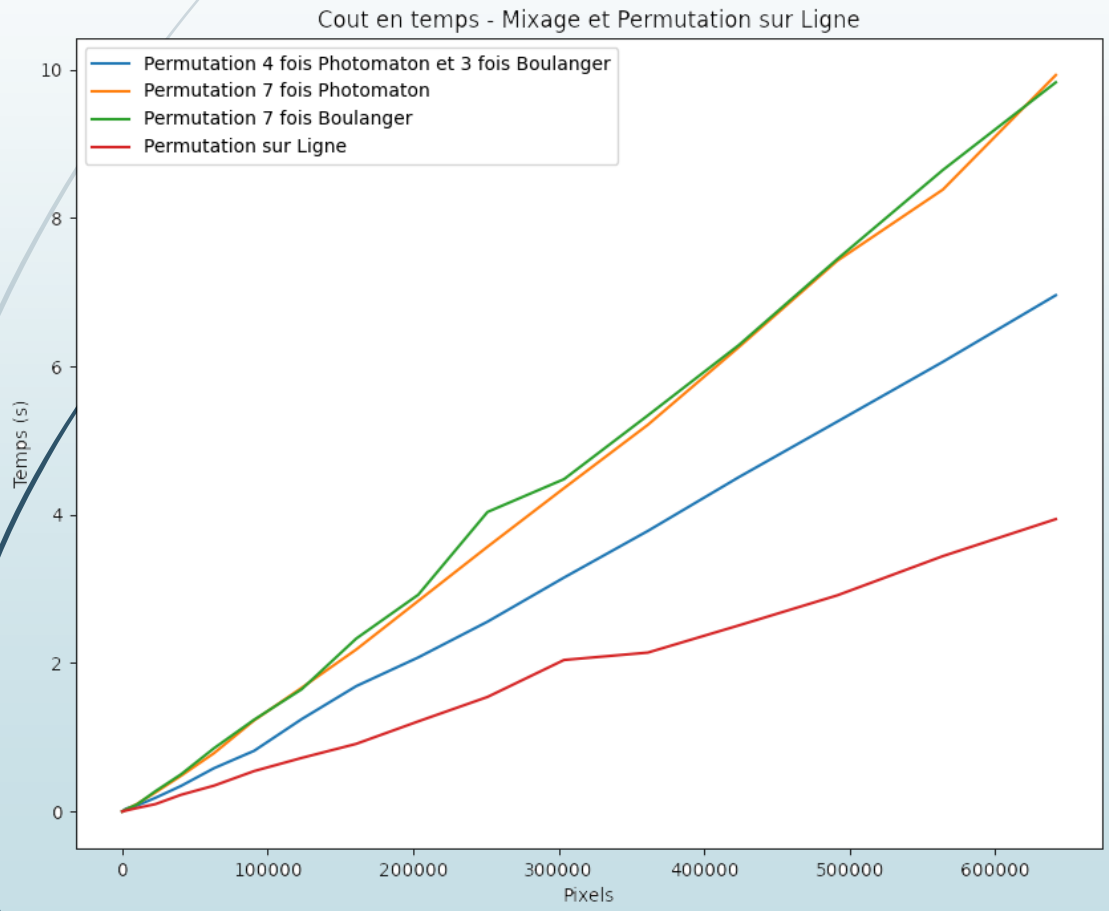
Comparaison Permutation Tout Image et Permutation Sur Ligne





Analyse des résultats et comparaisons

Mixage et Comparaisons



Code coût en temps de la permutation sur ligne

```python
from time import perf_counter
from PIL import Image
import matplotlib.pyplot as plt
from numpy import array

def creer_image(i):
    #Creer une image de taille ixi
    return array(PIL.Image.new('RGB', (i, i)))

def coutEnTemps(fonction, u, x0):
    pixels = [i**2 for i in range(1, 1000, 50)]
    temps = []
    for i in range(1, 1000, 50):
        image = creer_image(i)
        time1 = perf_counter()
        fonction(image, u, x0)
        time2 = perf_counter()
        temps.append(time2-time1)
    return pixels, temps

abcisse, ordonee = coutEnTemps(cryptageSurLigne, 3.58, 0.001)
plt.plot(abcisse, ordonee)
plt.xlabel("Pixels")
plt.ylabel("Temps (s)")
plt.title("Cout en temps - Permutation sur Ligne")
plt.legend()
plt.show()

```

On voit donc l'avantage d'utiliser la nouvelle Permutation sur Ligne par rapport à celle du Photomaton et du Boulanger

Conclusion

Le temps de cryptage devenant de plus en plus grand en fonction des pixels ne permets pas de traiter de manière rapide de grandes images avec des pixels très élevés. Pour palier à ce problème, on prends l'initiative de crypter par bloc comme le fait par exemple l'algorithme AES qui utilise des blocs de 8 ou 16 octets.

Annexe

Code Python complet de la présentation

C: > Users > Utilisateur > Pictures > TIPE.py

```python
import matplotlib.pyplot as plt
from numpy import *
from time import perf_counter
from PIL import Image

## Fonctions Usuelles
def lectureImage():
    chemin = input("Entrer le Chemin de l'image : ")
    image = plt.imread(chemin)
    return chemin, image

def nomDeImageSortie(chemin):
    liste = chemin.split("/")
    texte = "/".join(liste[:-1])
    texte += "/Cryptage_" + liste[-1]
    return texte

def suiteRobertMay(u, x0, n):
    for i in range(5):
        x0 = u*x0*(1-x0)
    listeValeurs = []
    for i in range(1, n+1):
        listeValeurs.append(x0)
        x0 = u*x0*(1-x0)
    return listeValeurs

def creationDePermutation(u, x0, n):
    listeValeurs = suiteRobertMay(u, x0, n)
    listeTriee = sorted(listeValeurs)
    permutation = [listeValeurs.index(i) for i in listeTriee]
    return permutation

def verificationSensibilite(u, x0, x1, n):
    #Sensibilite aux conditions initiales
    abcisses = [i for i in range(n)]
    permutation1 = creationDePermutation(u, x0, n)
    permutation2 = creationDePermutation(u, x1, n)

    plt.subplot(1, 2, 1)
    plt.plot(abcisses, permutation1)
    plt.title("Avec X = {}".format(x0))

    plt.subplot(1, 2, 2)
    plt.plot(abcisses, permutation2)
    plt.title("Avec X = {}".format(x1))

    plt.suptitle("SENSIBILITE AUX CONDTIONS INITIALES")
    plt.show()
```

C: > Users > Utilisateur > Pictures > TIPE.py

```python
def calculPositionSurLigne(nombre):
    #Calcul y et z des positions sur une  ligne
    y = nombre // 3
    z = nombre % 3
    return y, z

def calculPositionToutImage(nombre, largeur):
    #Calcul x, y, z a partir d'une valeur "nombre"
    x = nombre // (largeur * 3)
    nombre = nombre - largeur * 3 * x
    y = nombre // 3
    z = nombre % 3
    return x, y, z

## Programme de Cryptage

def cryptageSurLigne(image, u, x0):
    #Renvoyer deux images apres permutation sur ligne et ensuite des lignes
    nouvelleImage = zeros_like(image)

    hauteur = image.shape[0]
    largeur = image.shape[1]

    f = creationDePermutation(u, x0, largeur*3)

    for ligne in range(hauteur):
        for nombre in range(largeur*3):
            y, z = calculPositionSurLigne(nombre)
            nouveauY, nouveauZ = calculPositionSurLigne(f[nombre])
            nouvelleImage[ligne][y][z] = image[ligne][nouveauY][nouveauZ]

    imageFinale = zeros_like(image)

    f = creationDePermutation(u, x0, hauteur)

    for ligne in range(hauteur):
        imageFinale[ligne] = nouvelleImage[f[ligne]]

    return nouvelleImage, imageFinale

def cryptageToutImage(image, u, x0):
    #Retourne l'image apres une permuation des valeurs des pixels
    nouvelleImage = zeros_like(image)

    hauteur = image.shape[0]
    largeur = image.shape[1]

    f = creationDePermutation(u, x0, hauteur*largeur*3)
```

C: > Users > Utilisateur > Pictures > TIPE.py

```python
    f = creationDePermutation(u, x0, hauteur*largeur*3)

    for nombre in range(hauteur*largeur*3):
        x, y, z = calculPositionToutImage(nombre, largeur)
        nouveauX, nouveauY, nouveauZ = calculPositionToutImage(f[nombre], largeur)
        nouvelleImage[x][y][z] = image[nouveauX][nouveauY][nouveauZ]

    return nouvelleImage

def decryptageSurLigne(imageCryptee, u, x0):
    nouvelleImage = zeros_like(imageCryptee)

    hauteur = imageCryptee.shape[0]
    largeur = imageCryptee.shape[1]

    f = creationDePermutation(u, x0, hauteur)

    for ligne in range(hauteur):
        nouvelleImage[f[ligne]] = imageCryptee[ligne]

    imageOriginale = zeros_like(imageCryptee)

    f = creationDePermutation(u, x0, largeur*3)

    for ligne in range(hauteur):
        for nombre in range(largeur*3):
            y, z = calculPositionSurLigne(nombre)
            nouveauY, nouveauZ = calculPositionSurLigne(f[nombre])
            imageOriginale[ligne][nouveauY][nouveauZ] = nouvelleImage[ligne][y][z]

    return imageOriginale


def decryptageToutImage(imageCryptee, u, x0):
    imageOriginale = zeros_like(imageCryptee)

    hauteur = imageCryptee.shape[0]
    largeur = imageCryptee.shape[1]

    f = creationDePermutation(u, x0, hauteur*largeur*3)

    for nombre in range(hauteur*largeur*3):
        x, y, z = calculPositionToutImage(nombre, largeur)
        nouveauX, nouveauY, nouveauZ = calculPositionToutImage(f[nombre], largeur)
        imageOriginale[nouveauX][nouveauY][nouveauZ] = imageCryptee[x][y][z]

    return imageOriginale
```

C: > Users > Utilisateur > Pictures > TIPE.py

```python

def affichageImage(image1, image2, texte1, texte2, texte):
    #Affiche les images passees en parametres
    plt.subplot(1, 2, 1)
    plt.imshow(image1)
    plt.title(texte1)

    plt.subplot(1, 2, 2)
    plt.imshow(image2)
    plt.title(texte2)

    plt.suptitle(texte)
    plt.show()

def importTaillePair(chemin):
    #Importe une image en rendant les dimensions pairs
    image = plt.imread(chemin)
    image = image[:, :, :3]
    hauteur = image.shape[0] - image.shape[0]%2
    largeur = image.shape[1] - image.shape[1]%2
    image = image[0:hauteur]
    image = [ligne[0:largeur + 1] for ligne in image]
    return np.array(image)

def photomaton(x, hauteur, y, largeur):
    """ Calcul des nouvelles positions grace au principe
    du photomaton """
    if x%2 == 0:
        x = x//2
    else:
        x = x//2 + hauteur//2
    if y%2 == 0:
        y = y//2
    else:
        y = y//2 + largeur//2
    return x, y

def boulanger(x, hauteur, y, largeur):
    """ Calcul des nouvelles positions grace au principe
    du boulanger """
    if x%2 == 0:
        x, y = x//2, 2*y
    else:
        x, y = x//2, 2*y+1
    if y < largeur:
        return x, y
    else:
        return hauteur - 1 - x, 2*largeur - 1 - y
```

C: > Users > Utilisateur > Pictures > TIPE.py

```python

def applicationFonction(image, fonction):
    #Sert a appliquer a une image, une permutation donnee
    nouvelle_image = zeros_like(image)
    hauteur, largeur = image.shape[0], image.shape[1]
    for i in range(hauteur):
        for j in range(largeur):
            x,y = fonction(i, hauteur, j, largeur)
            nouvelle_image[x][y] = image[i][j]
    return nouvelle_image

## Exemple de PROGRAMME PRINCIPALE qui affiche la sensibilite
## ensuite l'image de la permutation tout image.

u = 3.58
x0 = 0.001
chemin, image = lectureImage()

verificationSensibilite(u, x0, x0+0.001, 100)

image1 = cryptageToutImage(image, u, x0)
affichageImage(image, image1)
nouveauNom = nomDeImageSortie(chemin)
plt.imsave(nouveauNom, image1)
```