

# Semaine 8 : Tatouage d'image par étirement de spectre

## TP8

Maxime CATTEAU  
Léane TEXIER

### 1ère partie : Introduction : enregistreur de macros

**Note :** La première partie nous a permis de nous familiariser avec l'enregistreur de macros proposé par le logiciel ImageJ. Cet outil permet de créer rapidement une macro grâce à l'enregistrement des différents traitements effectués par l'utilisateur afin qu'il puisse les refaire facilement sans perdre de temps à refaire toutes les manipulations à la main.

### 2ème partie : Filtres de contours

Duplicer 2 fois l'image peppers en renommant ces copies peppers\_dx et peppers\_dy (elles vont représenter les dérivées partielles selon les directions principales du plan image x et y). Convertir ces images sur 32 bits grâce au menu **Image → Type → 32-bit**. Pourquoi ces images doivent-elles être codées sur 32 bits ?

Afin de commencer notre expérience, nous devons effectuer une première manipulation qui consiste à convertir le type de notre image d'analyse. Ainsi, nous passons d'une image 8-bits à une image 32-bits.



A gauche, l'image 'peppers.png' en 8-bits. A droite, la même image convertie en 32-bits

L'image doit impérativement être convertie en 32-bits car les filtres de Sobel ont des valeurs qui peuvent être négatives. Le résultat obtenu (qui peut donc être aussi bien positif que négatif) ne rentre donc pas sur les 8 bits initiaux.

Dériver *peppers\_dx* par rapport à x et *peppers\_dy* par rapport à y, en utilisant les filtres de Sobel (rappelés ci-dessous). L'opération de convolution sous ImageJ se fait grâce au menu **Process → Filters → Convolve...**

Une fois que nous avons dupliqué notre image 2 fois (images créées sous les noms de *peppers\_dx.png* et *peppers\_dy.png*). Nous dérivons '*peppers\_dx.png*' par rapport à x (l'axe des abscisses) et '*peppers\_dy.png*' par rapport à y (l'axe des ordonnées) en utilisant les filtres de Sobel.

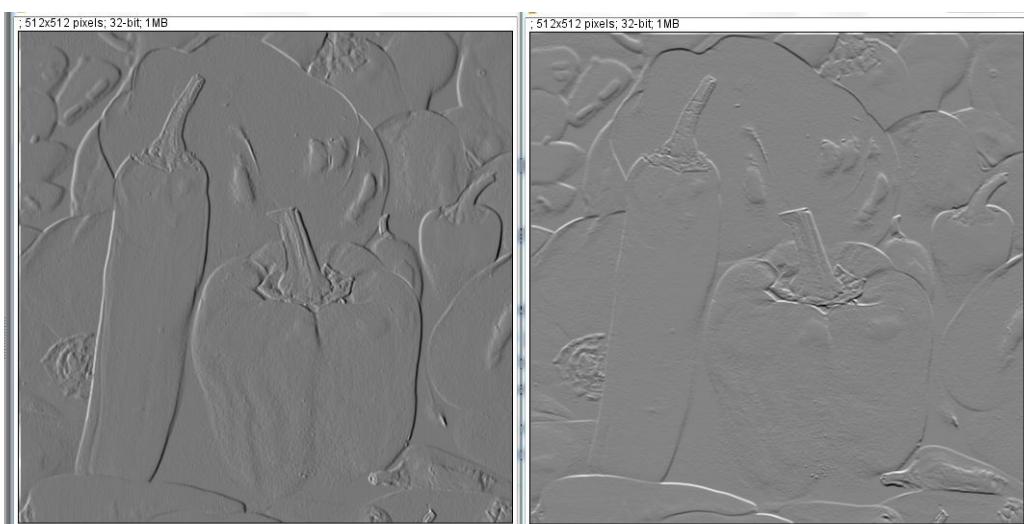
Pour rappel, les filtres de Sobel sont définis tels que :

Filtre de Sobel pour dériver suivant x =

$$\begin{array}{ccc} -0.125 & 0 & 0.125 \\ -0.25 & 0 & 0.25 \\ -0.125 & 0 & 0.125 \end{array}$$

Filtre de Sobel pour dériver suivant y =

$$\begin{array}{ccc} -0.125 & -0.25 & -0.125 \\ 0 & 0 & 0 \\ 0.125 & 0.25 & 0.125 \end{array}$$



Sur la gauche l'image '*peppers\_dx.png*' dérivée suivant x et sur la droite l'image '*peppers\_dy.png*' dérivée suivant y

On remarque ainsi que l'image dérivée suivant  $x$  a alors ses contours verticaux plus marqués étant donné que la dérivation s'est faite suivant l'axe de abscisses. Au contraire, sur l'image dérivée suivant  $y$ , les contours horizontaux sont plus marqués dû au fait que l'image a été dérivée suivant l'axe des ordonnées.

À partir des images `peppers_dx` et `peppers_dy`, grâce aux menus **Process → Math** et **Process → Image Calculator...**, calculer en chaque pixel la norme du gradient dans une nouvelle image (à renommer `peppers_gradNorm` grâce au menu **Image → Rename...**).

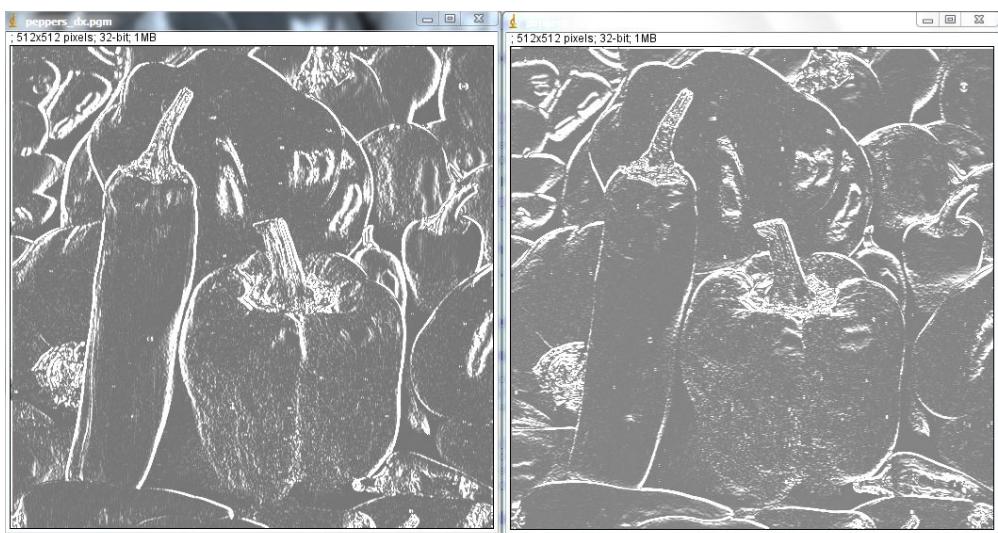
Suite à la dérivation de notre image suivant  $x$  et suivant  $y$ , nous avons créé une image en se basant sur le calcul de la norme du gradient pour chaque pixel. Pour rappel, la norme du gradient d'un pixel est calculé de la manière suivante :

$$|\delta| = \sqrt{\delta x^2 + \delta y^2}$$

Afin de pouvoir appliquer la norme du gradient à chaque pixel, il nous a alors fallu effectuer plusieurs étapes. Nous avions avant de faire ces étapes l'image de base dérivée suivant  $x$  et la même image dérivée suivant  $y$ . Nous avions donc les images correspondantes à  $\partial x$  et  $\partial y$ .

### Étape 1 :

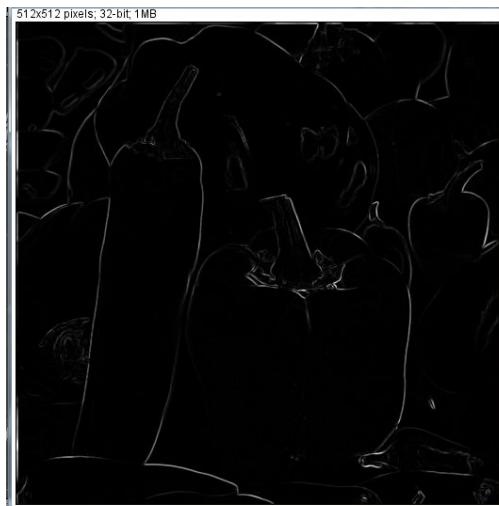
Dans cette étape, nous avons mis au carré chaque pixel de chacune des deux images afin d'avoir les images correspondantes à  $\partial x^2$  et  $\partial y^2$ . Nous avons effectué cela en appliquant la commande '**Process -> Math -> Square**' sur chacune des images. Voici le résultat alors obtenu à cette étape:



Sur la gauche l'image '`peppers_dx.png`' au carré et sur la droite l'image '`peppers_dy.png`' au carré

## Étape 2 :

Dans cette deuxième étape, nous avons additionné les deux images précédentes afin d'avoir l'image correspondant à  $\partial x^2 + \partial y^2$ . Nous avons effectué cela en appliquant la commande '**Process -> Image Calculator**'. Voici le résultat obtenu:



*Addition des 2 images précédentes = Addition de l'image 'peppers\_dx.png' au carré et de l'image 'peppers\_dy.png' au carré*

## Étape 3 :

Dans cette dernière étape, nous avons calculé la racine carrée de chaque pixel suivant l'image précédente. Nous avons alors obtenu l'image correspondant à  $\sqrt{\partial x^2 + \partial y^2}$  donc qui correspond à l'application de la norme du gradient en chaque point. Nous avons effectué cela en appliquant la commande '**Process -> Math -> Square Root**'. Puis, nous avons renommé cette image en '**peppers\_gradNorm**' grâce à la commande '**Image -> Rename**'. Voici le résultat obtenu:



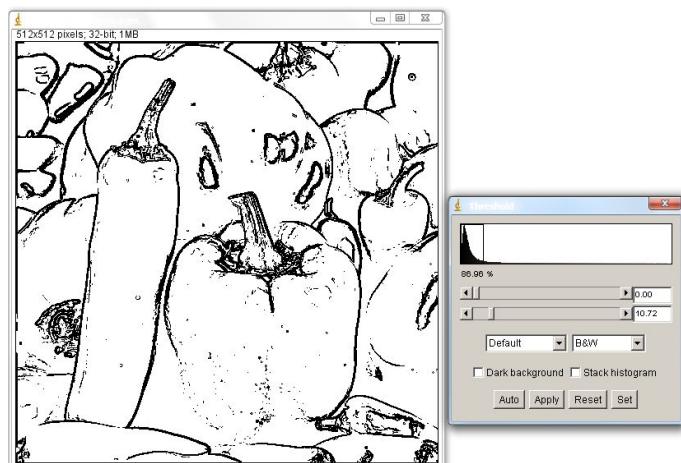
*Image 'peppers\_gradNorm' = Racine carrée de l'image précédente*

Étant donné que l'image est sur 32-bits, l'image finale est très sombre. Cela est dû au fait que le logiciel ne sait pas où est le noir et où est le blanc étant donné que les valeurs sont sur 32 bits.

Grâce au menu **Image → Adjust → Threshold...**, seuiller l'image de la norme du gradient. Est-il possible de trouver un seuil global pour l'image mettant en évidence les pixels contours de manière satisfaisante ?

Suite à l'application de la norme du gradient sur notre image, nous avons seuillé cette image grâce à la commande '**Image → Adjust → Threshold**'. Avant d'effectuer cette étape, il nous a fallu mettre la valeur max de l'image à 255 afin de pouvoir être plus précis dans le seuillage que nous allions faire. Nous avons effectué cela grâce à la commande '**Process -> Math -> Max...**'.

Il n'est pas possible de trouver un seuillage qui permet de ne voir que tous les contours. Cependant, il est possible d'en trouver un où la plupart des contours sont visibles et avec peu de bruit. Pour notre image, nous avons trouvé un seuillage à **10.72** qui nous semble satisfaisant. Voici le résultat obtenu :



*Image 'peppers\_gradNorm' seuillée avec le seuillage*

Avec l'image obtenue ci-dessus, nous avons créé un masque en appliquant notre seuillage grâce au bouton '**Apply**' et en décochant '**Set background pixels to NaN**'.



*Masque défini suivant l'image 'peppers\_gradNorm' seuillée*

Nous avons alors obtenu l'image précédente avec inversion du noir et du blanc. Cela nous permet ainsi d'avoir un masque.

### **3ème partie : Tatouage par étalement de spectre**

Lancer l'insertion (« embed ») d'un message pour une image. La macro vous demande alors de sélectionner une image hôte puis tatoue cette image. Que constate-t-on sur l'image tatouée 'WmkImage' ? Enregistrer cette image au format PGM. Faire varier le PSNR cible et trouver le PSNR permettant une imperceptibilité (pour vous) du signal de tatouage.

Afin de commencer la procédure d'encodage, nous choisissons de travailler sur l'image '*lena\_512.png*'.

#### Étape 1 :

La première étape consiste à choisir l'image sur laquelle nous allons travailler. En l'occurrence il s'agit de l'image '*lena\_512.png*'.



*Image de référence que nous avons choisi pour la procédure d'encodage*

#### Étape 2 :

La deuxième étape consiste à créer chaque tuile contenant la clé secrète.



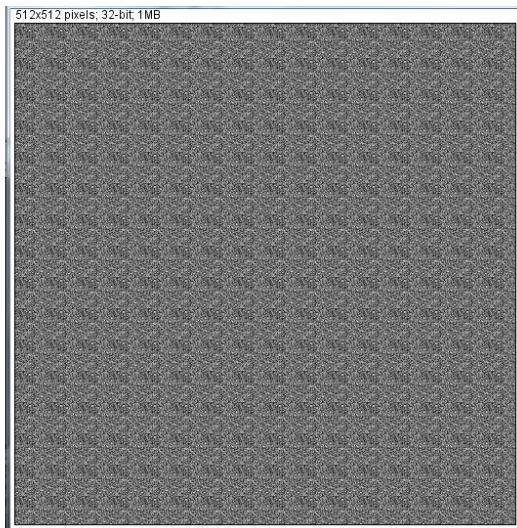
*Les 16 tuiles "secrètes" générées à partir de l'image et du message secret*

On remarque qu'il y a pour cette image 16 tuiles "secrètes" de 32x32 pixels chacunes. Cela correspond au fait que l'on souhaite forcément avoir des tuiles de cette taille

(32x32) et que notre image de base a une taille de 512x512. Or  $512 / 32 = 16$ , soit les 16 tuiles que l'on retrouve.

### Étape 3 :

Dans cette étape, il s'agit d'assembler nos 16 tuiles entre elles afin de former une image contenant le message secret.



*Les 16 tuiles secrètes générées assemblées entre elles*

Le résultat nous fournit une image de taille correspondante à la taille de l'image de base, soit 512x512 pixels.

### Étape 4 :

La dernière étape consiste à tatouer l'image.

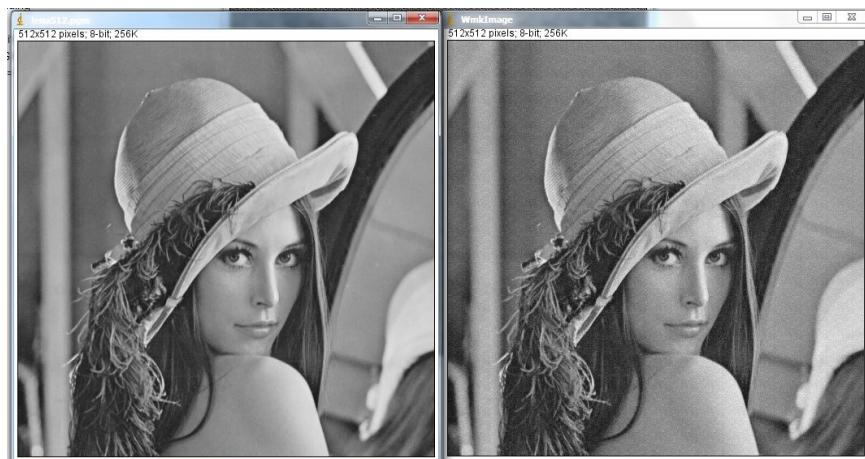


*Image de référence tatoué avec notre message*

Le résultat obtenu est notre image de base tatouée suivant l'image obtenue à l'étape précédente.

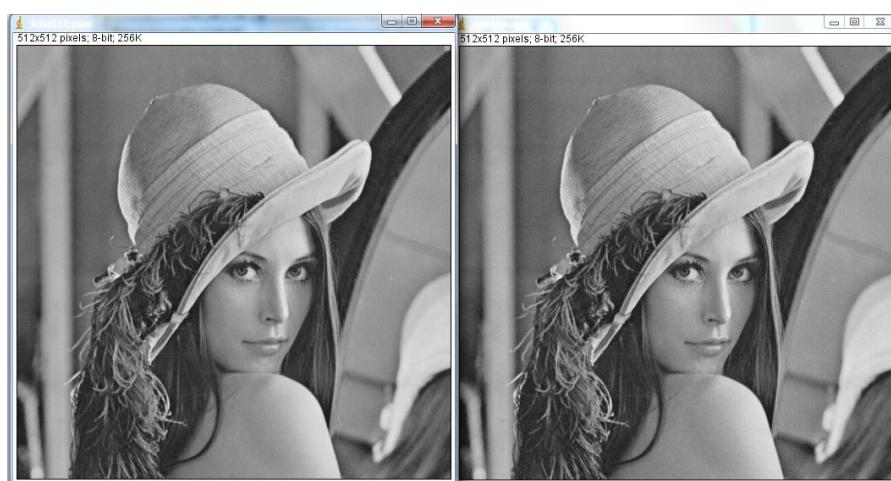
On constate que l'image 'WmkImage' est assez différente de l'image de base. En effet, le motif des tuiles est visible partout sur l'image. Le tatouage est donc très visible ce qui est embêtant car cela n'est alors pas discret. Ce tatouage ne respecte donc pas la contrainte de distorsion / imperceptibilité qui correspond au fait que l'utilisateur ne doit pas remarquer la présence du tatouage.

Afin de régler ce problème, nous avons relancé le programme en changeant/augmentant la valeur du PSNR souhaité afin que l'image obtenue corresponde mieux à celle de base . Nous avons tout d'abord testé en mettant cette valeur à 30 (contre 20 précédemment).



*Image de 'lena512.pgm' et image de 'lena512.pgm' tatouée avec un PSNR entre les deux images de 30*

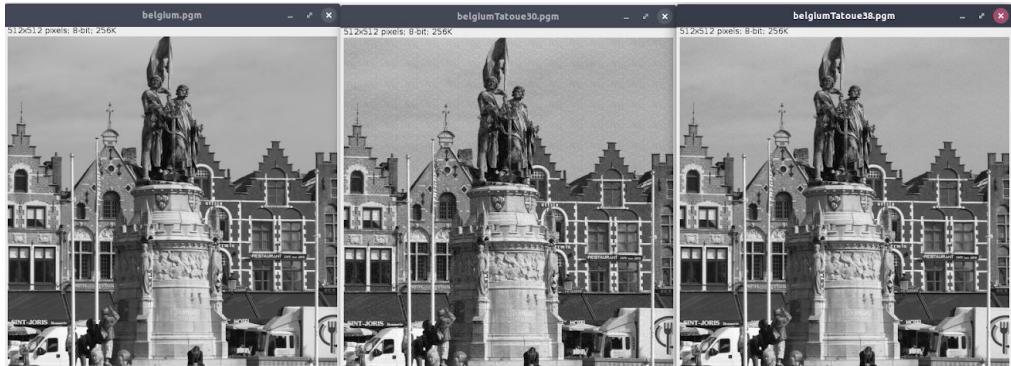
Nous remarquons, sur cette nouvelle image tatouée, que le tatouage est moins visible que précédemment. Cependant, il est encore bien visible à l'oeil nu et ne respecte donc toujours pas la contrainte d'imperceptibilité. Nous avons donc refait la même manipulation en augmentant encore le PSNR et en le mettant à 40.



*Image de 'lena512.pgm' et image de 'lena512.pgm' tatouée avec un PSNR entre les deux images de 40*

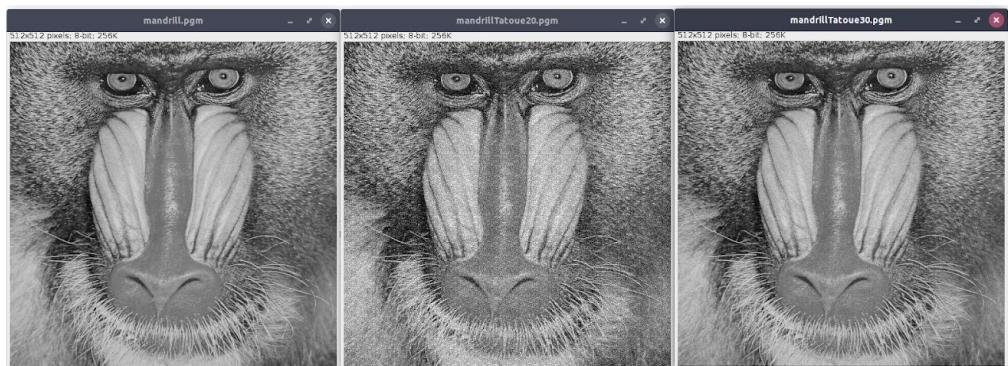
Sur cette nouvelle image tatouée avec un PSNR entre les deux images à 40, nous ne voyons plus vraiment le tatouage à l'oeil nu. Pour nous, ce PSNR permet donc une imperceptibilité du signal de tatouage pour cette image.

Nous avons testé sur d'autres images, en faisant également varier le PSNR afin d'avoir imperceptibilité du signal de tatouage. Nous avons testé sur les images 'belgium.pgm' et 'mandrill.pgm'. Ci-dessous vous trouverez les différents résultats obtenus.



*Image de 'belgium.pgm', image de 'belgium.pgm' tatouée avec un PSNR entre les deux images de 30 et image de 'belgium.pgm' tatouée avec un PSNR entre les deux images de 38*

On remarque alors que sur cette image, le PSNR qui permet une imperceptibilité du signal de tatouage à l'oeil nu est de 38. Ce PSNR est assez proche de celui trouvé pour 'lena512'. Cela est dû au fait que les deux images ont à peu près le même niveau de détails.



*Image de 'mandrill.pgm', image de 'mandrill.pgm' tatouée avec un PSNR entre les deux images de 20 et image de 'mandrill.pgm' tatouée avec un PSNR entre les deux images de 30*

On remarque alors que sur cette image, le PSNR qui permet une imperceptibilité du signal de tatouage à l'oeil nu est de 30. Ce PSNR est plus faible que précédemment. Cela s'explique par le fait que l'image de base est globalement assez semblable aux tuiles dû au fait que les poils du mandrill semblent comme pixelisés. C'est la même raison qui explique que sur l'image ayant un PSNR de 20 avec l'image de base, le signal du tatouage ne se voit pas sur les poils de l'animal mais seulement sur ses yeux et au niveau de son nez.

On peut ainsi remarquer que le PSNR qui permet une imperceptibilité du signal de tatouage à l'oeil nu dépend en partie de l'image de base. Avec une image qui ressemble plus à l'image obtenue avec les tuiles, le PSNR sera en effet plus faible.

## Détailler les instructions de la fonction makeWmkSignal.

Analysons, la fonction ‘makeWmkSignal’ qui crée l’image contenant le signal à ajouter à l’image. Cette fonction prend en paramètre l’id de l’image correspondant à la tuile (*IDcarriers*) les dimensions de l’image à tatouer (*widthImg* et *heightImg*) ainsi que le message à tatouer (*msg*).

Tout d’abord, la fonction sélectionne l’image dont l’identifiant est en paramètre (*selectImage(IDcarriers)*). Puis, elle appelle la fonction *getDimensions* afin d’enregistrer dans nos variables la largeur (*widthTile*) et la hauteur (*heightTile*) de la tuile. Le nombre de bits du message (*Nc*). Les paramètres *frames* et *channels* correspondent à des valeurs spécifiques de l’image.

Suite à cela, une image dont les pixels sont sur 32-bits est créée. C’est une image noire qui fait les mêmes dimensions que ceux entrés en paramètres (= dimensions de l’image à tatouer). Cette image est nommée ‘WmkSignal’. On enregistre ensuite son id dans une variable (*IDWmk = getImageID()*).

Suite à cela, une image dont les pixels sont sur 32-bits est créée. C’est une image noire dont les dimensions sont celles de la tuile. Cette image est nommée ‘WmkTile’. On enregistre ensuite son id dans une variable (*IDWmkTile = getImageID()*).

On fait ensuite une boucle de la longueur du message et pour chaque tour, on :

- Sélectionne notre image entrée en paramètre (= la tuile) (*selectImage(IDcarriers)*)
- Définit une portion spécifique de l’image (*setSlice(i+1)*)
- Si le bit du message qu’on traite est égal à ‘1’ alors on multiplie la portion définie par *setSlice(...)* par -1 (*run("Multiply...", "value=-1 slice")*)
- On additionne alors les images correspondant à la tuile entrée en paramètre et celle qui comporte le message (*imageCalculator("Add 32-bit", IDWmkTile, IDcarriers)*)

Cette boucle permet donc de créer la tuile avec le message inséré.

On sélectionne ensuite l’image correspondant à la tuile (*selectImage(IDWmkTile)*), puis on réinitialise son min et son max (*resetMinAndMax()*).

On crée alors un rectangle de même dimension que notre tuile. Puis nous copions notre tuile. Puis, on désélectionne tout ce qui est sélectionné.

On sélectionne, ensuite, notre image créé précédemment qui est aux dimensions de l’image à tatouer (*selectImage(IDWmk)*). Puis, nous parcourons cette image en largeur et en hauteur en avançant par pas correspondant à la largeur (respectivement en hauteur) de la tuile et nous créons alors un rectangle correspondant aux dimensions de la tuile où nous copions la tuile précédemment copié. Cette étape permet de répéter la tuile le nombre de fois nécessaire afin qu’elle soit présente partout sur l’image à tatouer.

Puis on désélectionne tout et pour finir, on retourne l’id de l’image aux dimensions de l’image à tatouer et qui contient notre tuile dupliquée (*return IDWmk*).

Coder la fonction bitErrorRate utilisée pour le décodage du message (cette fonction calcule le BER entre le message original et le message décodé) puis lancer le décodage d'une image tatouée pour vérification (« decode ») (la macro vous demande alors de sélectionner l'image à décoder).

Afin de quantifier la robustesse, il nous faut calculer le taux d'erreur binaire (**Bit Error Rate**). Pour rappel, le BER entre un message  $m$  et un message  $m'$  est défini grâce à la formule ci-dessous :

$$\text{BER} = \frac{dH(m, m')}{N_c}$$

où

- $N_c$  est le nombre de bits du message
- $dH$  est la distance de Hamming défini par  $dH(m, m') = \text{card}(\{m(i) \neq m'(i)\})$ , i.e. le nombre de bits différents entre deux vecteurs de bits.

Nous avons grâce à cela codée la fonction ‘bitErrorRate’ qui calcule le BER entre le message original et le message décodé. Voici le code :

```
function bitErrorRate(msg, msgEst){
    nbDifferentBits = 0;
    // calcul de la distance de Hamming
    for(i = 0; i < Nc; i++){
        if(msg[i] != msgEst[i]){
            nbDifferentBits++;
        }
    }
    if(nbDifferentBits == 0){
        print("Tous les bits sont identiques !");
        return -1;
    }
    // calcul du BER
    return nbDifferentBits / Nc;
}
```

*Code permettant de calculer le Bit Error Rate (BER) entre le message original et le message décodé*

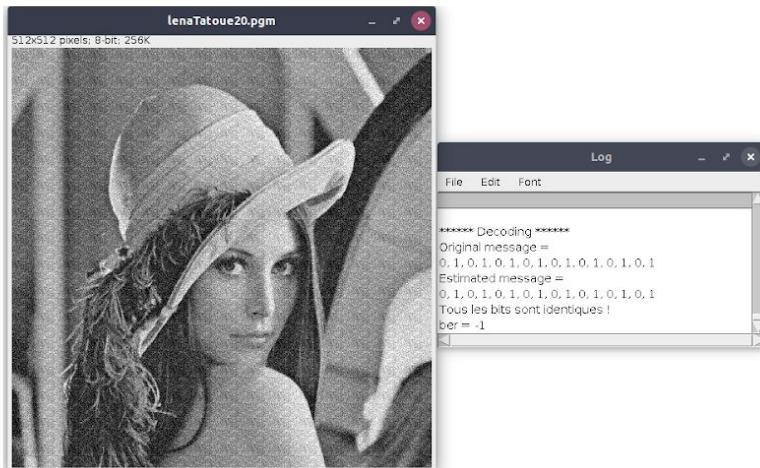
La fonction prend en paramètre les deux messages.

La première étape consiste à calculer la distance de Hamming. Pour faire cela, nous effectuons une boucle qui pour chaque bit du message original le compare avec celui du message décodé. Une variable est initialisée avant cette boucle à 0 et incrémentée de 1 à chaque fois que les bits sont différents. Elle nous permet donc d'avoir la distance de Hamming à la fin de cette boucle.

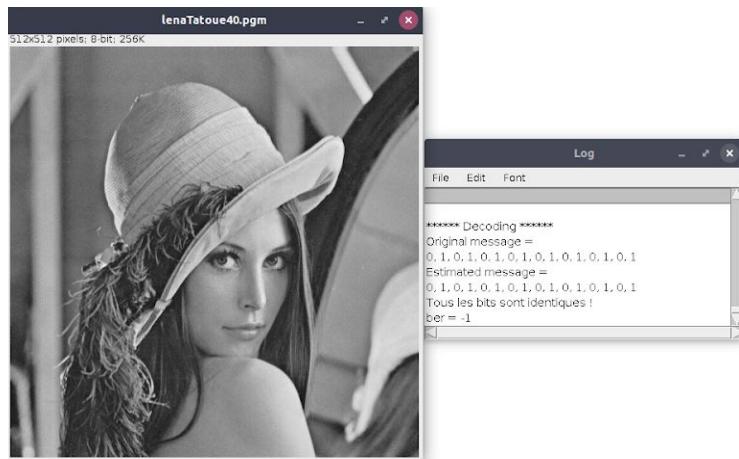
Si cette variable est à 0 à la fin de cette boucle, cela signifie que le message original et décodé sont identiques. cela signifie que le message est robuste.

Sinon, on retourne la valeur de cette variable divisée par le nombre de bits du message afin de pouvoir quantifier la robustesse.

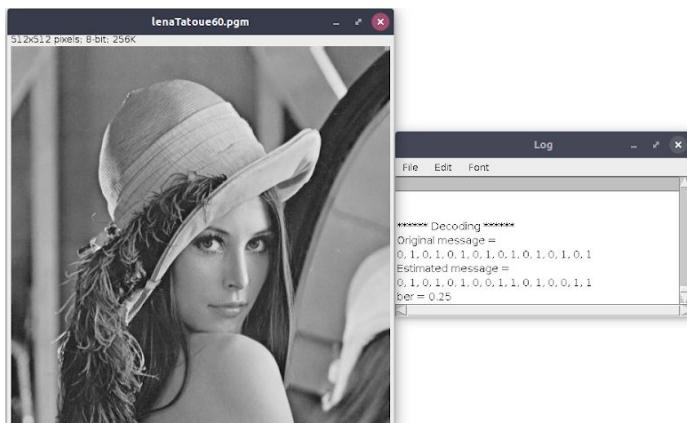
Nous avons testé notre fonction sur différentes images tatouées. Nous l'avons tout d'abord testé sur différentes images de lena tatouée suivant un PSNR avec l'image de base différent.



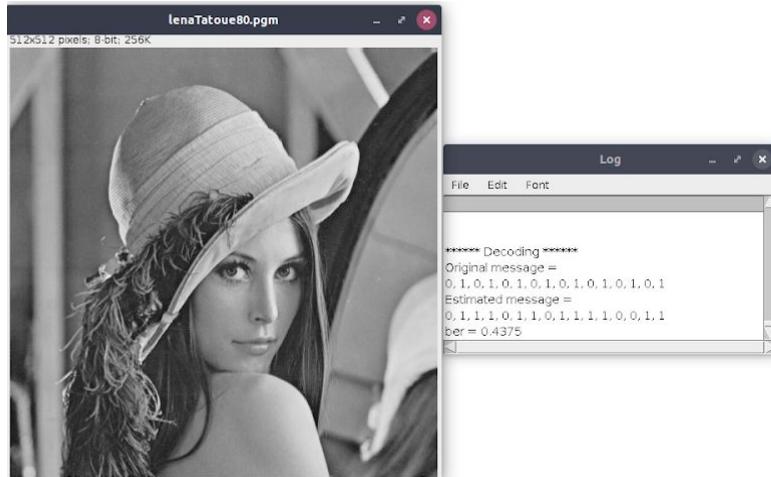
*Image de lena tatouée avec un PSNR avec l'image de base à 20 ainsi que le message original, le message décodé et le BER*



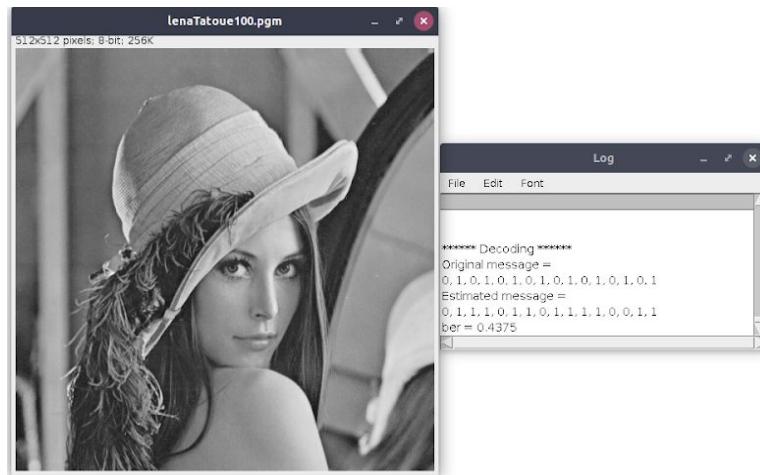
*Image de lena tatouée avec un PSNR avec l'image de base à 40 ainsi que le message original, le message décodé et le BER*



*Image de lena tatouée avec un PSNR avec l'image de base à 60 ainsi que le message original, le message décodé et le BER*

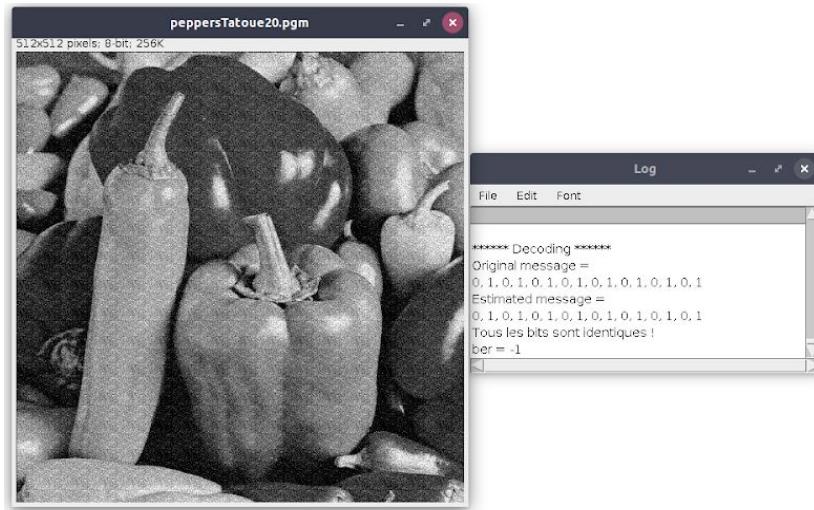


*Image de lena tatouée avec un PSNR avec l'image de base à 80 ainsi que le message original, le message décodé et le BER*

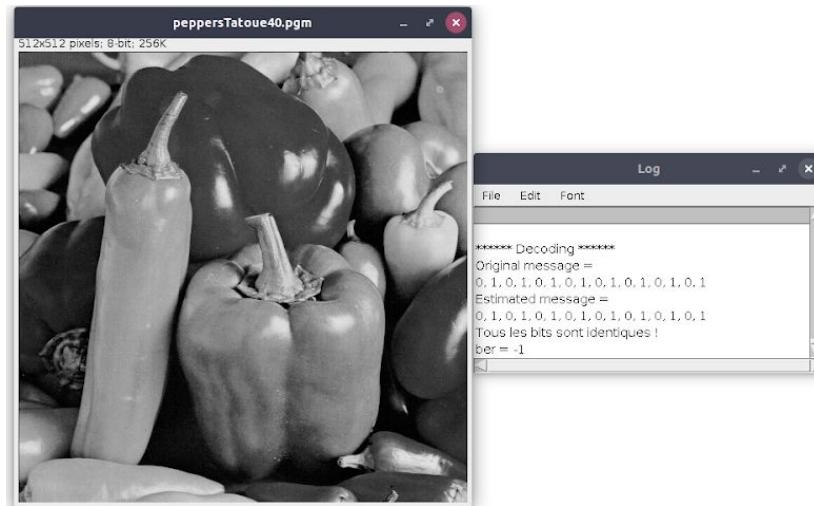


*Image de lena tatouée avec un PSNR avec l'image de base à 100 ainsi que le message original, le message décodé et le BER*

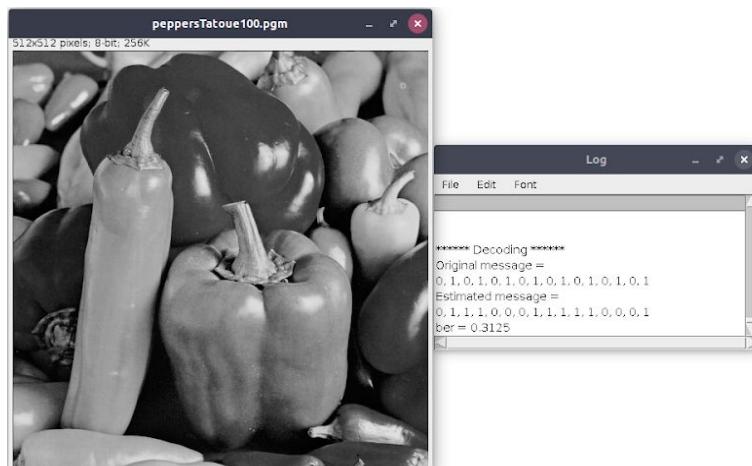
Grâce à ces différents exemples, on remarque que plus le PSNR avec l'image de base est élevé plus le message décodé est différent du message original, donc plus le BER est élevé. Cela est logique car la valeur du PSNR influe sur la valeur de l'Erreur Quadratique Moyenne (**EQM**). Plus la valeur du PSNR sera élevée, plus l'EQM sera élevée aussi et donc plus le message décodé sera différent du message original. Nous avons ensuite fait la même chose mais avec une autre image qui est 'peppers.pgm'.



*Image ‘peppers’ tatouée avec un PSNR avec l'image de base à 20 ainsi que le message original, le message décodé et le BER*



*Image ‘peppers’ tatouée avec un PSNR avec l'image de base à 40 ainsi que le message original, le message décodé et le BER*



*Image ‘peppers’ tatouée avec un PSNR avec l'image de base à 100 ainsi que le message original, le message décodé et le BER*

Grâce à ces différents exemples que l'on peut comparer à ceux obtenus avec l'image de lena, on remarque que pour un même PSNR entre l'image de base est celle tatoué, le BER n'est pas nécessairement le même. Cela s'explique par le fait que le niveau de détail de l'image originale importe beaucoup lors de l'encodage d'un message et que d'une image à une autre, le PSNR cible impliquant l'imperceptibilité du tatouage peut être différent.

Créer une fonction `makeGradientNorm` qui prend en entrée l'identifiant d'une image, crée une image contenant la norme du gradient de Sobel et retourne son identifiant. Modifier ensuite la partie « `embed` » afin de prendre en considération ce masquage psychovisuel lors de l'insertion du tatouage. Que constate-t-on ? Faire varier le PSNR cible et trouver le PSNR permettant une imperceptibilité (pour vous) du signal de tatouage.

Afin de faire la fonction ‘`makeGradientNorm`’ qui prend en entrée l’identifiant d’une image, crée une image contenant la norme du gradient de Sobel et retourne son identifiant, nous nous sommes aidés de la macro que nous avions enregistré lors de la partie 2 de ce TP, les étapes effectuées dedans sont donc les mêmes que pour cette partie (vous pourrez trouver le code correspondant en **Annexe 1**).

Il faut ensuite modifier la partie “`embed`” afin qu’elle prenne en compte le masquage psychovisuel. Nous faisons donc appel à la fonction `makeGradientNorm`, qui prend en paramètre l’ID de l’image que l’on va normer, en l’occurrence il s’agit de `IDHost`, que nous avons créée ci-dessus afin d’affecter ce masque à notre image ‘`peppers.pgm`’ (vous pourrez trouver le code correspondant en **Annexe 2**). Nous multiplions ensuite ces deux images afin d’y intégrer le tatouage selon la norme. Pour un PSNR entre l’image de base et l’image résultante cible de 20, nous obtenons le résultat suivant :

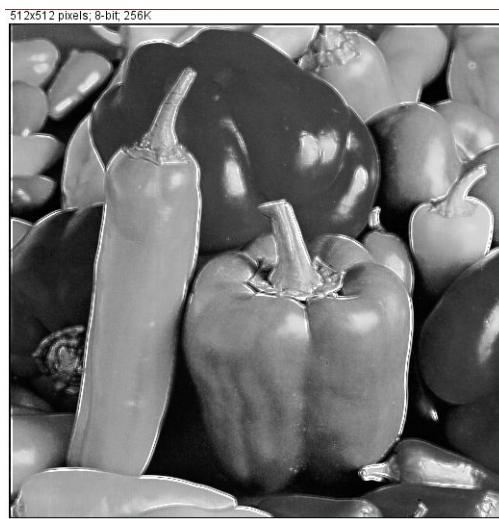
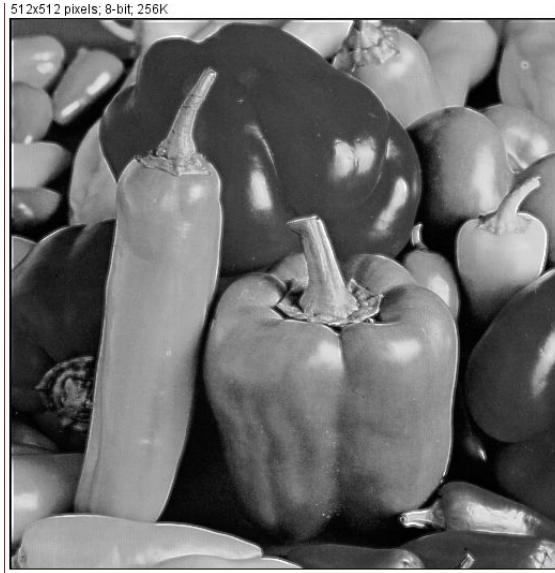


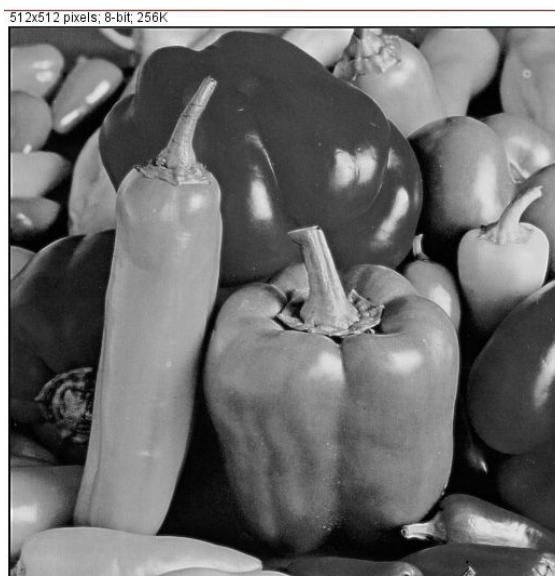
Image ‘peppers’ contenant la norme du gradient de Sobel avec un PSNR de 20

Nous pouvons constater que l’image dans sa majeure partie est très ressemblante par rapport à l’image de base, seuls les contours sont vraiment marqués et trahissent l’imperceptibilité. Nous avons alors effectué le même processus avec un PSNR cible de 30 :



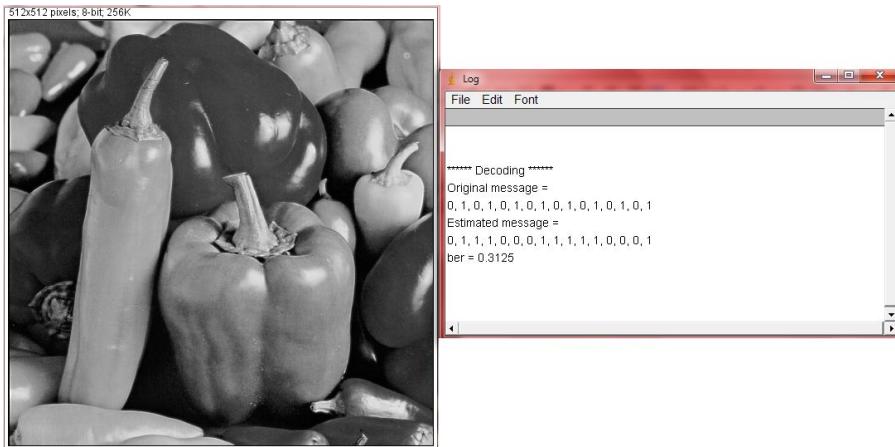
*Image ‘peppers’ contenant la norme du gradient de Sobel avec un PSNR de 30*

Cette fois-ci, les contours sont beaucoup moins marqués mais toujours visibles. Il faut encore augmenter le PSNR cible afin que l'imperceptibilité soit totale pour l'oeil humain. Nous ré-effectuons alors la manipulation avec un PSNR cible de 40 :



*Image ‘peppers’ contenant la norme du gradient de Sobel avec un PSNR de 40*

Les contours ne sont plus marqués, le tatouage est donc devenu imperceptible pour l'oeil humain. Nous avons ensuite évalué le BER de cette image.

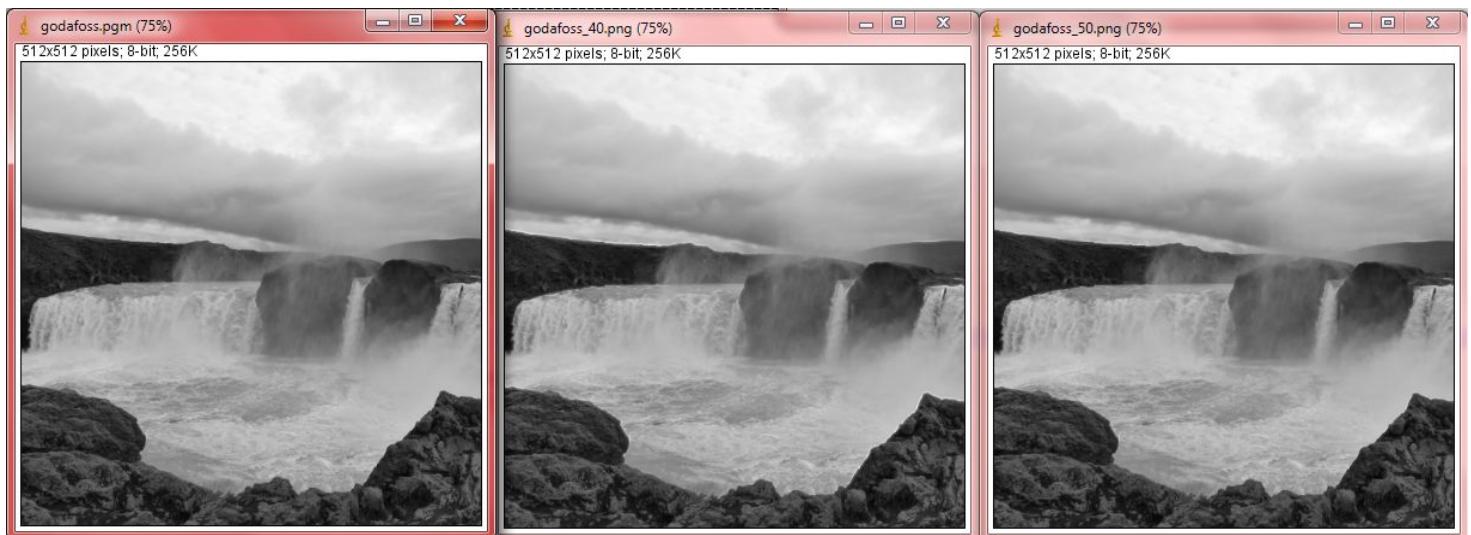


*Image ‘peppers’ tatouée avec un PSNR de 40 et son décodage associé*

Nous constatons alors que le message original et le message décodé sont différents. Pour rappel, pour le même PSNR cible (=40) sans la norme du gradient de Sobel, nous avions pas de bit de différence entre le message original et le message cible. Cela s'explique par le fait qu'en ajoutant la norme du gradient de Sobel, nous avons ajouté du bruit.

Pour la suite du TP, nous conservons cette image avec un BER de base de 0.3125, il faudra donc en tenir compte pour les manipulations que nous effectuerons par la suite.

Afin de mieux comprendre l'impact de l'ajout de la norme du gradient de Sobel, nous avons effectués la même procédure sur d'autres images. La première est ‘godafoss.pgm’.



*Image ‘godafoss.pgm’, ‘godafoss.pgm’ tatouée avec PSNR entre l'image de base et l'image obtenue de 40 et ‘godafoss.pgm’ tatouée avec PSNR entre l'image de base et l'image obtenue de 50*

Dans le cas de ‘godafoss.pgm’, il est plus difficile de retrouver une image avec un tatouage imperceptible car les contours présents sont très marqués (frontière entre les rochers et l'eau). En effet, il y a une grande différence de couleur entre les rochers sombres et l'eau claire. On a ainsi pu remarquer que plus les contours sont marqués, plus le PSNR requis pour obtenir l'imperceptibilité du tatouage doit être élevé.

Prenons maintenant l'exemple d'une image dont les contours sont beaucoup moins forts. Il s'agit de l'image 'mandrill.pgm'. Voici les résultats obtenus :

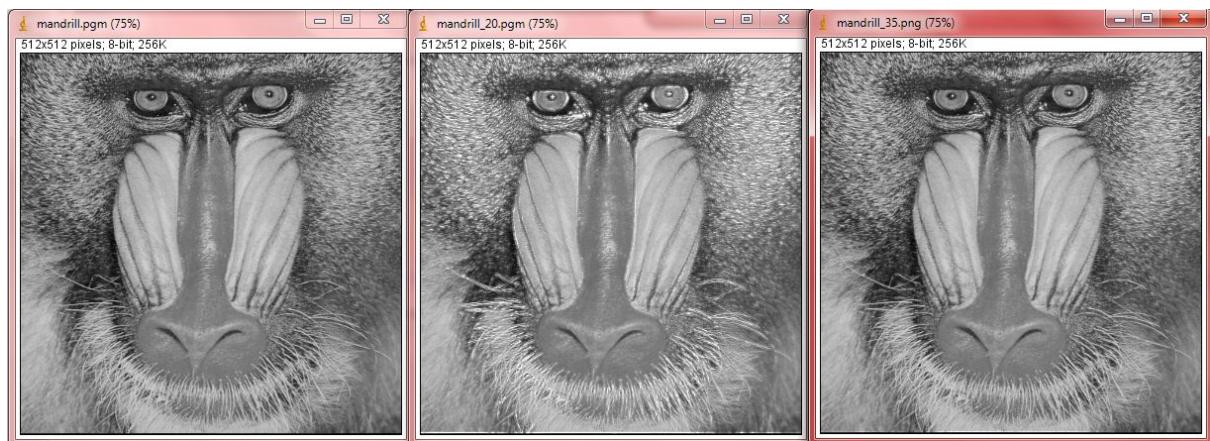


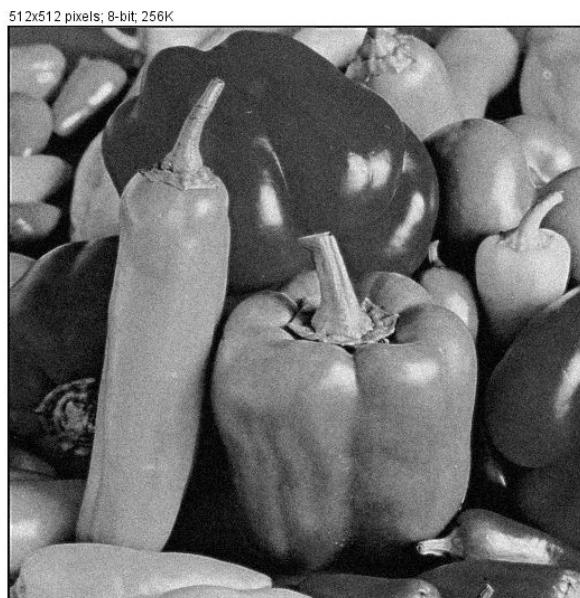
Image 'mandrill.pgm', 'mandrill.pgm' tatouée avec PSNR entre l'image de base et l'image obtenue de 20 et 'mandrill.pgm' tatouée avec PSNR entre l'image de base et l'image obtenue de 35

Les contours de l'image 'mandrill.pgm' sont beaucoup moins marqués que sur les images 'peppers.pgm' et 'godafoss.pgm'. Le PSNR requis pour garantir l'imperceptibilité du tatouage est donc inférieur aux deux images précédentes puisqu'il est de 35. Les seuls contours forts que nous pouvons retrouver ici sont au niveau des yeux, c'est à dire une zone mineure par rapport à l'ensemble de l'image. Si on ne tenait pas compte de cette zone (les yeux), l'image ayant un PSNR cible de 20 nous permet une imperceptibilité du tatouage à l'oeil nu.

## 4ème partie : Robustesse du schéma de tatouage

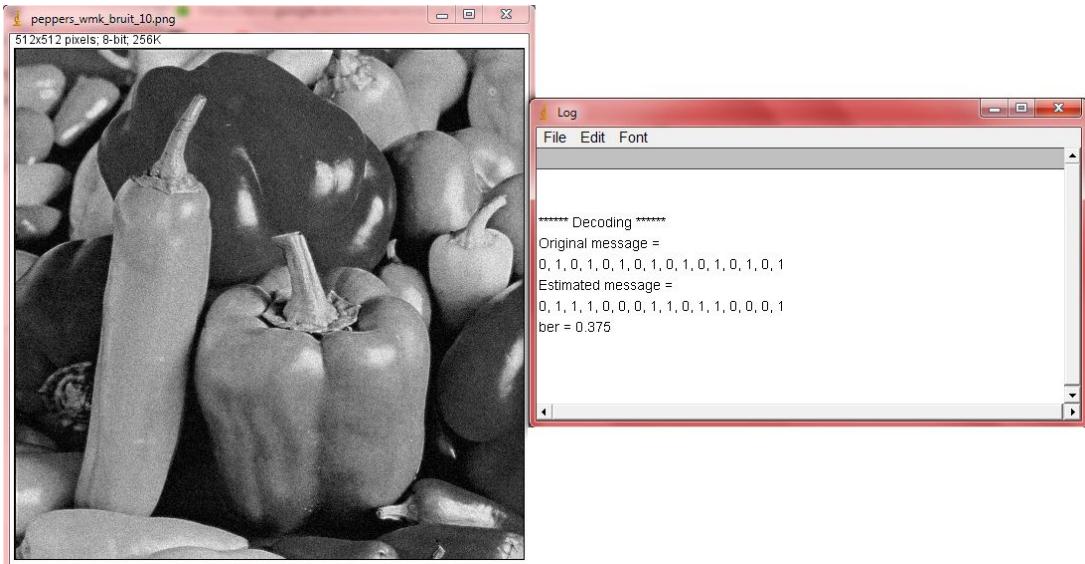
Après avoir tatoué une image, on lui ajoutera un bruit gaussien en spécifiant son écart-type via Process → Noise → Add Specified Noise... avant de procéder au décodage. Pour quel écart-type du bruit ajouté à l'image commence-t-on à obtenir des erreurs de décodage ? Ce schéma de tatouage est-il robuste à l'ajout de bruit ?

Dans un premier temps, nous tatouons notre image sur laquelle nous allons effectuer nos manipulations. L'image utilisée sera '*peppers.pgm*'. L'image résultante tatouée avec un PSNR cible de 40 sera appelée par la suite '*peppers\_wmk.pgm*'. Nous commençons par ajouter un bruit gaussien sur l'image '*peppers\_wmk.pgm*' grâce à la commande **Process** → **Noise** → **Add Specified Noise...** avec un écart type de 10. Nous obtenons le résultat suivant :



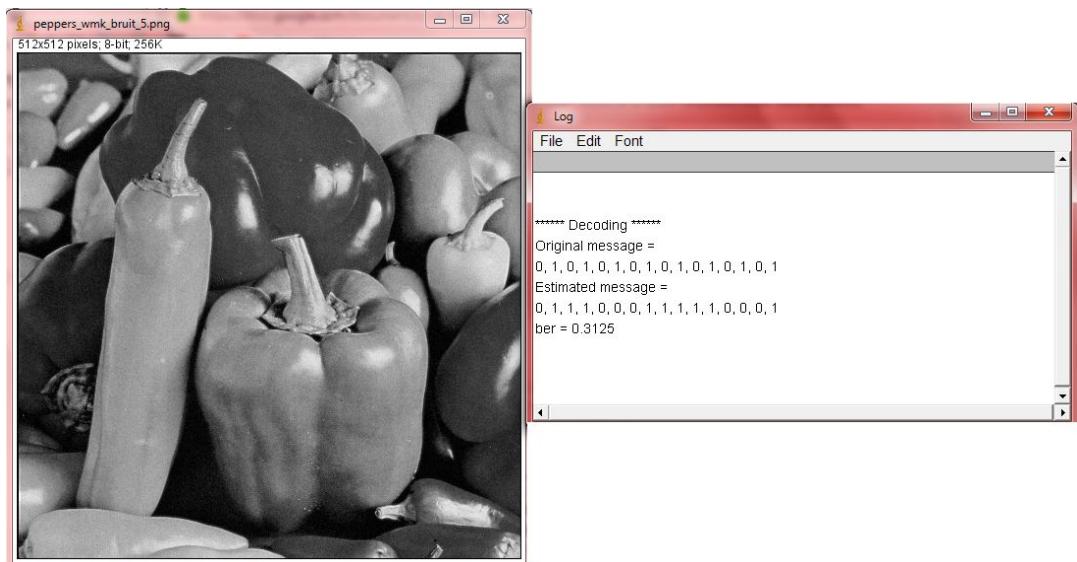
*Image 'peppers\_wmk' avec un bruit gaussien d'écart-type 10*

Nous décodons ensuite cette image affectée par un bruit gaussien afin de savoir si le message a réussi à être lu sans erreur. Nous obtenons le résultat suivant :



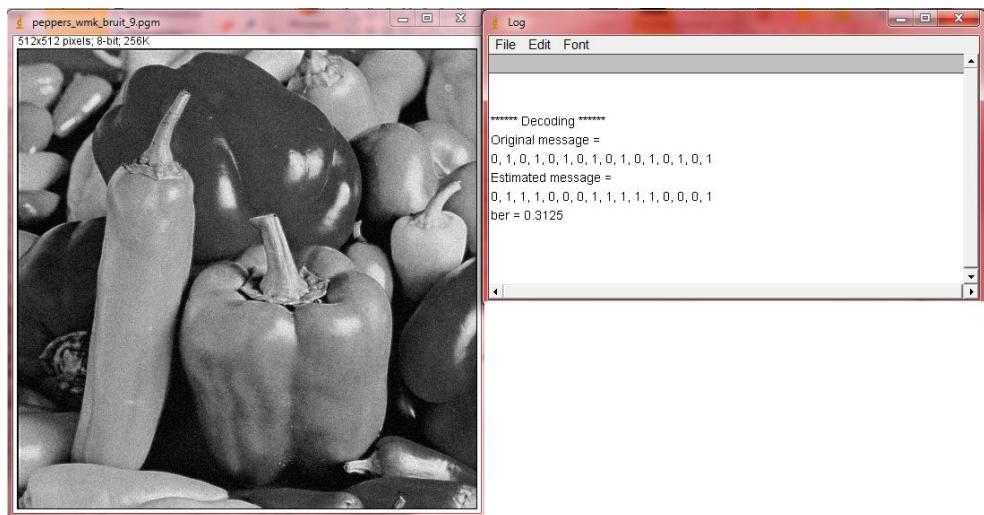
*Image ‘peppers\_wmk’ affectée par un bruit gaussien de valeur 10 ainsi que le message original, le message décodé et le BER*

Nous pouvons nous apercevoir qu'il y a plusieurs bits d'erreurs entre le message original et le message décodé. Cela est dû au fait que la valeur d'écart-type que nous avons indiqué (10) est trop élevée par rapport à la valeur du BER de base qui est de 0,3125. Essayons la même manipulation avec un bruit gaussien d'écart-type 5 :



*Image ‘peppers\_wmk’ affectée par un bruit gaussien de valeur 5 ainsi que le message original, le message décodé et le BER*

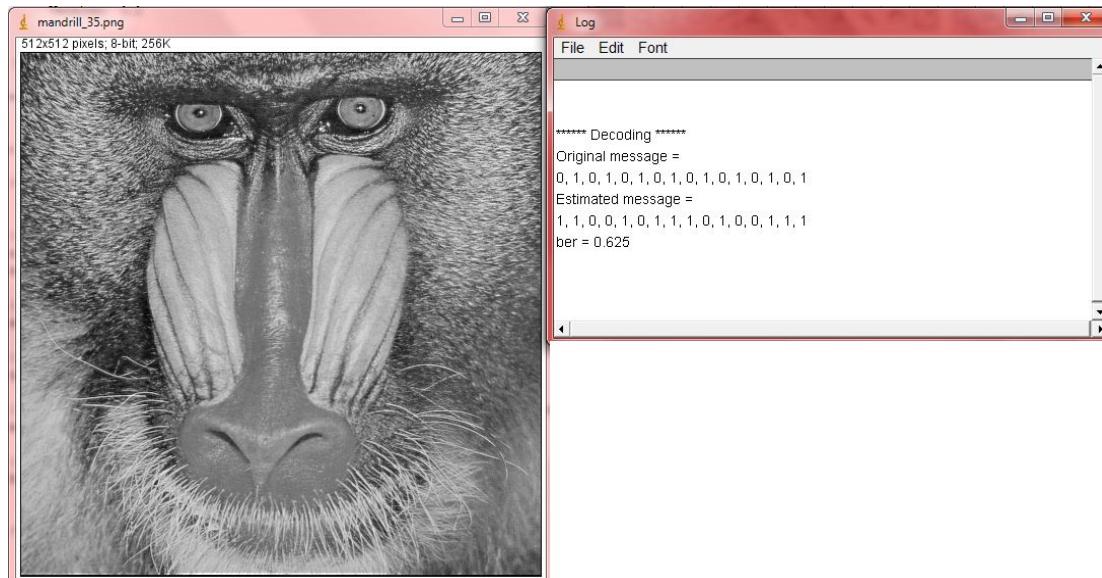
Nous essayons de trouver la limite, pour cela, nous ajoutons un bruit gaussien d'écart-type 9 à l'image ‘peppers\_wmk’ :



*Image ‘peppers\_wmk’ affectée par un bruit gaussien d’écart-type 9 ainsi que le message original, le message décodé et le BER*

Nous pouvons observer que lorsque notre image ‘peppers\_wmk’ subit un bruit gaussien d’écart-type 9, le message ne subit pas de nouvelles erreurs par rapport à l'image de base tatouée. Cela signifie que notre image tatouée est robuste au bruit gaussien lorsque son écart-type est inférieur à 10.

Nous effectuons les même manipulations sur l'image ‘mandrill.pgm’ sur laquelle nous avons ajouté un tatouage avec un PSNR cible de 35. Cette image, si on la décode, a un BER de 0,625 comme nous pouvons le voir sur l'image qui suit. Ce BER est dû à la norme du gradient de Sobel.



*Image ‘mandrill.pgm’ tatouée avec un PSNR de 35, son décodage associé ainsi que son BER (= 0,625)*

Nous ajoutons un bruit gaussien de valeur 10 et nous lançons la procédure de décodage afin de vérifier si le BER est modifié :

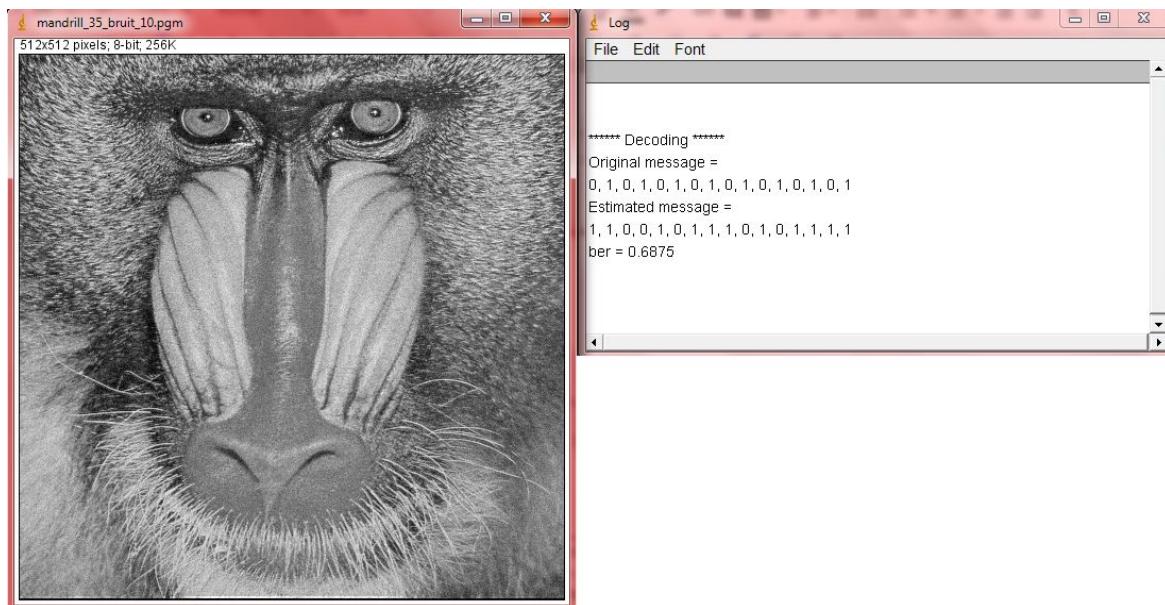


Image ‘mandrill.pgm’ tatouée avec un PSNR de 35 et affectée par un bruit gaussien d’écart-type 10, son décodage associé ainsi que son BER (= 0,6875)

Nous constatons que le BER de l'image ‘mandrill.pgm’ affectée par un bruit gaussien d'écart-type 10 est supérieur au BER précédemment obtenu. Cela signifie que notre image tatouée n'est pas robuste à un bruit gaussien de cette valeur. Essayons maintenant avec un bruit gaussien d'écart-type 8 :

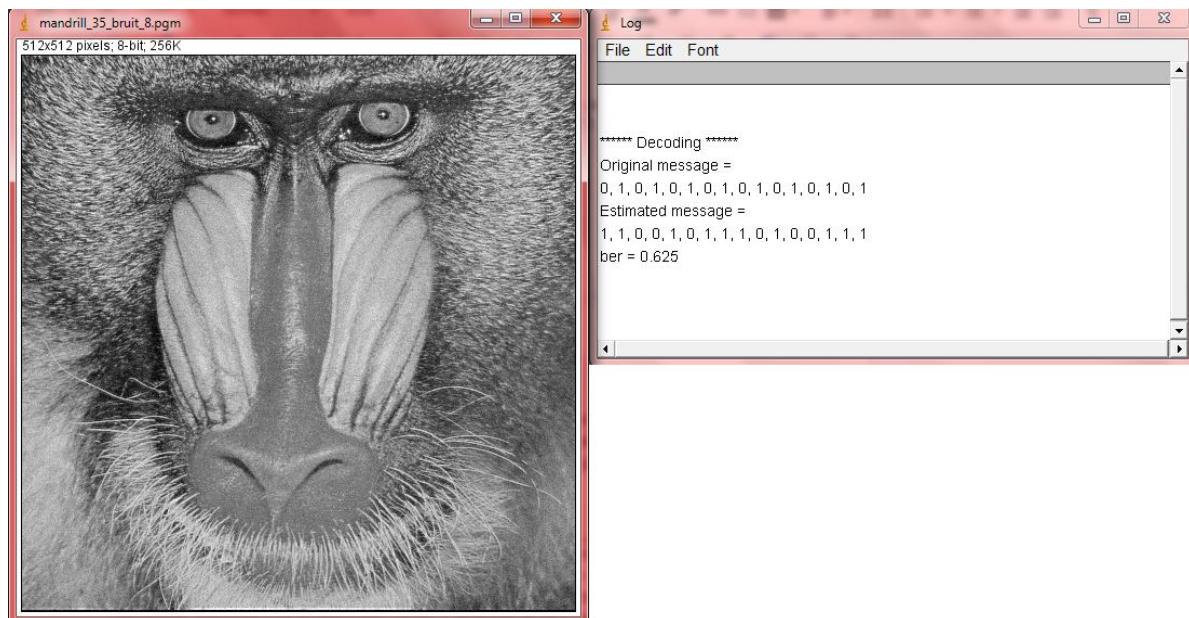
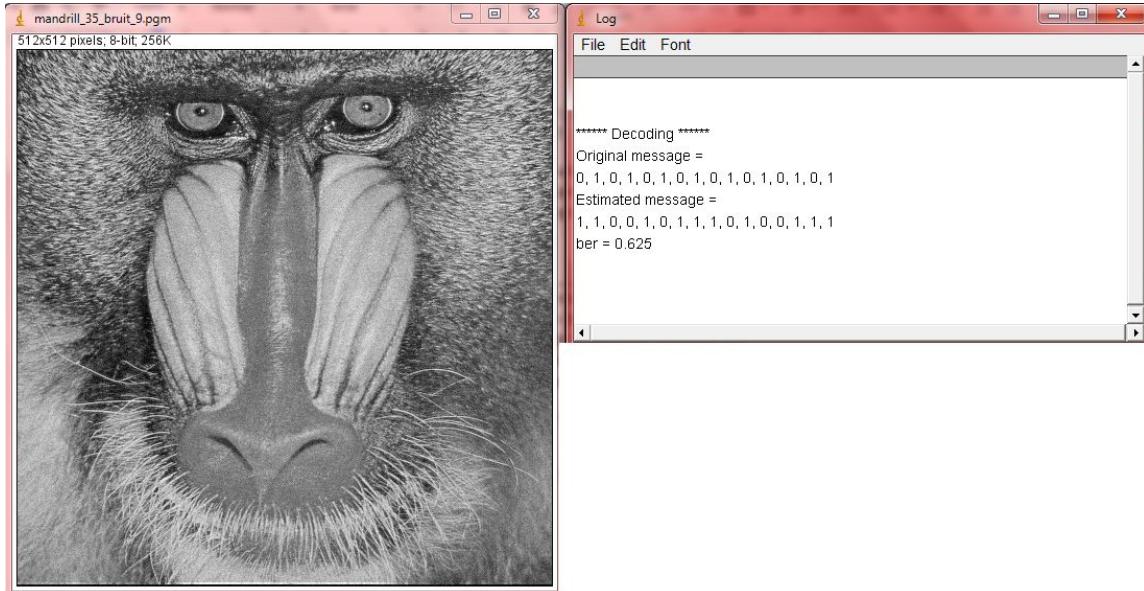


Image ‘mandrill.pgm’ tatouée avec un PSNR cible de 35 et affectée par un bruit gaussien d’écart-type 8, son décodage associé ainsi que son BER (= 0,625).

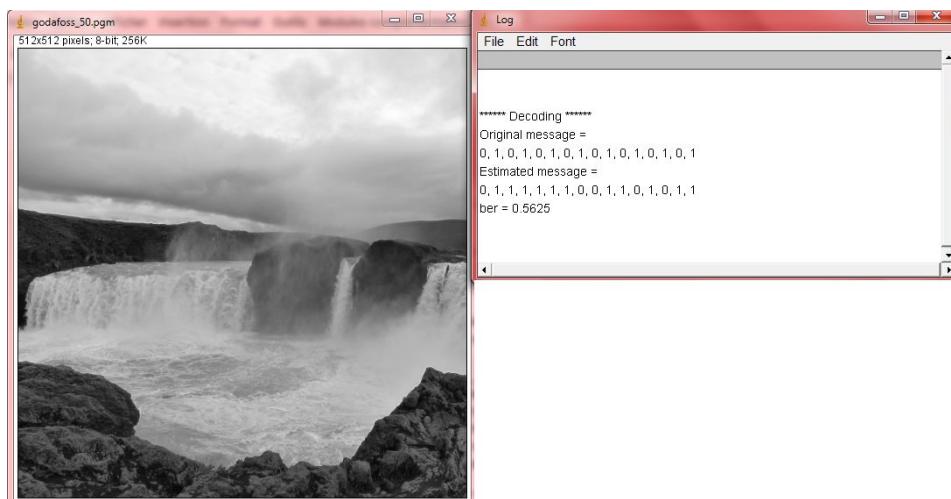
Cette fois-ci, le BER est le même que celui de l'image tatouée de base ce qui signifie que notre image n'est pas affectée par le bruit gaussien d'écart-type 8. Il ne nous reste plus qu'à faire la manipulation avec 9 afin de connaître la limite d'écart-type ajoutant des bits d'erreurs :



*Image ‘mandrill.pgm’ tatouée avec un PSNR cible de 35 et affectée par un bruit gaussien d'écart-type 9, son décodage associé ainsi que son BER (= 0,625).*

Le BER est le même que l'image tatouée de base. Nous pouvons donc conclure que l'image ‘*mandrill.pgm*’ tatouée avec un PSNR de 35 est robuste au bruit gaussien lorsque celui-ci possède une valeur d'écart-type inférieure à 10.

Nous effectuons les même manipulations sur l'image ‘*godafoss.pgm*’ sur laquelle nous avons ajouté un tatouage avec un PSNR de 50. Cette image, si on la décode, a un BER de 0,5625 (à cause du gradient de la norme de Sobel) comme nous pouvons le voir sur l'image qui suit :



*Image ‘godafoss.pgm’ tatouée avec un PSNR cible de 50, son décodage associé ainsi que son BER (= 0,5625)*

Nous ajoutons un bruit gaussien de valeur 20 et nous lançons la procédure de décodage afin de vérifier si le BER est modifié :

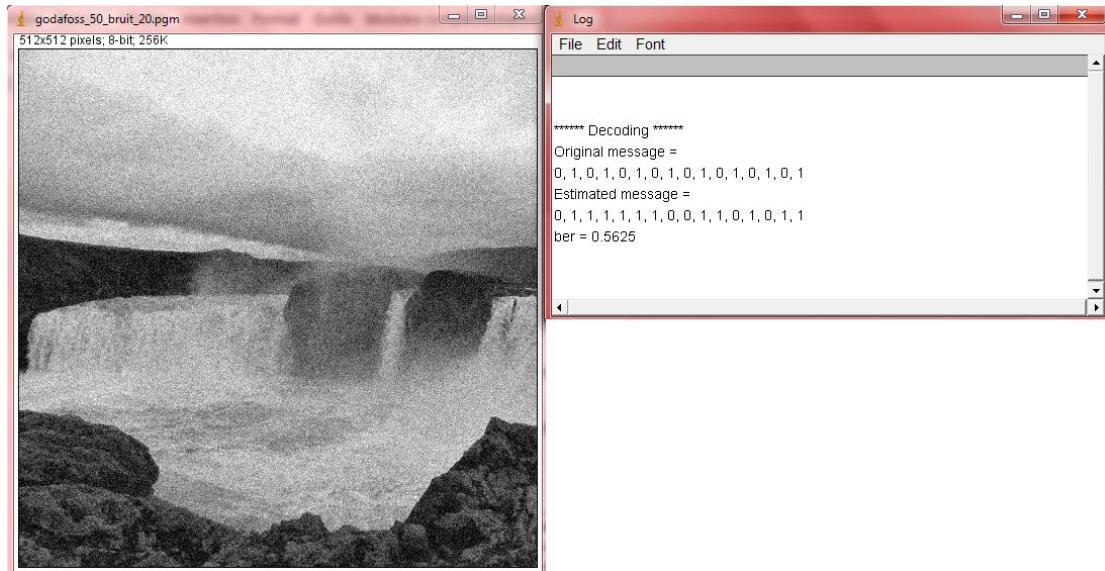


Image ‘godafoss.pgm’ tatouée avec un PSNR cible de 50 et affectée par un bruit gaussien d’écart-type 20, son décodage associé ainsi que son BER (= 0,5625).

Nous constatons que le BER est le même que celui de l'image tatouée de base ce qui signifie que notre image n'est pas affectée par le bruit gaussien d'écart-type 20. Nous testons maintenant un bruit gaussien d'écart-type 30 :

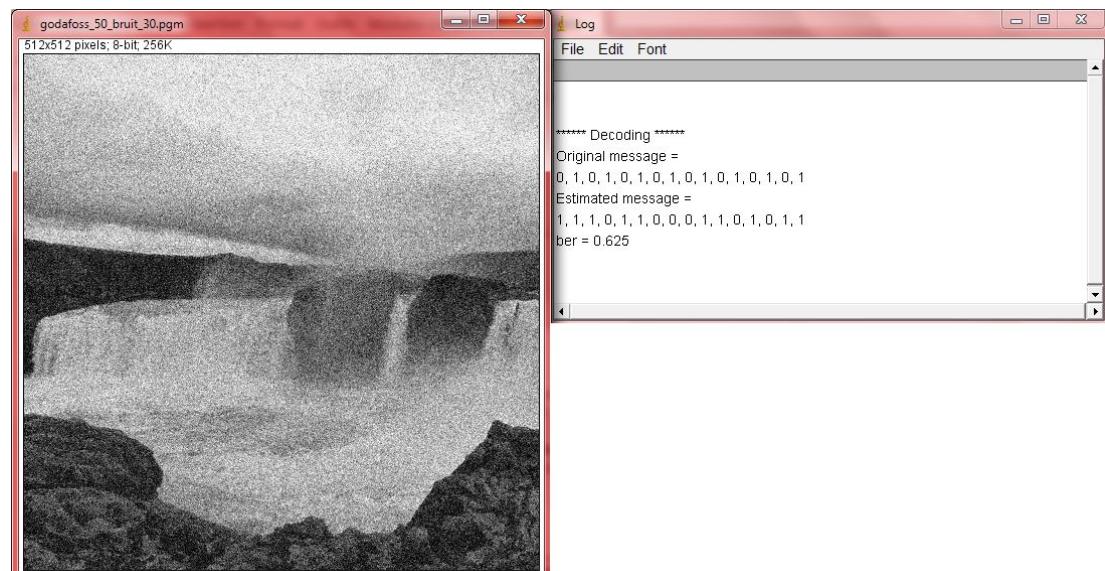


Image ‘godafoss.pgm’ tatouée avec un PSNR cible de 50 et affectée par un bruit gaussien d’écart-type 30, son décodage associé ainsi que son BER (= 0,625).

Nous constatons que le BER de l'image ‘godafoss.pgm’ affectée par un bruit gaussien d'écart-type 30 est différent ce qui signifie que notre image tatouée n'est pas robuste à un bruit gaussien de cette valeur. Essayons maintenant avec un bruit gaussien d'écart-type 29 :

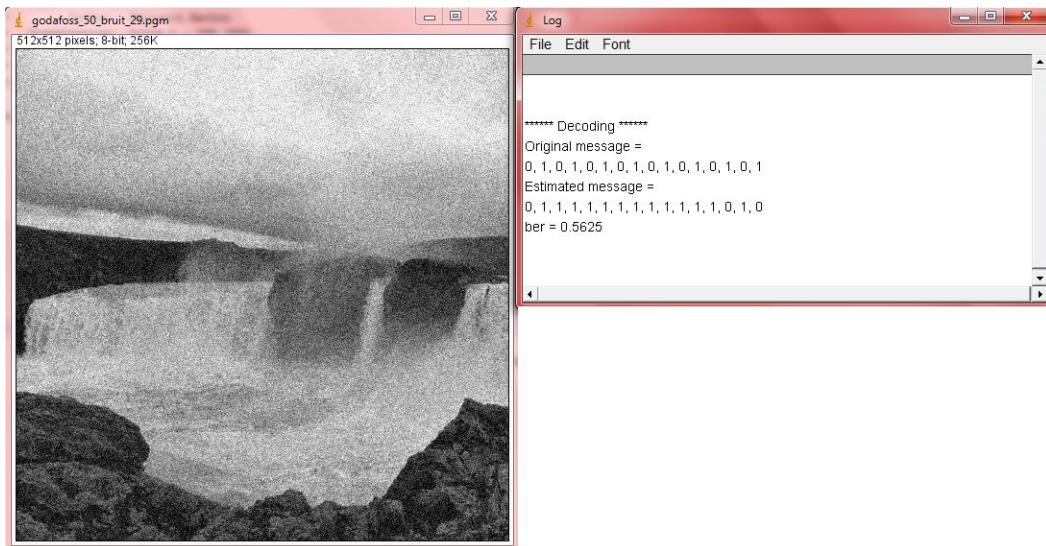
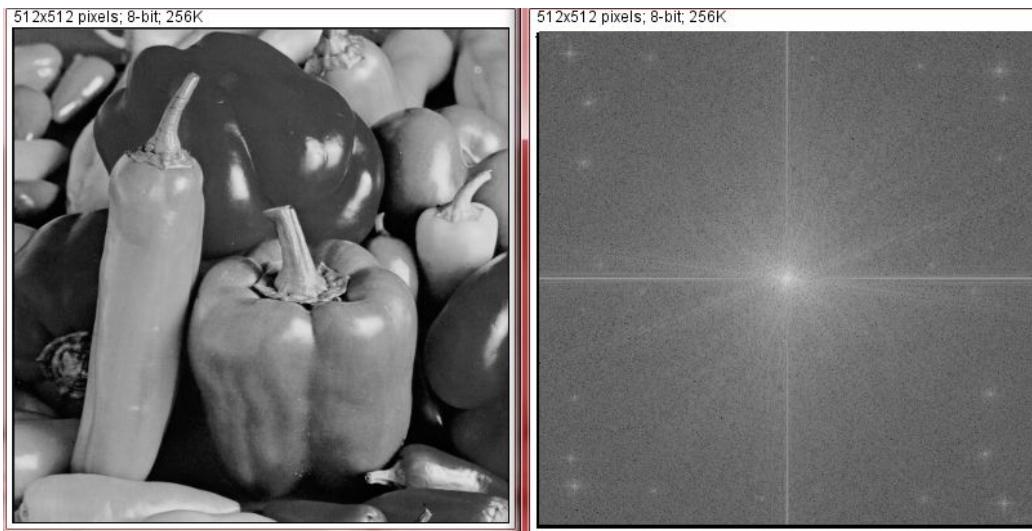


Image ‘godafoss.pgm’ tatouée avec un PSNR cible de 50 et affectée par un bruit gaussien d’écart-type 29, son décodage associé ainsi que son BER (= 0,5625).

Le BER est le même que l'image tatouée de base. Nous pouvons donc conclure que l'image ‘godafoss.pgm’ tatouée avec un PSNR de 50 est robuste au bruit gaussien lorsque celui-ci possède une valeur d'écart-type inférieure à 30. Cela est probablement dû au fait que cette image possède plus de basses fréquences puisque celles-ci ont la propriété d'être moins imperceptibles (d'où le PSNR élevé) mais sont généralement plus robustes (bruit gaussien d'écart-type élevé).

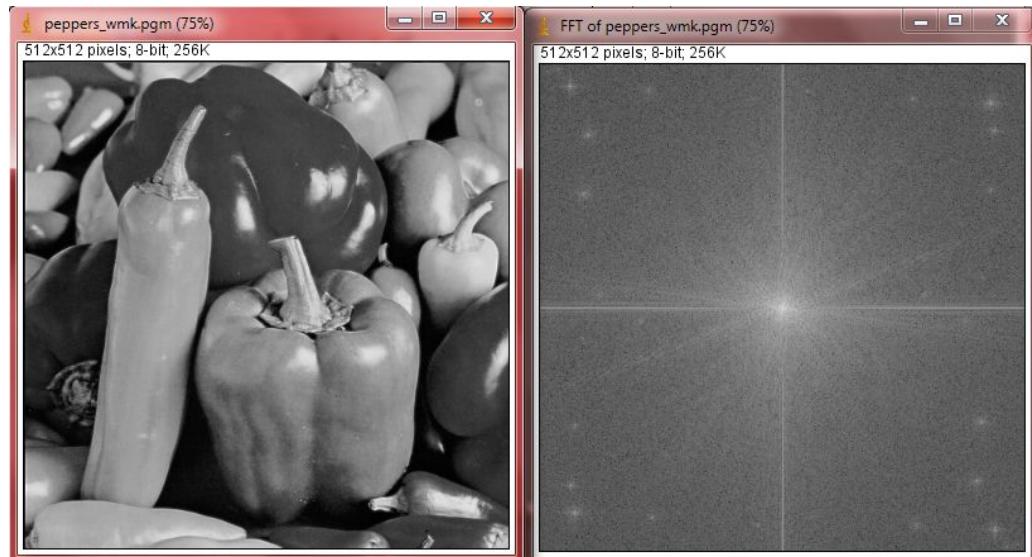
Tatouer une image puis analyser sa FFT (en la seuillant si nécessaire). Comparer avec la FFT de l'image hôte. Que constatez-vous ? Effectuer une rotation de l'image tatouée de 10° grâce au menu Image → Transform → Rotate... Enregistrer cette image puis procéder au décodage. Retrouve-t-on le message caché ? Analyser la FFT de cette image et proposer (sans la coder) une méthode permettant de rendre le schéma actuel de tatouage robuste face à la rotation.

Nous choisissons de travailler sur l'image ‘peppers.pgm’. Dans un premier temps, nous analysons sa FFT grâce au menu **Process** → **FFT** → **FFT**. Nous obtenons le résultat suivant.



*Image ‘peppers.pgm’ et sa FFT*

Nous effectuons la même manipulation, cette fois-ci avec l'image ‘*peppers.pgm*’ tatouée (l'image ‘*peppers\_wmk.pgm*’) et nous observons le résultat suivant :



*Image ‘peppers.pgm’ tatouée et sa FFT*

Nous constatons que leur FFT sont similaires à l'oeil ce qui nous confirme que le tatouage est imperceptible à l'oeil humain.

Nous effectuons ensuite une rotation de l'image ‘*peppers.pgm*’ tatouée de 10° grâce au menu **Image** → **Transform** → **Rotate....** Le résultat obtenu est le suivant :



*Image ‘peppers.pgm’ tatouée ayant subi une rotation de 10°*

Après avoir enregistré cette image sous le nom de ‘peppers\_wmk\_rot.pgm’, nous tentons de décoder le message. Voici ce que nous obtenons :



*Image ‘peppers\_wmk\_rot.pgm’ ainsi que le décodage de son message et son BER*

Nous remarquons que le message estimé ne correspond plus au message original. Le BER le prouve car il est beaucoup plus élevé que le BER de “base” (=celui obtenu avec le gradient de Sobel) de l'image ‘peppers.pgm’ tatouée (0,3125). Nous décidons maintenant d'analyse la FFT de l'image ‘peppers\_wmk\_rot.pgm’.

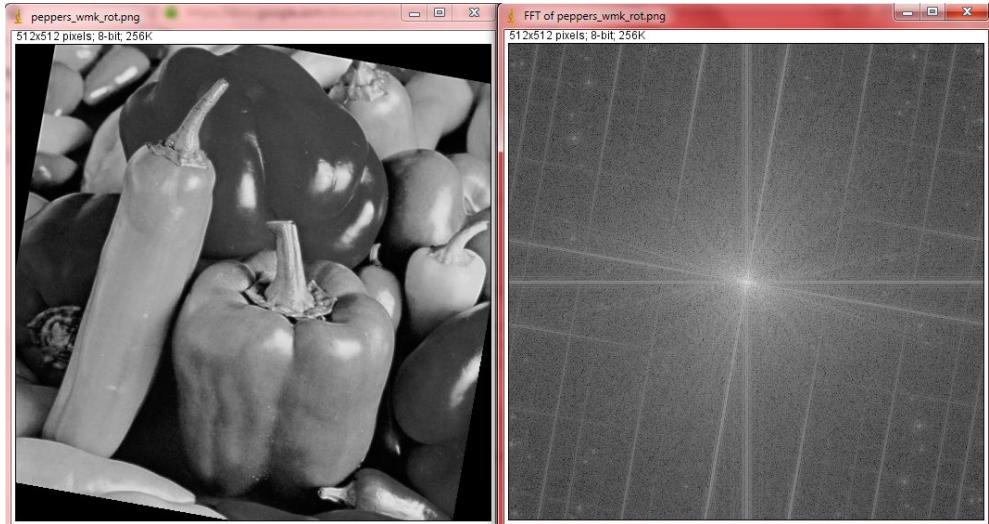


Image ‘peppers\_wmk\_rot.pgm’ et sa FFT

La FFT de l'image ‘*peppers\_wmk\_rot.pgm*’ est très différente de la FFT de l'image ‘*peppers.pgm*’. Cela nous permet donc de conclure que notre tatouage n'est pas robuste face à la rotation.

Il existe cependant une solution afin de rendre le tatouage robuste face à la rotation. Pour cela, il est alors nécessaire de tatouer l'image de manière cyclique suivant le centre de l'image.

## Annexes

### Annexe 1 : Fonction *makeGradientNorm*

```
function makeGradientNorm(imageID){  
    selectImage(imageID);  
    run("Duplicate...", "title=dx.pgm");  
    selectImage(imageID);  
    run("Duplicate...", "title=dy.pgm");  
    selectImage("dx.pgm");  
    run("32-bit");  
    selectImage("dy.pgm");  
    run("32-bit");  
    selectImage("dx.pgm");  
    run("Convolve...", "text1=[-0.125 0 0.125\\n-0.25 0 0.25\\n-0.125 0 0.125\\n] normalize");  
    selectImage("dy.pgm");  
    run("Convolve...", "text1=[-0.125 -0.25 -0.125\\n0 0 0 \\n0.125 0.25 0.125\\n] normalize");  
    selectImage("dx.pgm");  
    run("Square");  
    selectImage("dy.pgm");  
    run("Square");  
    imageCalculator("Add create 32-bit", "dx.pgm","dy.pgm");  
    selectImage("Result of dx.pgm");  
    run("Square Root");  
    rename("gradNorm.pgm");  
  
    return getImageID();  
}
```

*Code de la fonction makeGradientNorm*

## Annexe 2 : Modification de la fonction d'encodage

```
***** Embedding process *****/
if (type == choice[0]) {
    /* open host image */
    open();
    /* convert to 32-bit */
    run("Duplicate...", "title=host");
    run("32-bit");
    print("\n\n***** Embedding *****\n");
    print("Number of bits to hide = "+Nc);
    print("Targeted PSNR = "+tpsnr);
    IDhost = getImageID();
    widthImg = getWidth();
    heightImg = getHeight();
    /* make secret carriers into a Nc-stack */
    IDcarriers = makeCarriers(Nc,widthTile,heightTile,secretKey);
    /* compute watermark image */
    IDWmk = makeWmkSignal(IDcarriers,widthImg,heightImg,msg);
    /* psychovisual masking using the norm of the gradient */
    IDgradient = makeGradientNorm(IDhost);
    IDWmk = makeGradientNorm(IDhost);
    imageCalculator("Multiply create 32-bit", IDWmk, IDgradient);
    /* watermarking strength */
    selectImage(IDWmk);
    eqm = 0.0;
    for (j= 0; j< widthImg; j++){
        for (i= 0; i< heightImg; i++){
            value = getPixel(j,i);
            eqm += value*value;
        }
    }
    eqm = eqm / (heightImg*widthImg);
    gamma = sqrt( 255*255/( eqm*pow(10.,tpsnr/10.) ) );
    run("Multiply...", "value="+gamma);
    /* update display LUT */
    resetMinAndMax();
    run("Duplicate...", "title=WmkImage");
    IDWmkImage = getImageID();
    /* add host image to watermark signal to produce watermarked image */
    imageCalculator("Add 32-bit", IDWmkImage, IDhost);
    selectImage(IDWmkImage);
    run("8-bit");
    /* final psnr */
    psnr = psnrImg(IDWmkImage, IDhost);
    print("Final PSNR = "+psnr);
    selectImage(IDhost);
    close();
    selectImage(IDWmkImage);
} //end of embedding process
```

*Code de la fonction d'encodage modifiée*