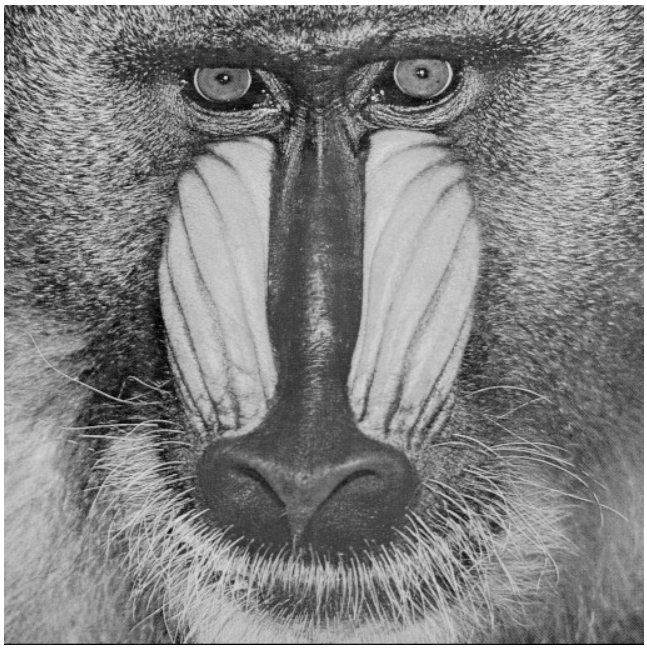
HUNOT-MARTIN
Alaric
M2 Imagine
Université Montpellier

Compte Rendue TP1 HAI918

Chiffrement d’images par mélange et substitution

Partie 1 : Chiffrement par mélange

Pendant toute la durée du tp j’ai travaillé avec l’image suivante :



baboon.pgm

a) Voici le code de permutation.cpp

```c
void generate_permutation(int* permutation, int size, int key) {
    srand(key);

    for (int i = 0; i < size; i++) {
        permutation[i] = i;
    }

    for (int i = size - 1; i > 0; i--) {
        int j = rand() % (i + 1);
        int temp = permutation[i];
        permutation[i] = permutation[j];
        permutation[j] = temp;
    }
}
```

fonction pour génerer la permutation

```c
int main(int argc, char* argv[]) {
    char cNomImgLue[250], cNomImgEcrite[250];
    int nH, nW, nTaille, key;

    if (argc != 4) {
        printf("Usage: ImageIn.pgm key ImageOut.pgm\n");
        exit(1);
    }
    sscanf(argv[1], "%s", cNomImgLue);
    sscanf(argv[2], "%d", &key);
    sscanf(argv[3], "%s", cNomImgEcrite);

    OCTET *ImgIn, *ImgOut;

    lire_nb_lignes_colonnes_image_pgm(cNomImgLue, &nH, &nW);
    nTaille = nH * nW;

    allocation_tableau(ImgIn, OCTET, nTaille);
    lire_image_pgm(cNomImgLue, ImgIn, nTaille);
    allocation_tableau(ImgOut, OCTET, nTaille);

    int *permutation = (int*)malloc(nTaille * sizeof(int));
    generate_permutation(permutation, nTaille, key);

    for (int i = 0; i < nTaille; i++) {
        ImgOut[permutation[i]] = ImgIn[i];
    }


    ecrire_image_pgm(cNomImgEcrite, ImgOut, nH, nW);

    free(ImgIn);
    free(ImgOut);
    free(permutation);

    return 0;
}
```

fonction main

1. Lecture de l'image d'entrée :

- Le programme commence par charger une image PGM. Il récupère la taille de l'image (largeur et hauteur).

- La taille totale de l'image, c'est-à-dire le nombre total de pixels, est calculée par la multiplication de nH (hauteur) et nW (largeur).

2. Allocation de mémoire :

- Deux tableaux dynamiques, ImgIn et ImgOut, sont alloués pour stocker respectivement les données des pixels de l'image d'entrée et les données des pixels réarrangés de l'image de sortie. Ces tableaux contiennent des OCTET (probablement des unsigned char), correspondant à chaque pixel.

3. Génération de la permutation aléatoire :

- Le programme utilise une fonction de permutation aléatoire generate_permutation pour générer une séquence d'indices aléatoires en fonction d'une clé fournie par l'utilisateur. Cette clé est utilisée pour initialiser le générateur pseudo-aléatoire avec srand(key), garantissant que la permutation sera toujours la même pour une clé donnée.

- La permutation est réalisée en utilisant l'algorithme de **Fisher-Yates**. Cet algorithme mélange efficacement les indices dans le tableau permutation en parcourant le tableau de la fin vers le début et en échangeant chaque élément avec un autre élément aléatoire.

4. Réarrangement des pixels :

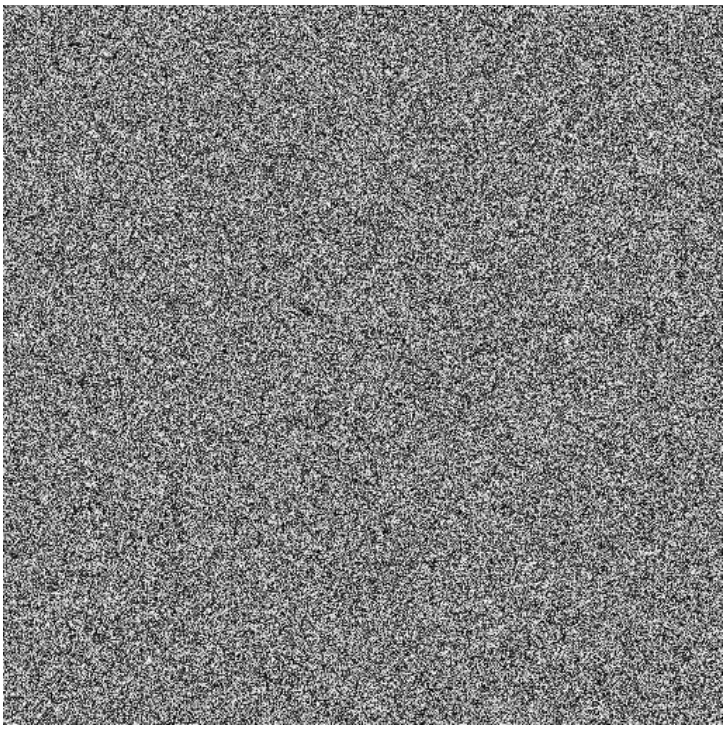
- Une fois la permutation générée, le programme applique cette permutation aux pixels de l'image d'entrée (ImgIn). Pour chaque pixel à l'index i, le programme place sa valeur à l'index permutation[i] dans l'image de sortie (ImgOut).

- Cela signifie que les pixels de l'image d'entrée sont réarrangés selon l'ordre déterminé par la permutation aléatoire, créant ainsi une version mélangée de l'image.

5. Écriture de l'image de sortie :

- Après avoir appliqué la permutation, le programme écrit l'image modifiée dans un fichier PGM de sortie spécifié par l'utilisateur, en utilisant la fonction ecrire_image_pgm.

On obtient l’image suivante qui est bel est bien chiffrée en utilsant la clé 100 à l’éxecution.



baboon.pgm aprés chiffrement par permutation avec la clé 100

b) J'ai répété l'experience avec une autre image pgm et la clé 91

image originale

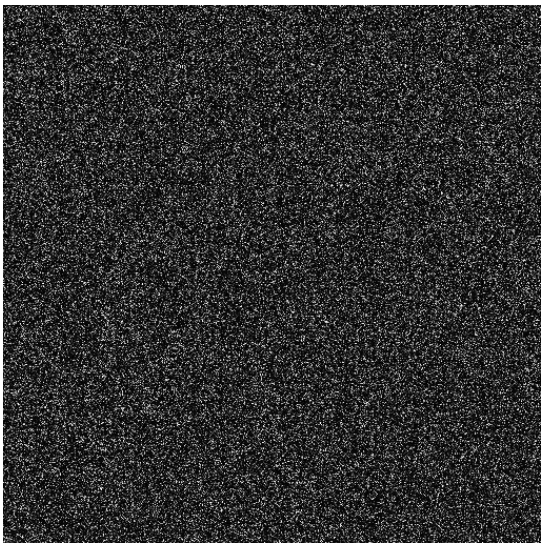


image chiffré

c) a. en utilisant la formule du psnr :

$$PSNR = 10\log_{10}\frac{255^2}{MSE}\text{dB}$$

On obtient un psnr de a peu prés 11.5db ce qui est tres bas mais ce qui est justifié car il n'y a que trés peu de différence possible entre les deux images

```
for (int i = 0; i < nTaille; i++)
{
  sommeErreur += (ImgIn[i] - ImgIn2[i]) * (ImgIn[i] - ImgIn2[i]);
}

float EQM = sommeErreur / nTaille;


float PSNR = 10.0 * log10((255.0 * 255.0) / EQM);
printf("PSNR = %f dB\n", PSNR);
```

```
PSNR = 11.541135 dB
```

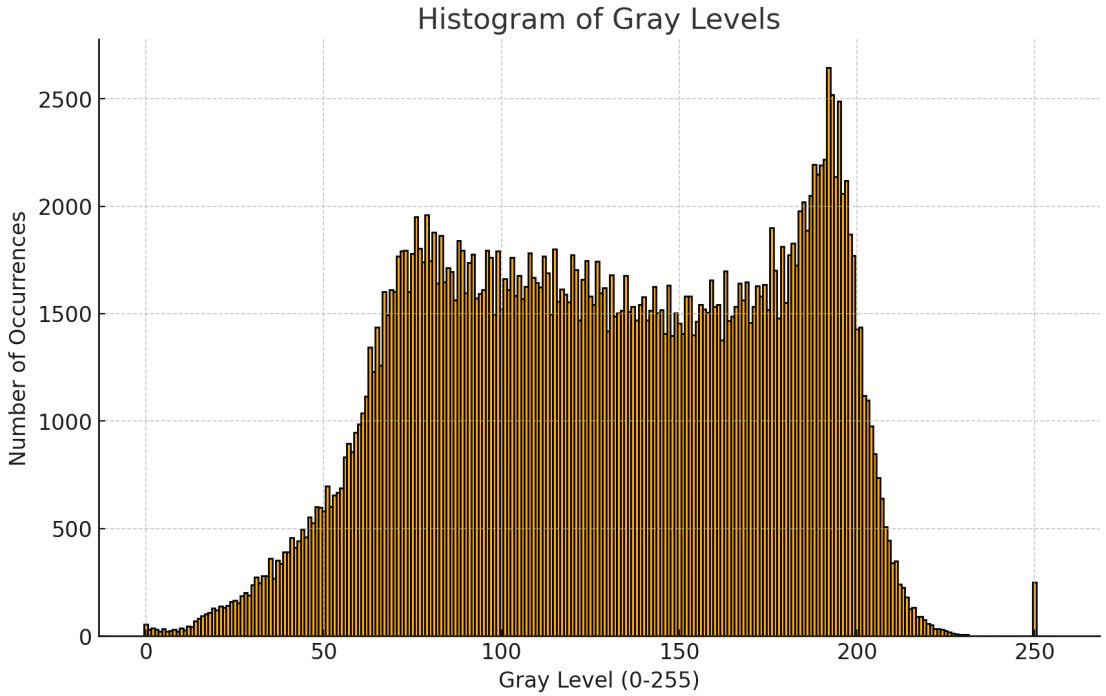
c) b. en utilisant la formule de l'entropie : $$\mathrm{H}(X) = -\sum_{i=1}^{n} \mathrm{P}(x_i) \log \mathrm{P}(x_i)$$

on remarque que l'entropie de l'image originale et celle de l'image modifier sont identiques ce qui est logique car on ne fait qu'échanger les positions des pixels, on preserve donc la meme distribution en niveau de gris.

```
for (int j=0; j < 255; j++)
{
    histo[j]= histo[j]/nTaille;
    controle += histo[j];
    transi[j] = histo[j]*log2f(histo[j]);
    if (transi[j]!=transi[j]) transi[j]=0;
    entropie += transi[j];
    printf ( "value entropie = %f \n",transi[j]);
}
printf ( "value entropie = %f \n",entropie);
entropie = entropie * -1 ;
```

```
value entropie = 7.474432
```

d) On remarque que le histogrammes des 2 images sont identiques car comme on ne fait que permuter des pixels la distribution en niveau de gris des 2 images est identique.



histogramme des 2 images

e) Voici le code de perm_inv.cpp

```c
#include <stdio.h>
#include <stdlib.h>
#include "image_ppm.h"
#include <time.h>

int main(int argc, char* argv[])
{
    char cNomImgLue[250], cNomImgEcrite[250];
    int nH, nW, nTaille;
    int K;

    if (argc != 4)
    {
        printf("Usage: ImageEncrypted.pgm Key(0-255) ImageDecrypted.pgm\n");
        exit(1);
    }

    sscanf(argv[1], "%s", cNomImgLue);
    sscanf(argv[2], "%d", &K);
    sscanf(argv[3], "%s", cNomImgEcrite);

    if (K < 0 || K > 255)
    {
        printf("Key K must be between 0 and 255.\n");
        exit(1);
    }

    OCTET *ImgIn, *ImgOut;

    lire_nb_lignes_colonnes_image_pgm(cNomImgLue, &nH, &nW);
    nTaille = nH * nW;

    allocation_tableau(ImgIn, OCTET, nTaille);
    lire_image_pgm(cNomImgLue, ImgIn, nTaille);
    allocation_tableau(ImgOut, OCTET, nTaille);

    int *indices;
    allocation_tableau(indices, int, nTaille);
```

```c
for (int i = 0; i < nTaille; i++)
    indices[i] = i;

srand(K);

for (int i = nTaille - 1; i > 0; i--)
{
    int j = rand() % (i + 1);
    int temp = indices[i];
    indices[i] = indices[j];
    indices[j] = temp;
}

for (int i = 0; i < nTaille; i++)
{
    ImgOut[i] = ImgIn[indices[i]];
}

ecrire_image_pgm(cNomImgEcrite, ImgOut, nH, nW);
free(ImgIn);
free(ImgOut);
free(indices);

return 0;
```

Lecture de l'image d'entrée :

- Le programme commence par charger une image PGM, en niveaux de gris, en utilisant la fonction lire_nb_lignes_colonnes_image_pgm pour obtenir ses dimensions : hauteur (nH) et largeur (nW).
- La taille totale de l'image, c'est-à-dire le nombre total de pixels, est calculée par la multiplication de nH (hauteur) et nW (largeur), soit nTaille.

Allocation de mémoire :

- Deux tableaux dynamiques, ImgIn et ImgOut, sont alloués pour stocker respectivement les données des pixels de l'image d'entrée et celles de l'image après réarrangement.
- Le tableau ImgIn contient les valeurs de pixels lues à partir de l'image PGM d'entrée, tandis que ImgOut sera utilisé pour stocker l'image après avoir réappliqué la permutation.
- Un troisième tableau, indices, est également alloué pour contenir les indices des pixels afin de générer la permutation.

Génération de la permutation aléatoire :

- Le tableau indices est initialisé avec les valeurs de 0 à nTaille-1, chaque indice représentant la position d'un pixel dans l'image.

- Une clé (K), fournie par l'utilisateur, est utilisée pour initialiser le générateur pseudo-aléatoire via la fonction srand(K). Cette clé garantit que la permutation sera toujours la même pour une clé donnée.
- Ensuite, l'algorithme de **Fisher-Yates** est utilisé pour mélanger les indices du tableau indices. Cet algorithme parcourt le tableau de la fin au début et échange chaque élément avec un autre élément aléatoire précédemment rencontré dans le tableau. Cela génère une permutation aléatoire des indices des pixels.

Réarrangement des pixels (Permutation inverse) :

- Le programme applique la permutation inversée aux pixels de l'image d'entrée. Pour chaque pixel à l'index `i` dans ImgIn, le programme place sa valeur à l'index indices[i] dans le tableau ImgOut.
- Ainsi, la permutation des pixels est inversée, rétablissant l'ordre original ou créant une version mélangée de l'image en fonction de la clé utilisée.

Écriture de l'image de sortie :

- Après avoir réappliqué la permutation, l'image modifiée est écrite dans un fichier PGM de sortie spécifié par l'utilisateur, en utilisant la fonction ecrire_image_pgm.
- Le programme libère ensuite la mémoire allouée pour les tableaux ImgIn, ImgOut, et `indices` avant de terminer.

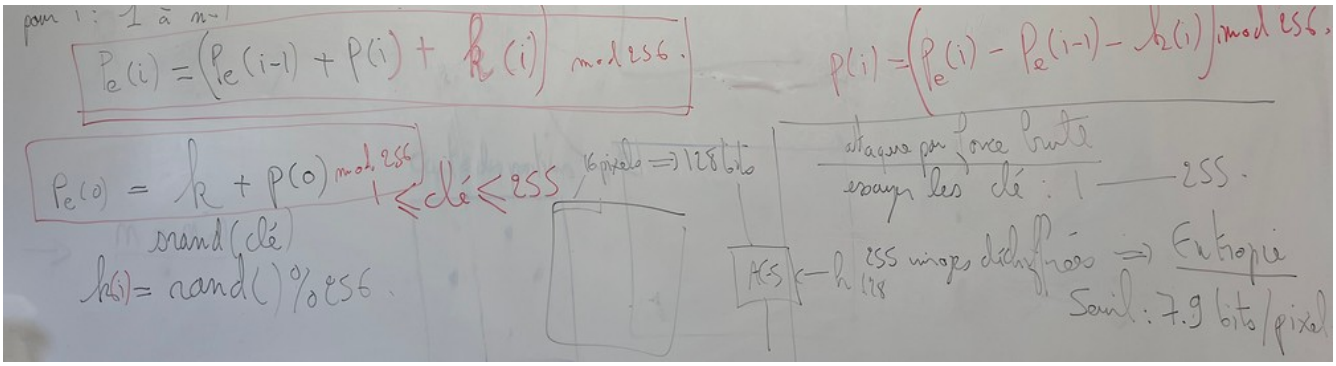
En déchiffrant l’image avec la clé 100 on retombe bien sur l’image originale pour baboon.pgm et cette clé est bel est bien la seule permettant de déchiffrer l’image.

L’image originale et l’image déchiffrer sont identiques: le psnr entre les deux images est infini.

```
PSNR = inf
```

Cette méthode de chiffrement n’est pas optimal car on remarque la preservation de l’entropie et de la distribution en niveau de gris pouvant donner des renseignement sur le contenu de l’image d’origine. Il nous faut donc arriver à une methode plus performante avec une distribution uniforme des niveaux de gris de l’histogramme. On va donc utiliser une deuxieme méthode de chiffrement par substitution.

Partie 2 : Chiffrement par substitution



formules du cour chiffrement par substitution

Voici le code de substitution.cpp :

```
allocation_tableau(ImgIn, OCTET, nTaille);
lire_image_pgm(cNomImgLue, ImgIn, nTaille);
allocation_tableau(ImgOut, OCTET, nTaille);
k = rand() % 256;
ImgOut[0] = (ImgIn[0] + k) % 256;
for (int i = 0; i < nTaille; i++) {
    k = rand() % 256;
    ImgOut[i] = (ImgOut[i-1]+ImgIn[i] + k) % 256;
}

ecrire_image_pgm(cNomImgEcrite, ImgOut, nH, nW);
free(ImgIn);
free(ImgOut);
```

1. Lecture des arguments et initialisation : Le programme prend trois arguments en entrée : le fichier image PGM d'origine, une clé numérique, et le fichier de sortie où l'image modifiée sera enregistrée. La clé est utilisée pour initialiser un générateur de nombres pseudo-aléatoires qui contrôlera la modification des pixels. Cela garantit que si la même clé est utilisée, le même résultat sera obtenu à chaque exécution.

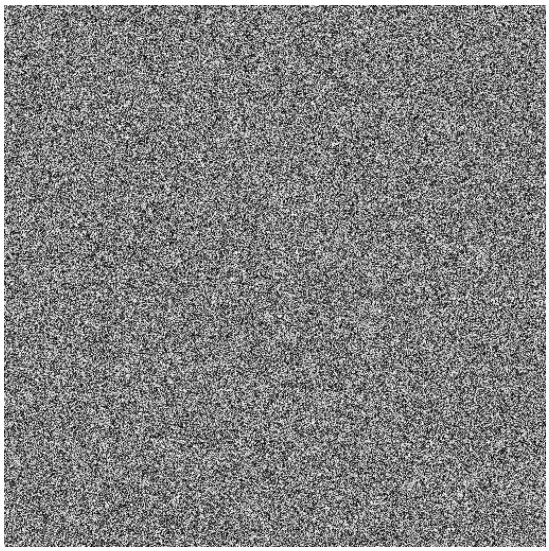
2.Chiffrement par substitution basé sur les pixels : La technique utilisée ici est un type de chiffrement par substitution, où chaque pixel de l'image est modifié en fonction de la valeur du pixel précédent, de la valeur du pixel actuel, et d'une valeur aléatoire générée à partir de la clé. Pour le premier pixel, une valeur aléatoire est simplement ajoutée à la valeur du pixel, et le résultat est pris modulo 256 pour s'assurer qu'il reste dans la plage valide des niveaux de gris (0 à 255). Pour les pixels suivants, le programme calcule la nouvelle valeur du pixel de sortie en ajoutant la valeur du pixel précédent dans l'image modifiée, la valeur du pixel dans l'image originale, et une nouvelle valeur aléatoire générée à partir de la clé.

3.Application de la substitution à tous les pixels : Le processus de substitution est appliqué à chaque pixel de l'image, en s'assurant que les modifications des pixels sont enchaînées (chaque pixel de sortie dépend du pixel précédent modifié). Ce mécanisme introduit un effet de cascade

où chaque pixel est influencé à la fois par sa propre valeur et par la séquence des pixels précédents, ce qui renforce l'effet de chiffrement.

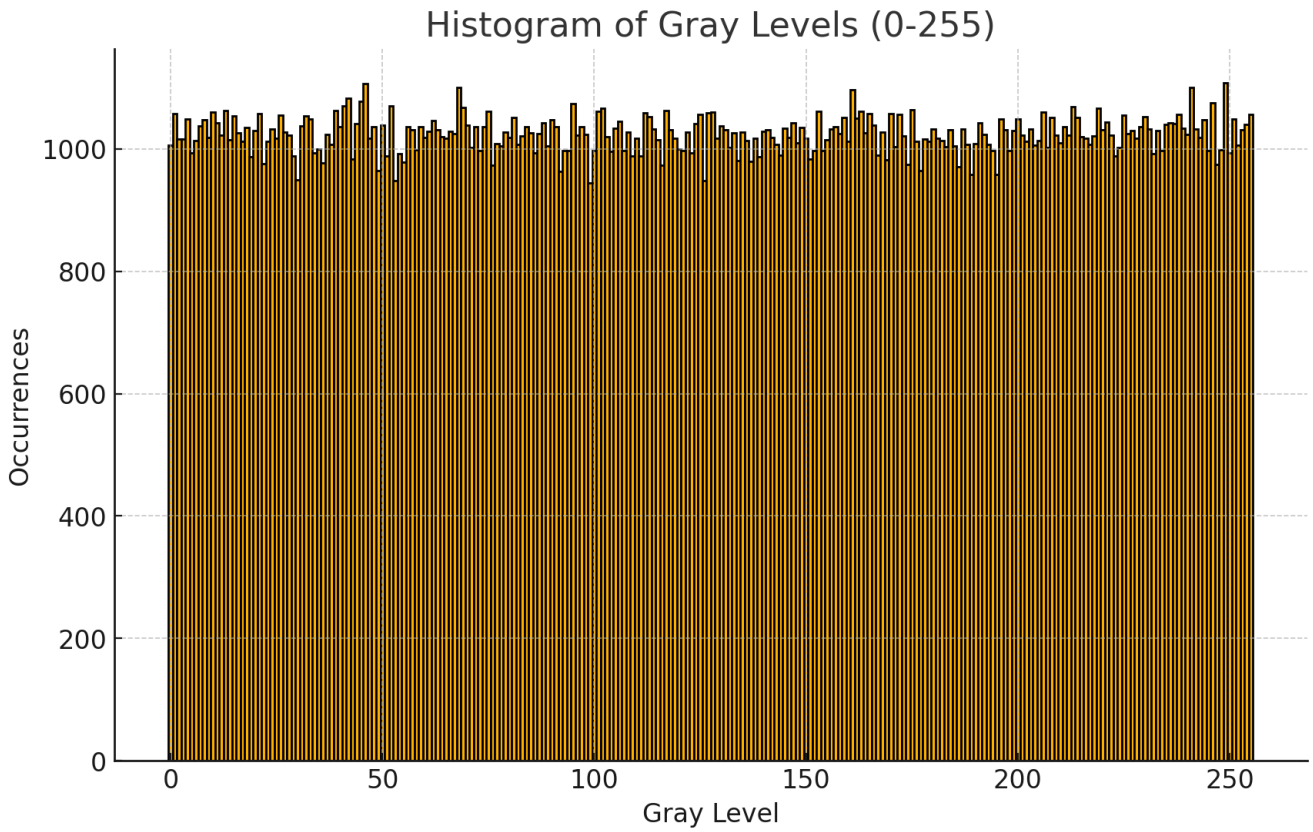
4.Enregistrement de l'image modifiée : Une fois que tous les pixels de l'image ont été modifiés en fonction de cette technique de substitution pseudo-aléatoire, le programme enregistre l'image transformée dans le fichier de sortie PGM. Ce chiffrement par substitution permet de rendre l'image méconnaissable par rapport à l'originale, tout en étant réversible si on utilise la même clé pour décrypter.

On obtient cette image avec baboon.pgm et la clé 100 :



baboon.pgm chiffré par substitution

En analysant l'histogramme de l'image on remarque une distribution plutot uniforme des niveaux de gris de l'image autour de la valeur 1000. Cette méthode est donc plus sécurisé.



histogramme de baboon.pgm chiffré par substitution

L'entropie de l'image par substitution : `value entropie = 7.967914`

Voici le code mainetenant pour déchiffrer l'image, sub_inv.cpp :

```
allocation_tableau(ImgIn, OCTET, nTaille);
lire_image_pgm(cNomImgLue, ImgIn, nTaille);
allocation_tableau(ImgOut, OCTET, nTaille);
k = rand() % 256;
ImgOut[0] = (ImgIn[0] - k) % 256;
for (int i = 0; i < nTaille; i++) {
    k = rand() % 256;
    ImgOut[i] = (ImgIn[i]-ImgIn[i-1] - k ) % 256;
}

ecrire_image_pgm(cNomImgEcrite, ImgOut, nH, nW);
free(ImgIn);
free(ImgOut);
```

1.Lecture des arguments et initialisation :
Le programme prend trois arguments : le fichier de l'image chiffrée au format PGM, une clé numérique, et le fichier où l'image déchiffrée sera enregistrée. La clé numérique est utilisée pour initialiser un générateur de nombres pseudo-aléatoires afin de recréer la séquence aléatoire utilisée lors du chiffrement. Cela garantit que la même clé appliquée permet de décrypter l'image de manière cohérente.

2.Déchiffrement par substitution basé sur les pixels :
Ce programme implémente une technique de déchiffrement par substitution inverse. Le processus commence par lire l'image chiffrée et appliquer une opération inverse pour restaurer les pixels à leur état d'origine. Pour le premier pixel, une valeur aléatoire est générée avec la clé, et le programme soustrait cette valeur au premier pixel de l'image chiffrée. Ce résultat est ensuite pris modulo 256 pour rester dans l'intervalle valide des valeurs de niveaux de gris (0 à 255).

Pour chaque pixel suivant, le programme génère une nouvelle valeur aléatoire et applique une soustraction au pixel actuel dans l'image chiffrée. La valeur du pixel déchiffré est calculée en prenant la valeur actuelle du pixel chiffré, en soustrayant la valeur du pixel précédent dans l'image originale (non chiffrée) et la nouvelle valeur aléatoire, puis en prenant le résultat modulo 256 pour maintenir les valeurs entre 0 et 255.

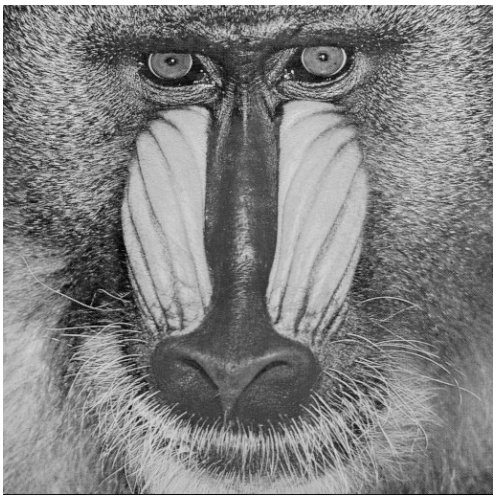
3.Application de la substitution inverse :
Ce processus de déchiffrement est appliqué à chaque pixel de l'image, pixel par pixel. Chaque pixel est donc restauré en fonction du pixel précédent et de la séquence pseudo-aléatoire générée par la clé. La relation entre chaque pixel et son prédécesseur est inversée pour redonner les valeurs originales des pixels de l'image non chiffrée.

4.Enregistrement de l'image déchiffrée :
Après avoir restauré tous les pixels, le programme enregistre l'image déchiffrée dans le fichier

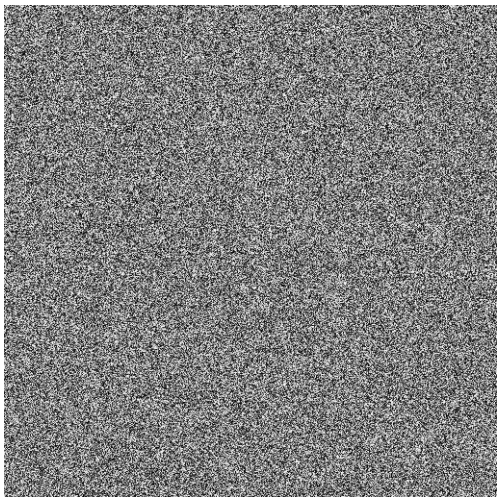
de sortie PGM spécifié. La structure de l'image est donc restaurée et réécrite sous sa forme non chiffrée.

5.Libération de la mémoire :
Enfin, le programme libère la mémoire allouée pour les tableaux contenant les pixels des images d'entrée (chiffrée) et de sortie (déchiffrée) avant de terminer.

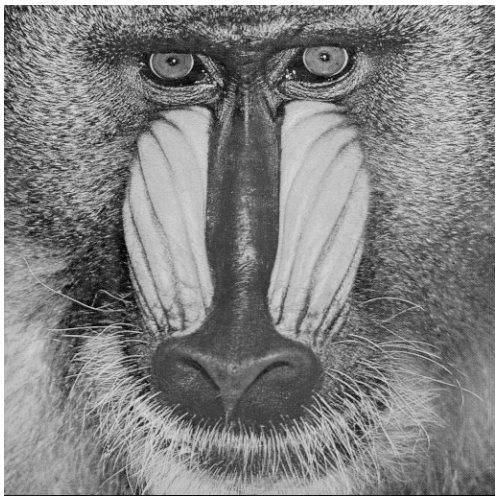
On retombe bien sur l’image originale avec la clé 100 qui est l’unique clé qui fonctionne pour déchiffrer l’image.



originale



substitution



substitution inverse

Conclusion :

Ce TP nous a permis d'explorer en profondeur la technique de chiffrement par mélange et substitution. Nous avons appris à utiliser une clé pour générer une séquence pseudo-aléatoire qui permet de chiffrer et de déchiffrer des images en niveaux de gris. Le processus consiste à brouiller les données de l'image en modifiant chaque pixel en fonction de la clé, puis à les restaurer en appliquant l'opération inverse avec la même clé. Cette méthode, bien que simple, montre l'importance d'une clé unique et sécurisée dans le processus de chiffrement. Elle est efficace pour protéger des informations, mais reste vulnérable en cas de divulgation de la clé.

Pour aller plus loin, il serait intéressant de se pencher sur des méthodes de chiffrement plus avancées, comme l'algorithme AES (Advanced Encryption Standard), qui est largement utilisé dans les applications modernes de cryptographie. AES offre une sécurité plus robuste grâce à des clés plus longues et des transformations plus complexes, rendant les tentatives de décryptage beaucoup plus difficiles. Comparé au chiffrement par substitution que nous avons étudié, AES représente une avancée

majeure en termes de protection des données dans un monde où la sécurité de l'information est devenue une priorité.