Université de Nîmes
Licence 3 Mathématiques Informatique
Module Projet Informatique
Année 2008-2009
Enseignant : Marc Chaumont

Utilisation des codes correcteurs d'erreurs en stéganographie : De l'algorithme F5 et sa stéganalyse aux codes à papier mouillé

Rémi Watrigant

Table des matières

1 Introduction

Ce dossier est le compte-rendu du projet effectué durant le second semestre de ma troisième année de Licence Mathématiques-Informatique, à l'université de Nîmes, dans un module dirigé par M. Chaumont.

Ce projet a consisté à étudier les principes de la stéganographie adapatée aux images. Cette dernière est définie comme l'art consistant à insérer un message dans une image, en cachant l'existence même du message. Elle se différencie du tatouage par le rôle de l'attaquant au sein des schémas. La stéganographie existe depuis de nombreuses années, avec des techniques comme l'encre invisible (apparu au 1er siècle avant J-C), ou par la technique du micropoint de Zapp, utilisé pendant la deuxième guerre mondiale par les allemands, consistant à réduire la taille d'une photo en un point pouvant être inséré dans du texte, à la place des points des i par exemple. De nos jours, et particulièrement durant les deux dernières décennies, avec l'essor d'Internet, la stéganographie s'est adapatée aux formats numériques, et notamment dans les images, qui sont énormément partagées sur la toile. Elle est utilisée par toutes les personnes cherchant à communiquer des messages discrètement, comme par exemple l'armée, ou les organisations terroristes. Cependant, étudier les techniques de stéganographie amène également à étudier les techniques de stéganalyse : l'art d'attaquer les schémas de stéganographie, et donc de déterminer la présence ou non de messages cachés au sein des supports. La stéganographie est donc un domaine de recherche de plus en plus étudié de nos jours.

Je me suis intéressé à ce projet car il traite de nombreux domaines qui m'intéressent : la théorie de l'information, avec la communication au sein de canaux non sûrs, et le traitement d'images, notamment en domaine fréquentiel.

Je remercie M. Chaumont pour l'aide qu'il m'a apporté dans la compréhension des concepts, et pour avoir mis à ma disposition de la littérature et des codes sources sur le sujet.

La partie 2 traite du concept de stéganographie en général, dans le contexte des formats numériques. La partie 3 décrit une technique de stéganographie utilisant la théorie des codes correcteurs, avec notamment deux applications : les codes de Hamming, et les codes à papier mouillé. Enfin, la partie 4 traite d'une implémentation d'un schéma de stéganographie et de son attaque : l'algorithme F5. Elle contient également la présentation de mes implémentations et quelques résultats expérimentaux. Le dossier est conclu partie 5, et les codes sources sont disponibles en annexe.

2 La stéganographie en général

2.1 Définition d'un schéma de stéganographie

De manière intuitive, la stéganographie peut se résumer à l'histoire suivante :

Alice et Bob sont en prison, enfermés dans deux cellules séparées l'une de l'autre, et souhaiteraient plannifier une évasion. Ils sont autorisés à communiquer par le biais de messages surveillés, afin qu'ils ne discutent pas d'un plan pour s'échapper. La personne qui surveille les messages est Eve, la gardienne. Si Eve détecte le moindre signe de conspiration, elle transfèrera Alice et Bob dans des prisons de haute sécurité, d'où personne n'a jamais pu s'évader... Alice et Bob sont conscients de ça, et ont par conséquent partagé un secret commun avant d'être enfermés, qu'ils vont utiliser afin de cacher leur plan dans des messages innocents. Les projets d'Alice et Bob réussiront s'ils arrivent à s'échanger des informations afin de s'échapper de la prison, sans que Eve soit suspicieuse.

Après avoir décrit la stéganographie de manière intuitive, nous allons maintenant nous intéresser à son formalisme mathématique et à une description plus précise.

Un schéma stéganographique est la donnée de deux fonctions et d'une ou plusieurs clés. De la même manière qu'en cryptographie, on peut discerner deux types de stéganographie : à clé privée et à clé publique. Dans le cas d'un algorithme de stéganographie à clé privée, l'émetteur, Alice, et le récepteur, Bob, utilisent un canal sécurisé (ou bien un système de partage de secret robuste) afin de définir une clé privée commune, qui sera utilisée pour insérer un message et pour l'extraire. Dans le cas d'un algorithme à clé publique, l'emetteur dispose d'une clé publique pour insérer le message, et le récepteur d'une clé privée, lui seul pouvant ainsi extraire le message (Dans la pratique, la clé privée est le plus souvent l'image de la clé publique par une fonction dite "à sens unique").

Nous ne nous intéresserons ici uniquement à des algorithmes à clé privée.

Dans la suite de ce dossier, nous utiliserons les notations suivantes :

K représente l'ensemble des clés possibles, M représente l'ensemble des messages possibles à insérer, et C l'ensemble de tous les supports possibles.

Un schéma stéganographique est définie par deux fonctions : $Emb : C \times M \times K \to C$ qui est la fonction d'insertion (embedding), qui prend en paramètre un support, un message et une clé privée, et retourne un élément de C, et $Ext : C \times K \to M$ qui est la fonction d'extraction.

Le but d'un schéma stéganographique est de retrouver un message inséré dans un support. Ceci se traduit mathématiquement par l'égalité : $Ext(Emb(c, m, k), k) = m,\ \forall (c, m, k) \in C \times M \times K$. Le résultat de la fonction *emb* est habituellement appelé stégo-medium, ou stego-work, et le support c est appelé médium, ou cover-work. Le fonctionnement d'un schéma de stéganographie est résumé par la figure 1.

Les algorithmes étudiés dans ce projet sont destinés à insérer des messages de type texte dans des images. Ainsi, on assimile l'ensemble des messages M à l'ensemble $\mathbb{F}_2^l = \{0,1\}^l$, l étant la taille du message en bits, et $\mathbb{F}_2$ étant le corps de Galois à deux éléments, en prenant par exemple pour chaque caractère la représentation en base 2 de son code ASCII (nombre entre 0 et 255). De même, on assimile l'ensemble des supports C à $\mathbb{F}_2^n = \{0,1\}^n$, n étant la taille du support en bits, à l'aide d'une certaine fonction, par exemple en prenant la valeur du LSB (Least Significant Bit : bit de poids faible, c'est à dire le bit se trouvant le plus à droite dans la représentation d'un nombre entier en binaire) de la composante lumineuse de chaque pixel de l'image.

Certains algorithmes utilisent en plus une technique de cryptographie, en insérant alors non pas le message en lui même, mais le cryptogramme associé à celui-ci par un algorithme de cryptographie. Le but de cette technique et d'ajouter une sécurité supplémentaire au schéma, en empêchant de retrouver le message dans le cas d'une detection de l'insertion par un attaquant. Cependant, la stéganographie consistant à cacher l'existence même du message, l'utilisation de cryptographie est facultative et n'a pas besoin d'exister si le schéma stéganographique est sûr. Ainsi, nous ne nous

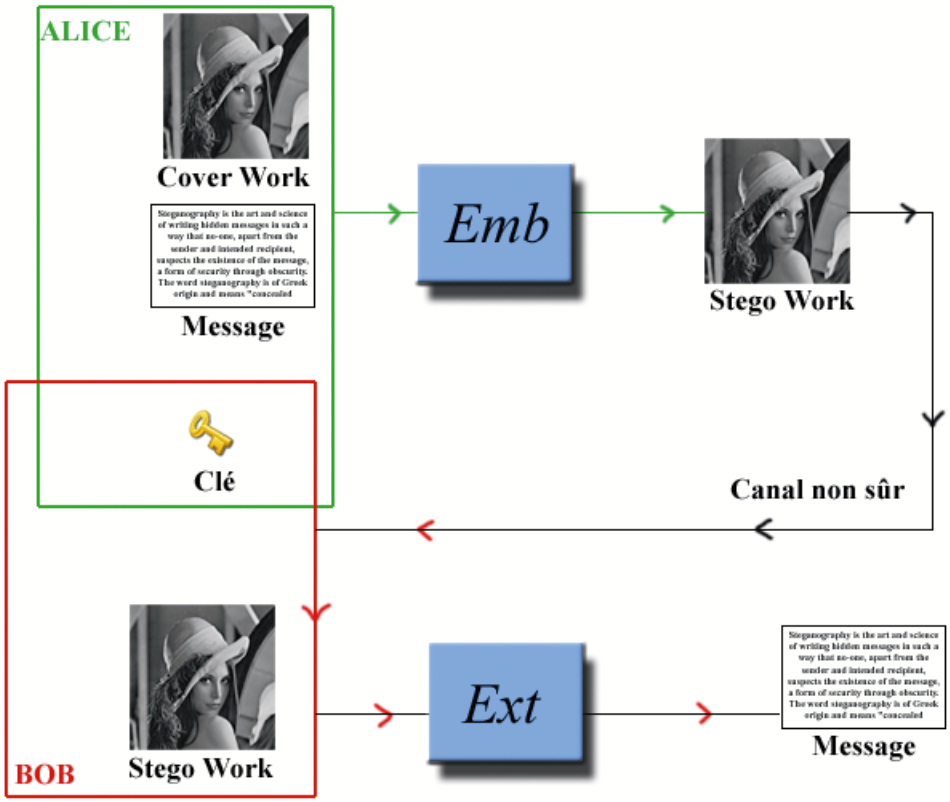

FIG. 1 – Schéma des fonctions d'insertion et d'extraction d'un message dans une image

intéresserons pas à l'ajout d'une technique de cryptographie dans les algorithmes étudiés.

2.2 Attaque d'un schéma de stéganographie

Cette section aborde la stéganalyse : les systèmes visants à attaquer les systèmes de stéganographie.

Tout d'abord, si la stéganographie vise à cacher l'existence d'un message dans un support, alors le stéganalyste cherche en premier à déterminer, pour un support donné, si celui-ci est porteur d'un message ou non. L'attaquant donne donc une réponse binaire (ou une probabilité) selon si un message a été inséré ou non.

Ensuite, dans le cas d'une réponse positive au test précédent, le stéganalyste peut alors chercher à extraire le plus d'information concernant le message inséré, et dans le meilleur des cas, extraire le message en lui même.

Enfin, s'il y a eu cryptographie lors de l'insertion, alors l'attaquant doit cryptanalyser le message extrait.

Nous nous plaçons ici dans le cas d'un attaquant passif. En reprenant l'exemple de l'introduction, Eve est donc le gardien, et contrôle les échanges entre les deux prisonniers Alice et Bob, sans les modifier (d'où le terme passif). Eve sait évidemment ce qui constitue une communication légitime ou non entre les deux prisonniers, et connait également les spécifications des techniques de stéganographie utilisées. Ce qu'il ignore en revanche, c'est la clé secrète partagée par Alice et Bob.

Ainsi, deux approches peuvent être distinguées : une attaque par recherche exhaustive sur la clé, et une attaque par analyse de la légitimité de l'échange. La première étant irréalisable en pratique, nous allons nous intéresser à la seconde.

L'attaque par analyse de la légitimité de l'échange consiste dans notre cas à effectuer une mesure sur l'image interceptée, et comparer cette mesure avec celle d'une image non stéganographiée. Là

encore, deux possibilités s'offrent au stéganalyste : une attaque ciblée ou non ciblée.

Une attaque ciblée consiste à étudier en profondeur une technique précise, et à mettre en valeur une propriété caractéristique des l'images après l'insertion d'un message. Nous étudierons un exemple d'attaque ciblée dans la section 4.4 page 22, dans le cas de l'algorithme F5.

Une attaque non ciblée, elle, consiste à construire un système de discrimination : tout d'abord, le système doit apprendre et classer des quantités d'images dont on sait si elles ont été stéganographiée ou non (par n'importe quel algorithme de stéganographie, d'où le terme "non ciblée"). Ensuite, le système peut dresser une "frontière" entre les images porteuses d'un message et les images normales, et ainsi déterminer dans quelle classe se trouve une image interceptée. Ce genre de système porte le nom de Machines à Support Vectoriel (SVM).

La prise en compte de ces attaques, notamment les attaques ciblées, peuvent servir à établir un critère de sécurité dans les schémas stéganographiques. Ce critère de sécurité, ainsi que d'autres plus généraux, sont décrits dans la partie suivante.

2.3 Sécurité d'un schéma de stéganographie

Le but d'un schéma stéganographique est d'insérer un message dans un support, en cachant l'existence même de ce message. Ainsi, on peut discerner deux critères d'évaluation :

- la capacité.
- l'indétectabilité de ce message.

Il est évident de remarquer à première vue que ces deux critères sont en opposition : plus le message est gros, plus il sera difficile à cacher. L'emetteur et le recepteur vont donc devoir trouver un compromis entre la capacité et l'indetectabilité afin de construire un bon schéma stéganographique.

Concernant la capacité, plusieurs mesures sont utilisées. Tout d'abord la capacité d'insertion, définie par $log_2|\mathcal{M}|$, $|\mathcal{M}|$ étant le nombre de message possibles. On peut également définir la capacité d'insertion relative, qui est la capacité divisé par la taille du support. Enfin, le critère le plus utilisé est "l'efficacité d'insertion" (embedding efficiency), qui est par définition le nombre de bits insérés pour une modification dans le support. Evidemment, plus ce nombre est grand, plus nous avons la possibilité d'insérer un grand nombre de bits dans le message, car il y aura alors peu de modifications, donc peu de détection. Cependant, le nombre de modifications apportées au support n'est pas directement le critère le plus adapté pour caractériser la sécurité d'un schéma. En effet, nous allons voir que la manière de le modifier est plus importante pour préserver l'indétectabilité des données cachées.

La notion d'indétectabilité se réfère à la résistance à une attaque ciblée. Ce critère a été adapté des concepts de la théorie de l'information, et ennoncé pour la première fois par Cachin dans [4].

Afin de déterminer la sécurité d'un système, nous allons comparer la distribution d'un support, avant et après insertion.

En effet, soit c un support de couverture de distribution de probabilité P_c, assimilé à une variable aléatoire sur l'alphabet Σ, et s, de distribution de probabilité P_s, l'image de c par la fonction d'insertion d'un schéma stéganographique. La sécurité de ce schéma est alors dépendante des similitudes entre P_c et P_s. Plus précisément, Cachin utilise l'entropie relative des deux distributions de probabilité pour définir ce critère. Cette distance (qui n'est pas une distance au sens mathématique du terme, car non symétrique) est définie par :

$$D(P_c||P_s) = \sum_{i\in\Sigma} P_c(i) log_2 \frac{P_c(i)}{P_s(i)}$$

Le schéma est alors considéré comme sûr si $D(P_c||P_s) = 0$, et considéré comme ε-sûr si $D(P_c||P_s) < \varepsilon$.

Afin de mieux comprendre le terme de ε-sûr, plaçons nous du côté de l'attaquant. Considérons que celui-ci peut commettre deux erreurs lors de l'interception d'un message[1] entre Alice et Bob :

- Erreur 1 : le message est considéré comme illégal alors qu'il ne contient pas de données cachées. Supposons la probabilité de cette erreur égale à α.
- Erreur 2 : le message est considéré comme légal alors qu'il contient des données cachées. Supposons la probabilité de cette erreur égale à β.

Face à un message intercepté, l'attaquant doit répondre 0 (message légal) ou 1 (message illégal). Alors :

- face à un message légal, l'attaquant va répondre 1 avec une probabilité égale à $alpha$, et 0 avec une probabilité égale à $1 - \alpha$.
- face à un message illégal, l'attaquant va répondre 1 avec une probabilité égale à $1 - \beta$, et 0 avec une probabilité égale à β.

Alors l'entropie relative entre les deux distributions du detecteur de l'attaquant est égale à :

$$D(\alpha||\beta) = (1-\alpha)log_2\frac{1-\alpha}{\beta} + \alpha log_2\frac{\alpha}{1-\beta} \tag{1}$$

α et β résultants de traitements sur P_c et P_s, leur entropie relative ne peut pas être supérieure à $D(P_c||P_s)$. Ainsi on a :

$$D(\alpha||\beta) \leq D(P_c||P_s) \tag{2}$$

Si l'on fixe $\alpha = 0$, c'est à dire que l'on interdit à l'attaquant de considérer des messages illégaux alors qu'ils sont légaux, l'équation 1 devient : $D(\alpha||\beta) = log_2\frac{1}{\beta}$.
Ainsi, l'équation 2 devient :

$$log_2\frac{1}{\beta} \leq D(P_c||P_s)$$

Or, comme $D(P_c||P_s) < \varepsilon$, on a :

$$log_2\frac{1}{\beta} < \varepsilon$$

ce qui équivaut à

$$\beta > 2^{-\varepsilon}$$

Ainsi, plus ε est petit, plus la probabilité qu'un message illégal ne soit pas détecté est grande. Le terme ε-sûr prend donc ici tout son sens dans l'expression du critère de sécurité d'un schéma stéganographique.

[1] Ici, le mot "message" correspond aux données circulant sur le canal non sécurisé, et non pas aux données insérées dans le support, comme défini dans la section précédente. Nous parlerons d'un message "légal" pour un message ne contenant pas de données cachées, "illégal" pour le contraire.

3 Quelques techniques de stéganographie

3.1 Technique LSB

3.1.1 Description

La première technique de stéganographie étudiée est un exemple très simpliste de ce que peut être un schéma de stéganographie. C'est aussi un très bon exemple de système non sûr, où nous pourrons appliquer les principes vus dans la section précédente.

Cet algorithme nécessite d'abord de convertir l'image en une suite de nombre binaires. Pour cela, nous pouvons par exemple prendre la représentation en base 2 de la valeur de chaque pixel (un nombre si l'image est en niveaux de gris, 3 nombres si elle est en couleurs). Elle nécessite également de convertir le message à insérer en une suite de 0 et de 1. Pour cela, nous pouvons par exemple prendre la concaténation de la représentation en base 2 de chaque caractère ASCII.

Afin de garantir un minimum de sécurité (même si cet algorithme est très facilement attaquable, comme nous le verrons plus bas), la plupart des algorithmes de stéganographie mélangent d'abord les éléments du support, afin de modifier celui-ci de manière uniforme. Les figures 2 et 3, tirées de [6], illustrent ce principe : la figure 2 représente des modifications sur le support (en noir) sans avoir mélangé celui-ci auparavant : on remarque une concentration des modifications sur une zone du support. La figure 3 représente d'abord un mélange du support, puis des modifications (en noir), puis l'inverse du mélange effectué : on remarque une distribution uniforme des modifications sur tout le support.

La marche aléatoire est initialisée par la clé privée partagée par l'emetteur et le recepteur, de façon à ce que le récepteur puisse extraire le message.

Fig. 2 – Concentration des modifications (en noir) lors d'une insertion sans mélange du support

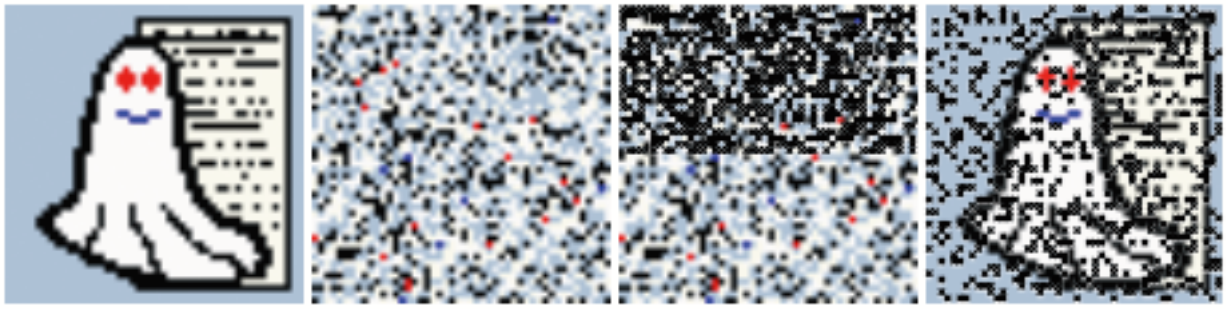

Fig. 3 – Modifications uniformément réparties lors d'une insertion après mélange du support

Le principe de cet algorithme est de remplacer le bit de poids faible de chaque pixel (LSB : Least Significant Bit, bit le plus à droite dans la représentation en base 2 d'un nombre) par un bit

du symbole. La figure 4 illustre ce principe.

Lors de l'extraction, le récepteur n'aura alors plus qu'à initialiser la marche aléatoire sur le support à l'aide de la clé privée, et de récupérer le LSB de chaque pixel pour reconstruire le message.

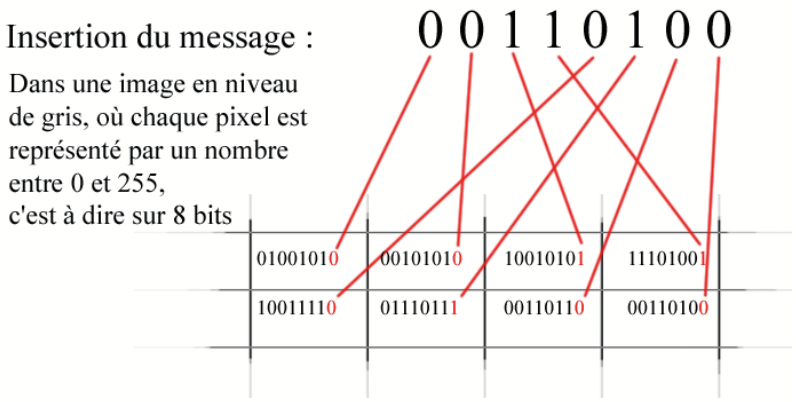

FIG. 4 – Stéganographie LSB sur une image en niveau de gris

3.1.2 Inconvénients, attaque

Le changement du LSB de chaque pixel entrainant une modification de +1 ou -1 sur celui-ci, l'oeil humain ne voit alors pas la différence entre une image normale et une image stéganographiée, le système semble sûr face aux attaques. Cependant, les modifications apportées entrainent un tel changement dans la distribution du support qu'il est très facilement attaquable.

Pourtant, à première vue, la distribution des LSB de l'image et des éléments du message étant uniformes, les changements ne semblent alors pas modifier grand chose. Cependant, si l'on observe maintenant l'histogramme de l'image, on peut alors y remarquer une caractéristique typique de ce type d'insertion :

En effet, un changement de ± 1 sur chaque pixel va transformer un pixel de valeur $2i$ en un pixel de valeur $2i+1$. Inversement, un pixel de valeur $2i+1$ deviendra un pixel de valeur $2i$. Ainsi, dans l'histogramme, les "barres" de chaque paire $(2i, 2i+1)$ auront tendance à s'égaliser.

En effet, si par exemple l'histogramme présente 10 pixels d'intensité 30, et 400 pixels d'intensité 31, alors, comme chaque pixel a une chance sur deux d'être modifié, environ 5 pixels d'intensité 30 deviendront d'intensité 31, et environ 200 pixels d'intensité 31 deviendront d'intensité 30. Nous obtiendrons alors dans l'histogramme de l'image stéganographiée environ le même nombre de pixels pour la valeur 30 et 31 : environ 205. La figure 5 illustre ce phénomène.

Nous pouvons donc voir que même si l'image stéganographiée ne parait pas modifiée, le changement dans la distribution des valeurs de pixels est si fort qu'une simple analyse statistique des paires de valeurs permettent d'attaquer le système. Ce genre d'attaque est le principe du test dit du χ^2.

3.2 Matrix embedding

3.2.1 Point de vue intuitif

Dans l'exemple ci-dessus, la capacité d'insertion est de n : la taille du support. Etant donné qu'un bit a une chance sur deux d'être modifié, l'efficacité d'insertion est de 2 : nous pouvons insé-

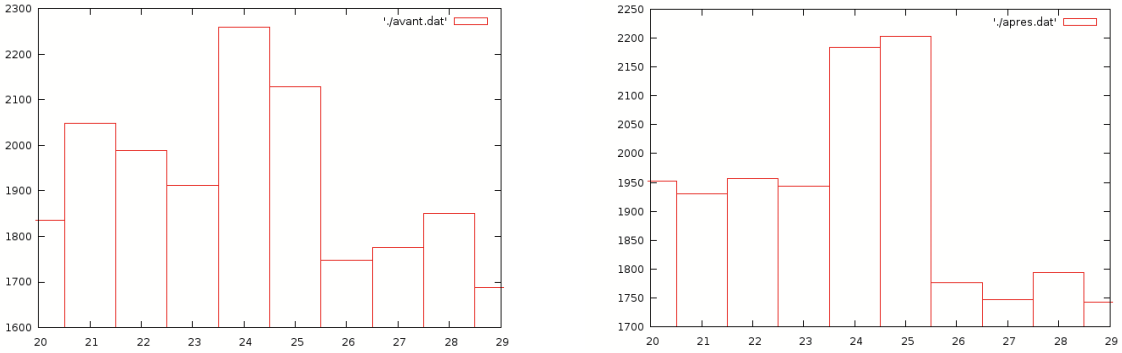

FIG. 5 – Partie de l'histogramme de l'image de Lena avant insertion LSB (à gauche) et apres insertion LSB (à droite) : les paires de valeurs s'égalisent

rer deux bits en ne générant qu'une modification en moyenne. Cette efficacité peut cependant être augmentée, si le message à insérer est plus petit. En effet, observons l'exemple suivant :

Soit deux bits b_1, b_2 à insérer à l'aide de trois bits du support : x_1, x_2 et x_3. Soit $a_1 = x_1 \oplus x_2$ et $a_2 = x_2 \oplus x_3$. Afin d'insérer nos deux bits, discernons les quatre cas suivants :
- si $a_1 = b_1$ et $a_2 = b_2$: pas de changements.
- si $a_1 \neq b_1$ et $a_2 = b_2$: changeons la valeur de x_1.
- si $a_1 = b_1$ et $a_2 \neq b_2$: changeons la valeur de x_3.
- si $a_1 \neq b_1$ et $a_2 \neq b_2$: changeons la valeur de x_2.

Lors de l'extraction, le recepteur n'aura alors qu'à calculer a_1 et a_2 qui seront les deux bits insérés.

Nous pouvons alors remarquer que pour insérer deux bits, la probabilité d'effectuer une modification sur le support est de $\frac{3}{4}$, alors qu'il était de 1 pour la technique précédente. Le prix à payer est que la capacité d'insertion est de $\frac{2n}{3}$, c'est à dire que la taille relative[2] du message par rapport au support doit être de $\frac{2}{3}$, alors qu'elle était de 1 précédemment.

La partie suivante explique qu'en fait, cette technique n'est qu'un cas particulier d'une autre, appelée matrix embedding, et que l'efficacité d'insertion peut encore être augmentée, si la taille relative diminue.

3.2.2 Généralisation

La technique de matrix embedding est une méthode de codage par syndrome, utilisant la théorie des codes correcteurs, en particulier les codes linéaires.

Codes correcteurs d'erreurs, codes linéaires

Un code linéaire est un code correcteur d'erreurs structuré comme sous-espace vectoriel d'un corps fini. Ici, le corps utilisé est le corps de Galois à deux éléments : $\mathbb{F}_2 = \{0, 1\}$.

Un code linéaire est défini par trois paramètres : $[n, k, \delta]$, avec $n > k$, où des message de k bits sont transmis comme des mots de n bits. δ représente la distance minimale entre chaque mot du code. La distance utilisée est la distance de Hamming, définie par : $d(x, y) = \sum_{i=1}^{n} x_i \oplus y_i$ pour tout $(x, y) \in \mathbb{F}_2^n \times \mathbb{F}_2^n$. Nous définissons de plus le poids de Hamming d'un vecteur x par $d(x, 0)$, c'est à dire le nombre de 1 dans le vecteur.

[2]La taille relative est définie comme la longueur du message à insérer divisé par la taille du support disponible

Dans la théorie des codes correcteurs, deux entités souhaitent communiquer à travers un canal bruité. Pour cela, l'emmeteur, qui désire envoyer un message $m \in \mathbb{F}_2^k$, transforme celui-ci en un mot de code $c \in \mathbb{F}_2^n$ par :

$$c = Gm$$

avec $G \in \mathbb{M}_{n,k}$ la matrice dite *génératrice* du code linéaire. Le récepteur, qui reçoit un mot $c' \in \mathbb{F}_2^n$, calcule alors ce que l'on appelle le *syndrome* du code, $s \in \mathbb{F}_2^{n-k}$, par :

$$s = Hc'$$

avec $H \in \mathbb{M}_{n-k,n}$ la matrice dite *de parité* du code. Remarquons que $s \in \mathbb{F}_2^{n-k}$

Si $s = (0, ..., 0)$, alors $c' = c$ et le message ne comporte pas d'erreur. Autrement, si $s \neq (0, ..., 0)$, alors le message comporte une ou plusieurs erreurs.

De plus, H étant une application de $\mathbb{F}_2^n$ dans $\mathbb{F}_2^{n-k}$, plusieurs codes peuvent avoir le même syndrome s. On note alors $C(s)$ l'ensemble des codes ayant s comme syndrome. Cette ensemble s'appelle la classe (coset) de s, et est défini par : $C(s) = \{c \in \mathbb{F}_2^n : Hc = s\}$. On appelle alors chef de classe (coset leader) l'élément de la classe qui a le poids de Hamming le plus faible.

Utilisation des codes linéaires en stéganographie : matrix embedding

Dans notre utilisation des codes linéaires, le syndrome s représentera le message à transmettre, et les données traversant le canal, c', seront le support. Lors de l'extraction du message, le récepteur n'aura alors qu'à calculer le syndrome à l'aide de la matrice H pour obtenir le message caché. La difficulté se trouve ici du côté de l'emetteur.

En effet, le but est d'insérer un message $m \in \mathbb{F}_2^{n-k}$ dans un support $x \in \mathbb{F}_2^n$ en le modifiant le moins possible. Pour cela, le principe de la technique est de modifier x en y tel que

$$Hy = m \tag{3}$$

Avec $H \in \mathbb{M}_{n-k,n}$. Nous cherchons alors le vecteur $e \in \mathbb{F}_2^n$ qui modifie x en y, c'est à dire tel que

$$y = x + e \tag{4}$$

Injectant l'équation 4 dans 3, on obtient :

$$H(x + e) = m$$

$$\Leftrightarrow He = m - Hx \tag{5}$$

Le vecteur e recherché est donc un code ayant comme syndrome $m - Hx$. Etant donné que nous cherchons à modifier le moins possible notre vecteur x, nous allons choisir le code ayant le poids de Hamming le plus faible : le vecteur optimal est le chef de la classe $C(m - Hx)$. On montre en fait que le nombre maximum de changement est inférieur au rayon de couverture R du code, car le vecteur e aura un poids de Hamming inférieur à R.

En général, la résolution de l'équation 5 est compliquée, et nécessite l'utilisation du pivot de Gauss, de complexité cubique sur le nombre de lignes de H (ici n-k). Cependant, l'utilisation de bons codes linéaires peut simplifier le problème, notamment avec des codes dont le rayon de couverture est faible.

Exemple de matrix embedding : utilisation des codes de Hamming

Les codes de Hamming constituent une famille de codes linéaires permettant la détection et la correction d'une erreur si elle ne porte que sur un symbole du message. Leur distance minimale δ est égale à 3, et le rayon de couverture à 1.

Ces codes sont paramêtrés par un entier $p : n = 2^p - 1$ et $k = 2^p - 1 - p$. Les syndromes (donc les messages dans le cas de la stéganographie) sont donc de longueur $n - k = p$. La matrice de parité $H \in \mathbb{M}_{p,2^p-1}$ est formée en plaçant sur chaque colonne la représentation en binaire des entiers de 1 à $2^p - 1$.

Donc, si $m - Hx = He$ est un syndrome, alors (comme $m - Hx$ est de longueur p), il figure dans une des colonnes de H. Ainsi, e sera le vecteur formé d'un 1 à cette colonne, et de 0 ailleurs : un seul changement est nécessaire pour transformer notre x en y ! Voici un exemple pour $p = 3$:

Soit $m = (1, 0, 1)$ le message à insérer dans $x = (0, 1, 1, 1, 0, 0, 1)$.
La matrice H utilisée sera :

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Nous cherchons donc le vecteur $e = (e_1, e_2, e_3, e_4, e_5, e_6, e_7,)$ tel que $H(x + e) = m$.
Pour cela, calculons $m - Hx$:

$$\begin{aligned} m - Hx &= \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \end{aligned}$$

On a donc He = (1, 1, 1) qui correspond à la 7ème ligne de H, d'où

$$e = (0, 0, 0, 0, 0, 0, 1)$$

Ainsi, le message transmis sera $y = x + e = (0, 1, 1, 1, 0, 0, 0)$. Lors de l'extraction, le recepteur n'a alors plus qu'a calculer le syndrome :

$$\begin{aligned} Hy &= \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \\ &= m \end{aligned}$$

Nous venons donc de voir que la technique de matrix embedding permet d'insérer p bits dans un support de $2^p - 1$ bits en effectuant au maximum[3] un seul changement dans celui-ci. Si cela parait à première vue très efficace, remarquons qu'une image de 512x512 pixels comporte 262144 pixels, donc 262144 bits potentiels pour servir de support. Si l'on veut appliquer la technique précédente, et n'effectuer qu'un seul changement dans l'image, le message doit être de longueur 17, car

[3] En effet, la technique engendre soit 0 soit 1 changement

$2^{18} - 1 = 262143 > 262144$. Cela représente une longueur relative de $\frac{17}{262144} = 6,48.10^-5$, ce qui est peu.

Une technique possible pour palier à cette faiblesse est de diviser le message en plusieurs parties, et de dérouler la méthode avec chaque "morceau". Par exemple, pour p=3, la technique nous permet d'insérer 3 bits dans un support de 7 bits, en n'effectuant au maximum qu'un seul changement. Ainsi, si l'on veut insérer 9 bits, il est nécessaire d'avoir un support de $2^9 = 512$ bits. Cependant, si l'on coupe le message en 3, alors on peut effectuer 3 fois l'insertion par matrice avec $p = 3$. Le support doit alors être de longueur $3 * 7 = 21$ bits, et celui-ci subira 3 changements. La longueur relative est maintenant de $\frac{9}{21} \simeq 0.4$. Une technique LSB simple comme décrite partie 3.1 aurait engendrée en moyenne 4.5 changements.

C'est de cette manière que sont implémentés la plupart des algorithmes de stéganographie utilisant la technique de matrix embedding, comme nous le verrons à la partie 4 pour l'algorithme F5.

3.2.3 Efficacité d'insertion

Nous allons maintenant nous intéresser à l'efficacité d'insertion de la technique de matrix embedding. L'efficacité d'insertion est définie comme le nombre de bits pouvant être insérés pour une modification dans le support.

Lorsque nous insérons p bits, dans un support de $2^p - 1$ bits, nous effectuons 0 ou 1 changement dans celui-ci. En effet, si $m = Hx$ alors $e = (0, ..., 0)$; le vecteur e représentant un nombre binaire dans l'intervalle $[0, 2^p - 1]$, ce cas a une probabilité de $\frac{1}{2^p}$. Inversement, nous changeons 1 bits dans le support dans toutes les autres situations, donc avec une probabilité de $1 - \frac{1}{2^p}$. Ainsi, le nombre moyen de changements pour insérer p bits est de :

$$0 * \frac{1}{2^p} \ + \ 1 * (1 - \frac{1}{2^p}) = 1 - \frac{1}{2^p} = 1 - 2^{-p}$$

D'où le nombre de bits pouvant être insérés pour 1 seul changement, i.e. l'efficacité d'insertion :

$$\frac{p}{1 - 2^{-p}}$$

D'un autre côté, puisque nous insérons p bits dans $2^p - 1$ bits, la taille relative du message est de :

$$\frac{p}{2^p - 1}$$

La figure 6 (page 14) représente l'évolution de l'efficacité d'insertion, tandis que la figure 7 (page 15) représente l'évolution de la taille relative du message, toutes les deux en fonction du paramètre p.

Choix du paramètre p

Nous pouvons donc voir que l'efficacité d'insertion ne dépend uniquement que du paramètre p utilisé. Dans la plupart des cas, comme expliqué à la fin du paragraphe précédent, ce paramètre n'est pas directement égal à la taille du message à insérer. Ainsi, comment choisir ce paramètre de manière optimale ? En fait, nous allons voir que celui-ci ne dépend uniquement que de la taille relative du message à insérer. En effet, nous devons déterminer le nombre de fois que nous allons devoir diviser notre message afin de l'insérer. Soit K la longueur de notre message, N la longueur de notre support, $\alpha = \frac{K}{N}$ la taille relative du message et a le nombre de fois que l'on va devoir diviser ce dernier. Chaque changement engendrant une modification, nous devons choisir a le plus petit possible. Celui-ci doit en fait vérifier :

$$a(2^{\frac{K}{a}} - 1) \leq N < (a + 1)(2^{\frac{K}{a+1}} - 1) \tag{6}$$

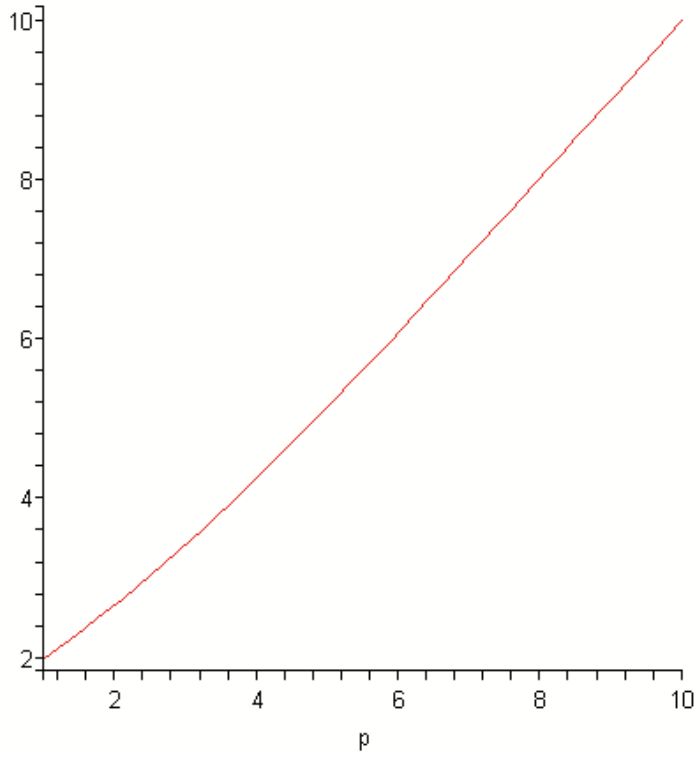

FIG. 6 – Evolution de l'efficacité d'insertion en fonction du paramètre p dans une technique de matrix embedding utilisant les codes de Hamming.

En effet, $L_a = (2^{\frac{K}{a}} - 1)$ est la longueur du support nécessaire pour un "morceau" de message de longueur $\frac{K}{a}$, donc aL_a est la longueur totale du support nécessaire. En divisant chaque membre de l'inégalité 6 par a ($a \neq 0$) on obtient :

$$\begin{aligned} 2^{\frac{K}{a}} - 1 \leq \frac{N}{a} < \frac{(a+1)(2^{\frac{K}{a+1}} - 1)}{a} &\Leftrightarrow \frac{a}{(a+1)(2^{\frac{K}{a+1}} - 1)} < \frac{a}{N} \leq \frac{1}{2^{\frac{K}{a}} - 1} \\ &\Leftrightarrow \frac{K}{a}\frac{a}{(a+1)(2^{\frac{K}{a+1}} - 1)} < \frac{K}{a}\frac{a}{N} \leq \frac{K}{a}\frac{1}{2^{\frac{K}{a}} - 1} \\ &\Leftrightarrow \underbrace{\frac{\frac{K}{(a+1)}}{(2^{\frac{K}{a+1}} - 1)}}_{\alpha_{a+1}} < \alpha \leq \underbrace{\frac{\frac{K}{a}}{2^{\frac{K}{a}} - 1}}_{\alpha_a} \end{aligned}$$

α_a et α_{a+1} sont les tailles relatives pour une opération de matrix embedding avec respectivement $p = \frac{K}{a}$ et $p = \frac{K}{a+1}$.

Ainsi, le choix optimal pour le paramètre p se calcule à partir de la taille relative du message. Donc, l'efficacité d'insertion ne dépend elle aussi que de la taille relative de celui-ci.

3.3 Codes à papier mouillé

3.3.1 Introduction

Cette partie traite d'une autre utilisation de la technique de matrix embedding, avec d'autres codes linéaires. Cette autre utilisation est motivée par une contrainte supplémentaire sur le schéma

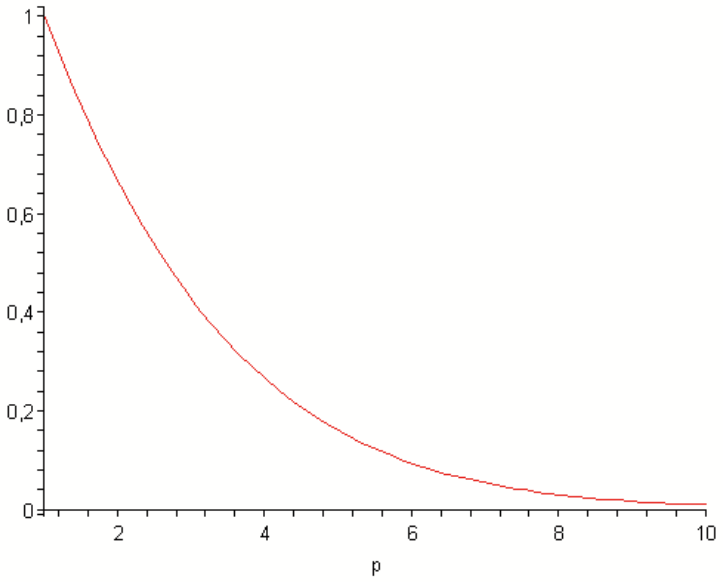

FIG. 7 – Evolution de la taille relative en fonction du paramètre p dans une technique de matrix embedding utilisant les codes de Hamming.

de stéganographie : la règle dite de *sélection non partagée*[4], aussi appelée *écriture sur papier mouillé*. La technique décrite ici est tirée de J. Fridrich, M. Goljan, D. Soukal [7]. Le principe de cette problèmatique, avant d'expliquer la méthaphore des codes à papier mouillé, suppose que l'emetteur du message ne souhaite pas utiliser tout le support disponible afin de cacher son message. Cette supposition est motivée par le fait que celui-ci veut choisir les "endroits" du support à modifier, dans un but de sélectionner les parties de l'image où les modifications se feront le moins sentir. Cette problèmatique est appelée *règle de sélection*. Elle peut être partagée par l'emetteur et le récepteur, cependant, cela suppose alors que la règle est la même pour tous les supports utilisés, ou bien partagée avant chaque communication. Dans les deux cas cela pose un problème soit de sécurité, soit de performance. Ainsi, lorsque cette règle de sélection n'est connue que de l'emetteur, on appelle cela *écriture sur papier mouillé*.

La méthaphore explique le fait que l'emetteur voudrait cacher un message à l'intérieur d'un papier étant resté sous la pluie : celui-ci ne peut pas écrire aux endroits mouillés. Lors de la réception, le papier ayant séché, la personne ne connait pas les endroits où a été caché le message, afin d'extraire celui-ci ; d'où l'idée de règle de sélection non partagée.

Le principe de ne pas pouvoir écrire à certains endroits dans le support entraîne plus de sécurité et minimise encore plus les changements dans le support. Par exemple, dans l'algorithme F5 d'A. Westfeld, décrit partie 4, le support utilisé est la liste des coefficients DCT après quantification et arrondi. Comme règle de sélection, nous pourrions par exemple prendre les coefficients faisant perdre le plus d'information lors de l'arrondi. En effet, un coefficient DCT qui, après quantification, est égal à 3.47, va être arrondi à 3. Le fait qu'il soit choisi pour insérer le message, et donc modifié en 3 ou en 4 ne changera pas grand chose au résultat, car au lieu de perdre 0.47 à cause de la compression JPEG, il perdrait soit 0.47 soit 0.53 à cause de l'insertion. Si l'on ne le discerne pas des autres, il pourra alors être changé en 2 si l'opération de matrix embedding décide de le changer, ce qui entraine une plus grosse modification. Cet exemple de mode de sélection pourrait alors cacher les modifications engendrées par l'insertion dans les modification dues à une compression JPEG.

[4]non-shared selection rule

3.3.2 Description

La technique décrite ici utilise également des codes linéaires. Cependant, dans notre support $x \in \mathbb{F}_2^n$, nous supposons ne pouvoir modifier que k d'entre eux : les $x_j,\ j \in J \subset \{1, ..., n\},$ avec $|J| = k$. Ainsi, les (n-k) bits $x_i,\ i \notin J$ ne peuvent pas être modifiés.

Utilisant la technique de matrix embedding, afin de cacher un message $m \in \mathbb{F}_2^m$, l'emetteur doit modifier x en y de telle sorte que :

$$Dy = m \tag{7}$$

Avec $D \in \mathbb{M}_{m,n}$ la matrice de parité d'un code linéaire, pouvant être retrouvée par l'emetteur. Décomposons le vecteur y en $y = x + v$, où v est le vecteur des modifications apportées à x. L'équation 7 devient donc :

$$Dv = m - Dx \tag{8}$$

De plus, la règle de sélection implique $v_j = 0, \forall j \in J$, d'où $y_j = x_j, \forall j \in J$. Ainsi, les inconnues de l'équation 8 sont au nombre de k : nous pouvons donc élaguer notre système, et enlever les $(n-k)$ colonnes de D, et les $(n-k)$ lignes de v, correspondants aux indices $j \notin J$ (les $(n-k)$ colonnes enlevées de D sont quelcquonques et ne nous interessent pas, puisque elles seront multipliées par les 0 de v). En notant u le vecteur obtenu à partir de v, z le vecteur $m - Dx$, et H la matrice obtenue à partir de D, nous obtenons l'équation suivante :

$$Hu = z \tag{9}$$

qui est un système à k inconnues : $u \in \mathbb{F}_2^k$, et où $H \in \mathbb{M}_{m,k}$ et $z \in \mathbb{F}_2^m$. Ainsi, trouver D revient à trouver H, et le problème à résoudre est le même qu'à l'équation 5 page 11, le vecteur u est le chef de classe de $C(z)$, dont le poids de Hamming est inférieur au rayon de couverture R du code. Cependant, on note une différence, et non des moindres. En effet, dans le problème précédent, nous imposions à H d'être la matrice de parité d'un code linéaire (code de Hamming par exemple) connu de l'emetteur et du récepteur, or ici, H est une sous-matrice de D, qui *dépend* de J, et donc du support. Ainsi, pour deux supports différents, J sera la plupart du temps différent, et donc H également. Une solution est de prendre pour D la matrice de parité d'un code linéaire quelcquonque, et de résoudre le système à l'aide du pivot de Gauss.

En effet, même pour les codes de Hamming vus précédemment, la technique ne marche plus. L'exemple suivant le montre :

Soit $m = (1, 0, 1)$ le message à insérer dans $x = (1, 0, 0, 0, 0, 0, 1)$, et soit $J = \{1, 2, 5, 6, 7\}$, c'est à dire que les bits n°3 et 4 ne peuvent pas être modifiés. Alors on a, si l'on suit la méthode décrite partie 3.2.2 page 11 :

$$\begin{aligned} He = m - Hx &= \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \end{aligned}$$

Sans la règle de sélection, la solution est evidemment $e = (0, 0, 1, 0, 0, 0, 0)$ puisque $m - Hx$ est la 3ème colonne de H. Mais puisque nous ne pouvons pas modifier les bits n°3 et 4, on a $e_3 = e_4 = 0$,

et on peut enlever les colonnes 3 et 4 de H. D'où le système à résoudre :

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} e_1 \\ e_2 \\ e_5 \\ e_6 \\ e_7 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

qui est un système classique à 5 inconnues, résolvable avec les techniques de combinaisons et substitutions, et dont la solution est $e = (1, 1, 0, 0, 0)$. La résolution de ce type de systèmes est de complexité cubique sur le nombre d'équations, donc à éviter en pratique. [7] propose alors de prendre comme matrice D un certain type de matrices creuses aléatoires, entraînant une complexité moindre pour résoudre le système.

En effet, supposons que notre matrice D a une distribution telle que l'on puisse y appliquer la méthode suivante :

- Prendre une colonne comportant exactement un 1, se trouvant à la i_1^{me} ligne. Permuter cette colonne avec la première colonne de la matrice, et la i_1^{me} ligne avec la première ligne de la matrice.
- Dans la matrice privée de la première colonne et de la première ligne, prendre une colonne comportant exactement un 1, se trouvant à la i_2^{me} ligne. Permuter cette colonne avec la deuxième colonne de la matrice, et la i_2^{me} ligne avec la deuxième ligne de la matrice.
- Répéter ces opérations avec la matrice privée des deux premières colonnes et des deux première lignes, et continuer le processus jusqu'à la fin de la matrice.

Ainsi, la matrice permutée obtenue est triangulaire supérieure, et la résolution se fait en "cascade" en partant de la dernière ligne, ne comportant qu'un coefficient. Ce processus est appelé *LT Process*, et est tiré des codes correcteurs appelés *LT Codes*. Si à un moment donné, on ne trouve pas de colonne avec un seul 1 à l'intérieur, alors le processus échoue.

Les auteurs montrent que pour que les matrices puissent être utilisées dans ce processus, il faut que le poids de Hamming de leurs colonnes suivent une distribution de probabilité dite RSD, c'est à dire :
La probabilité qu'une colonne de D ait un poids de Hamming i, $1 \leq i \leq k$ est $\frac{\rho[i]+\tau[i]}{\eta}$, où :

$$\rho[i] = \begin{cases} \frac{1}{m} \text{ si } i = 1 \\ \frac{1}{i(i-1)} \text{ sinon.} \end{cases}$$

et :

$$\tau[i] = \begin{cases} \frac{R}{im} \text{ si } i = 1, .., m/R - 1 \\ \frac{Rln(R/\delta)}{m} \text{ si } i = m/R \\ 0 \text{ si } i = m/R + 1, .., m \end{cases}$$

et :

$$\eta = \sum_{i=1}^{m} (\rho[i] + \tau[i])$$

et :

$$R = cln(m/\delta)\sqrt{m}$$

Avec δ et c des constantes bien choisies.

Afin d'appliquer ceci, l'emetteur peut par exemple générer à partir de la clé secrète une suite de nombre suivant la distribution précédente, les paramètres δ et c pouvant être publics. Il peut alors résoudre facilement le système précédent, et déterminer les changements à effectuer. Lors de l'extraction, le récepteur disposera alors de la même matrice D que l'emetteur, et calculera simplement le syndrome comme dans un schéma de matrix embedding classique. Notons qu'ici aussi, un paramètre spécifique doit nécessairement être transmis : la taille du message. Comme avec les codes de Hamming, où cela concernait le paramètre p de l'algorithme, celui-ci peut par exemple être introduit dans les premiers bits du support.

4 L'algorithme F5

4.1 Introduction

L'algorithme F5 est probablement la première implémentation pratique d'une technique de matrix embedding en stéganographie. Il a été inventé par A. Westfeld, et présenté en 1999 [6]. Le nom provient des précédents algorithmes développés par l'auteur : F3 et F4.
F5 insère le message pendant une compression JPEG, et utilise les LSB des coefficients DCT pour réaliser l'opération de matrix embedding. L'algorithme est sûr face aux attaques visuelles et statistiques car il s'adapte à la distribution spécifiques des coefficients DCT.

Dans un premier temps, nous décrirons brièvement l'algorithme de compression JPEG, puis nous verrons ensuite en détail le fonctionnement de l'algorithme. Ensuite, nous observerons quelques résultats expérimentaux obtenus après une implémentation de F5. Enfin, nous expliquerons une attaque de l'algorithme réalisée par J. Fridrich, M. Goljan, et D. Hogea [3], et présenterons là aussi quelques résultats expérimentaux de l'implémentation de cette attaque.

4.2 Compression JPEG

JPEG est l'acronyme de Joint Photographic Expert Group, qui est le nom de l'organisation qui créa la norme JPEG. C'est aussi le nom du fameux algorithme de compression des images. C'est une méthode avec perte, engendrant des taux de compression de 3 à 100, selon le facteur de qualité exigé.
Il se décompose en six étapes :

- Découpage en blocs de pixels
- Passage du mode RGB au mode YCbCr.
- Sous-echantillonnage sur les composantes de chrominance.
- Application de la Transformation en Cosinus Discrete (DCT).
- Quantification de chaque bloc par une matrice de quantification.
- Codage RLE et Huffman.

Découpage en blocs de pixels L'image est généralement découpée en blocs de 64 (8x8) pixels. Cette taille de blocs précise est celle qui apporte les meilleurs résultats lors de la compression.

Passage du mode RGC au mode YCbCr Chaque bloc de l'image, comportant trois composantes : rouge, vert et bleu, est transformé en un bloc de trois composantes : Y (luminance), Cb (chrominance bleue) et Cr (chrominance rouge) par une opération matricielle.

Sous-echantillonnage sur les composantes de chrominance L'oeil humain étant plus sensible à la luminance qu'aux chrominances, celles-ci sont sous échantillonnées : une moyenne est faite sur chaque bloc de 4x4 pixels. Le gain est ainsi de 50%, et l'oeil humain ne perçoit pas la différence sur des images photographiques. En revanche le résultat est plutôt mauvais sur du texte par exemple.

Application de la Transformation en Cosinus Discrete (DCT) La Transformée en Cosinus Discrete est en fait la partie réelle de la transformée de Fourier discrete. Celle-ci permet de passer du domaine dit *spatial* au domaine *fréquentiel*. Cette transformation s'effectue selon la formule suivante :

$$S_{u,v} = \frac{1}{4} C(u)C(v) \sum_{x=0}^{7} \sum_{y=0}^{7} s_{xy} cos \frac{(2x+1)u\pi}{16} cos \frac{(2y+1)v\pi}{16}$$

avec

$$C(0) = \frac{1}{\sqrt{2}} \text{ et } C(u) = 1 \text{ si } u \neq 0$$

s_{xy} représentant les valeurs des pixels dans le domaine spatial, et $S_{u,v}$ les valeurs des coefficients DCT.

Cette opération permet une représentation fréquentielle de l'image, les coefficients basses fréquences (correspondants à un changement long d'intensité) sont rangés dans le coin supérieur gauche du bloc, et les coefficients hautes fréquences (correspondants à un changement rapide d'intensité) sont rangés dans le coin inférieur droit du bloc. En effet, pour $(u, v) = (0, 0)$, nous observons que le coefficient placé dans le coin supérieur gauche représente la moyenne[5] des intensités dans l'image ; ce coefficient est appelé coefficient DC, alors que tous les autres sont appelés AC.

L'oeil humain étant plus sensible aux basses qu'aux hautes fréquences, nous allons pouvoir effectuer une perte au niveau de ces derniers. La figure 8 représente le passage du domaine spatial au domaine fréquentiel à l'aide de la DCT sur un bloc de pixels codés sur 8 bits (valeur entre 0 et 255).

$$\begin{bmatrix} 52 & 55 & 61 & 66 & 70 & 61 & 64 & 73 \\ 63 & 59 & 55 & 90 & 109 & 85 & 69 & 72 \\ 62 & 59 & 68 & 113 & 144 & 104 & 66 & 73 \\ 63 & 58 & 71 & 122 & 154 & 106 & 70 & 69 \\ 67 & 61 & 68 & 104 & 126 & 88 & 68 & 70 \\ 79 & 65 & 60 & 70 & 77 & 68 & 58 & 75 \\ 85 & 71 & 64 & 59 & 55 & 61 & 65 & 83 \\ 87 & 79 & 69 & 68 & 65 & 76 & 78 & 94 \end{bmatrix} \xrightarrow{\text{DCT}} \begin{bmatrix} -415 & -30 & -61 & 27 & 56 & -20 & -2 & 0 \\ 4 & -22 & -61 & 10 & 13 & -7 & -9 & 5 \\ -47 & 7 & 77 & -25 & -29 & 10 & 5 & -6 \\ -49 & 12 & 34 & -15 & -10 & 6 & 2 & 2 \\ 12 & -7 & -13 & -4 & -2 & 2 & -3 & 3 \\ -8 & 3 & 2 & -6 & -2 & 1 & 4 & 2 \\ -1 & 0 & 0 & -2 & -1 & -3 & 4 & -1 \\ 0 & 0 & -1 & -4 & -1 & 0 & 1 & 2 \end{bmatrix}$$

FIG. 8 – Evolution d'un bloc de pixel lors une Transformation en Cosinus Discrete

Quantification de chaque bloc par une matrice de quantification La DCT ayant été effectuée sur les blocs de chaque composante, l'algorithme de compression va alors quantifier ces coefficients à partir d'une matrice, dépendant d'un facteur de qualité donné par l'utilisateur (généralement entre 0 et 100), puis arrondir ces coefficients à l'entier le plus proche. C'est lors de cette étape que la perte d'information est la plus importante.

Cette matrice (il y en a en fait une pour la composante de luminance, et une pour les deux composantes de chrominance) comporte des coefficients se rapprochant de 10 dans le coin supérieur gauche, et se rapprochant de 100 sur le coin inférieur droit, ces nombres augmentant si le facteur de qualité est bas. Ainsi, lors de la quantification (division terme à terme des blocs de l'image par la matrice de quantification), les coefficients les plus altérés seront ceux correspondant aux hautes fréquences, une majorité d'entre eux seront d'ailleurs transformés en zéro. La figure 9 représente un bloc de coefficients DCT de la composante de luminance après quantification.

15	0	-1	0	0	0	0	0
-2	-1	0	0	0	0	0	0
-1	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

FIG. 9 – Bloc de coefficients DCT de luminance après une quantification

Les histogrammes des coefficients DCT d'une image après compression jpeg comportent des caractéristiques remarquables, comme on peut le voir dans la figure 10 : les coefficients les plus

[5] multipliée par 4 : on ne divise que par 4

importants sont ceux égaux à 0, et on observe une symétrie par rapport à la valeur 0. De plus, les histogrammes des images naturelles présentent plus de coefficients non nuls impairs que pairs (hormis 0).

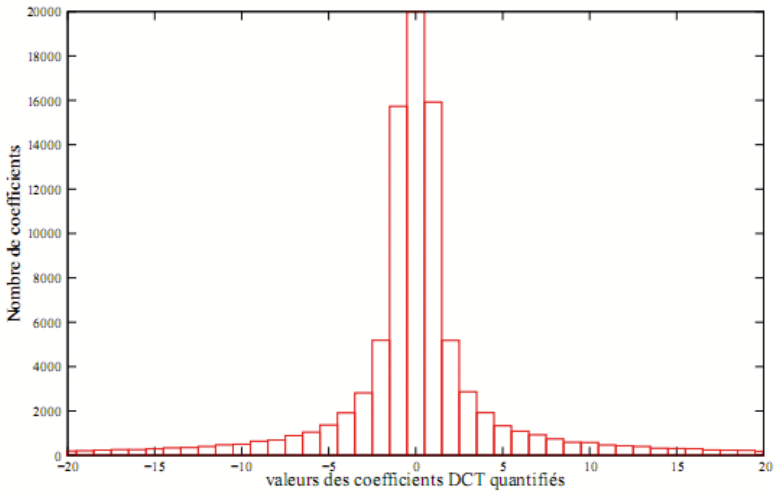

FIG. 10 – Histogramme des coefficients DCT d'une image naturelle

Codage RLE et Huffman Les algorithmes de compression RLE et Huffman sont des algorithmes sans perte. Ils utilisent les statistiques des coefficients DCT et leur corrélation afin de minimiser le coût de codage de l'image.

Le codage RLE permet de coder les plages de coefficients identiques par un seul symbole. Si l'on parcourt les blocs obtenus après la quantification sous forme de zig-zag, on peut facilement voir apparaitre des plages de zéros. Le codage RLE permet alors de gagner de la place.

Le codage de Huffman utilise la probabilité d'occurence de chaque coefficient afin de coder ceux qui aparaissent le plus souvent part des symboles courts, et de coder ceux qui aparaissent le moins souvent par des symboles plus longs, chaque symbole ne devant pas être le début d'un autre. Cette méthode permet d'approcher l'entropie de la source en coût de codage par coefficient.

4.3 Description de l'algorithme

L'algorithme F5 d'A. Westfeld est un algorithme ±1 sur les coefficients DCT quantifiés non nuls d'une image (pendant une compression JPEG), en utilisant la technique de matrix embedding. Afin d'être sûr face aux attaques statistiques, il préserve les caractéristiques de l'histogramme des coefficients DCT d'une image naturelle : lors d'un changement de valeur d'un bit, l'algorithme décrémente la valeur absolue du coefficient (il n'y a alors plus d'"égalisation" des paires de valeurs comme dans un LSB classique). Le seul artéfact produit est une augmentation du nombre de coefficients à 0, cependant, cette caractéristique ressemblera simplement à une compression JPEG avec un facteur de qualité inférieur. De plus, pour préserver la symétrie de l'image, les coefficients positifs impairs et négatifs pairs ont un LSB de valeur 1, alors que les coefficients négatifs impairs et positifs pairs ont un LSB de valeur 0.

L'algorithme présenté par A. Westfeld prend 5 paramètres en entrée pour insérer un message :
- une image servant de support (compressée ou non).
- un message à être inséré.
- le facteur de qualité à être utilisé lors de la quantification.
- un mot de passe afin de générer une marche aléatoire sur le support et le message.
- un commentaire pouvant être inséré dans l'en-tête du fichier JPEG.

Il prend deux paramètres en entrée afin d'extraire un message :
– une image stéganographiée.
– le mot de passe utilisé lors de l'insertion.

Voici le fonctionnement de l'algorithme d'insertion :

1. Effectuer une compression JPEG de l'image, arrêter juste après la quantification.
2. Calculer la capacité C de l'image et en déduire le paramètre p pour effectuer la technique de matrix embedding.
3. Générer une marche pseudo-aléatoire afin de mélanger le support et le message.
4. Insérer dans les premiers bits du support la taille du paramètre p utilisé ainsi que la taille du message, en utilisant simplement une technique LSB.
5. Insérer des morceaux de message de longueur p dans des morceaux de support (coefficients AC non nul) de longueur $2^p - 1$ en utilisant la technique de matrix embedding. Afin de changer la valeur d'un bit, la valeur absolue du coefficient est décrémentée de 1. Si le coefficient devient 0, cette situation est appelée *effondrement*, et les p bits devant être insérés sont insérés à nouveau en retournant à l'étape 5.
6. Si tout le message a pu être inséré, alors le processus a réussi. Sinon, un *warning* est affiché.

Lors de l'extraction, le récepteur n'a alors plus qu'a générer la marche aléatoire, retrouver les paramètres de l'algorithme utilisés, et de calculer le syndrome de chaque bloc de coefficients AC non nuls afin de reconstruire le message.

La capacité C à calculer lors de l'étape 2 est égale à :

$$C = \overbrace{h_{DCT}}^{\text{nombre de coef DCT}} - \overbrace{\frac{h_{DCT}}{64}}^{\text{nombre de coef DC}} - \overbrace{h(0)}^{\text{coef à 0}} - \overbrace{h(1) + 0.49h(1)}^{\text{coef perdus par effondrement}}$$

4.4 Attaque de l'algorithme F5

L'attaque de l'algorithme F5 a été présenté en 2002 par J. Fridrich, M. Goljan, D. Hogea [3]. L'attaque est basée sur le fait de pouvoir reconstruire une estimation de l'histogramme de l'image juste avant l'insertion du message. Ensuite, une analyse des différences entre cette estimation et l'image interceptée permet d'en déduire une approximation de la probabilité de modification d'un coefficient, et donc de la taille du message inséré.

L'attaque se présente donc en deux points :
– estimer l'histogramme de l'image avant insertion des données.
– déduire de cet histogramme la longueur du message inséré.

Estimation de l'histogramme de l'image avant insertion des données

Afin de construire l'estimation de l'histogramme de l'image interceptée avant insertion, l'image est décompressée dans le domaine spatial, décalée de 4 pixels dans les deux directions (horizontalement et verticalement), puis recompressée en utilisant le même facteur de qualité que lors de l'insertion. Ce décalage permet de "casser" la structure des blocs des coefficients DCT, afin d'obtenir une approximation de leur valeur avant la quantification. En outre, un filtre passe-bas est appliqué juste après le décalage, et ce afin d'atténuer les effets de blocs dus à la quantification.

Estimation de la longueur du message inséré

Afin d'estimer la longueur du message inséré, les auteurs calculent d'abord la probabilité β qu'un coefficient AC non nul soit modifié.

Soit $H, h, \hat{h}$ les histogrammes respectifs de l'image stéganographiée, l'image avant insertion de données, et l'estimation de cette derniere. $H(i)$ représentera par exemple le nombre de coefficients de l'image stéganographié dont la valeur absolue est égale à i. De plus, pour un histogramme X de coefficients DCT, nous noterons $X_{k,l}$, $1 \leq k, l \leq 8$, l'histogramme des coefficients se trouvant à la place (k, l) dans le bloc 8x8 (par exemple $H_{0,0}$ est l'histogramme des coefficients DC de l'image stéganographiée)

Alors, nous avons les équations suivantes, pour $1 \leq k, l \leq 8$:

$$H_{k,l}(d) = \begin{cases} \beta h_{k,l}(d+1) + (1-\beta)h_{k,l}(d) \text{ si d} \neq 0, \\ \beta h_{k,l}(1) + h_{k,l}(0) \text{ si d} = 0. \end{cases} \tag{10}$$

En effet, avec une probabilité β, un coefficient de valeur aboslue $d+1$ sera transormé en un coefficient de valeur absolue d, et avec une probabilité $(1-\beta)$, ce coefficient ne sera pas modifié.

Etant donné que nous avons une estimation $\hat{h}$ de h, nous pouvons calculer $\hat{H}$: l'histogramme de l'image stéganographiée à partir de l'estimation obtenue de l'histogramme de l'image, en appliquant l'équation 10 pour $h = \hat{h}$:

$$\hat{H}_{k,l}(d) = \begin{cases} \beta \hat{h}_{k,l}(d+1) + (1-\beta)\hat{h}_{k,l}(d) \text{ si d} \neq 0, \\ \beta \hat{h}_{k,l}(1) + \hat{h}_{k,l}(0) \text{ si d} = 0. \end{cases}$$

Ainsi, nous cherchons β qui va minimiser les différences entre l'histogramme estimé $\hat{H}$ et celui intercepté H. Pour cela, les auteurs utilisent la méthode des moindres carrés. De plus, les calculs ne sont réalisés que pour $d = 0$ et $d = 1$, ces deux valeurs étant les plus touchées dans l'insertion par l'algorithme F5. En outre, étant donné que nous avons 8x8 résultats possibles pour β (un résultat pour chaque couple (h, k)), les auteurs choisissent de prendre la moyenne des résultats obtenus pour les cases $(2,1), (1,2), (2,2)$, correspondant aux coefficients AC des basses fréquences. Ceci s'explique par le fait que le décalage de 4 pixels des deux côtés engendre des discontinuités au milieu de chaque bloc (à cause de l'"effet bloc" de la compression JPEG), donc des hautes fréquences non voulues.

L'approximation par les moindres carrés donne la formule suivante pour les $\beta_{k,l}$:

$$\beta_{k,l} = \frac{\hat{h}_{k,l}(1)[H_{k,l}(0) - \hat{h}_{k,l}(0)] + [H_{k,l}(1) - \hat{h}_{k,l}(1)][\hat{h}_{k,l}(2) - \hat{h}_{k,l}(1)]}{\hat{h}^2_{k,l}(1) + [\hat{h}_{k,l}(2) - \hat{h}_{k,l}(1)]^2}$$

et :

$$\beta = \frac{\beta_{2,1} + \beta_{1,2} + \beta_{2,2}}{3}$$

A partir de β (la probabilité de changement d'un coefficient AC non nul), nous pouvons obtenir la taille du message inséré.
En effet, si M est la longueur du message, on a :

M = Efficacité d'insertion × Nombre de changements par matrix embedding

L'efficacité d'insertion étant le nombre de bits pouvant être insérés avec un seul changement, il est égal à $p\frac{2^p}{2^p-1}$ (cf 3.2.3).

Le nombre de changements par matrix embedding m se calcule de la manière suivante :
Si n est le nombre total de changements, s le nombre de changements à cause des effondrements, P le nombre total de coefficients AC non nuls, et P_s la probabilité qu'un coefficient entraine un effondrement, on a :
$P_s = \frac{h(1)}{P}$, donc $s = nP_s$.
De plus, on a $n = m + s \Leftrightarrow m = n(1 - P_s) \Leftrightarrow m = n(1 - \frac{h(1)}{P})$.

Et enfin, $n = \beta P$.

D'où la formule pour M :

$$M = \frac{2^p}{2^p - 1} p\beta(P - h(1))$$

4.5 Implémentations

Durant ce projet, j'ai été amené à implémenter certains algorithmes, et donc à développer 4 exécutables :

- compress : pour simuler une compression jpeg.
- embed : pour insérer un message dans une image en utilisant l'algorithme F5 d'A. Westfeld.
- extract : pour extraire un message inséré avec le précedent exécutable.
- attack : une implémentation de l'attaque de l'algorithme F5.

Ces programmes ont été écrits en langage C++, car c'est un langage rapide, avec beaucoup de possibilités, et où je possédais également quelques sources, notamment pour le traitement des images. Ils fonctionnent sous GNU/Linux et un makefile est fourni pour compiler le tout. La figure 11 illustre l'espace de travail des programmes écrits.

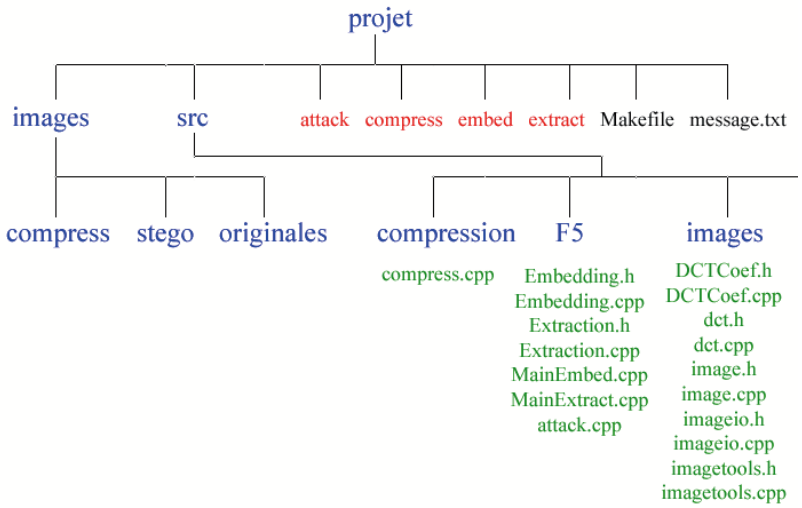

FIG. 11 – Architecture de l'espace de travail des programmes écrits. En bleu sont représentés les dossiers, en rouge les programmes, en vert les codes sources, et en noir les autres fichiers.

Le format de fichier JPEG étant plutôt complexe à maîtriser, et celui-ci n'étant pas le sujet principal du projet, j'ai dû contourner ce problème en stockant les données sous forme d'une liste de coefficients DCT, dans un fichier avec extension ".jpegLike". Lorsque le besoin d'afficher ces images se ressentait, il était alors nécessaire de les décompresser afin de les restituer dans leur format d'origine : ".pgm".

4.6 Implémentation et résultats de l'algorithme F5

Afin d'implémenter l'algorithme F5 vu partie 4, deux classes ont été créées : Embedding pour l'insertion, et Extraction pour l'extraction. La modélisation UML de ces classes est représentée

figure 12.

Le programme principal pour l'insertion est le fichier *mainEmbed.cpp*. Il génère un exécutable *embed* dont l'usage est le suivant :

./embed [imageIn] [imageOut] [message] [password] [qualityFactor]

Où :
imageIn est le chemin de l'image servant de support.
imageOut est le chemin de l'image générée en sortie.
message est le chemin du fichier contenant le message.
password est le mot de passe nécessaire.
qualityFactor est le facteur de qualité requis lors de la compression jpeg.

Par exemple :

./embed images/originales/lena.pgm images/stego/lena.pgm message.txt 007 80

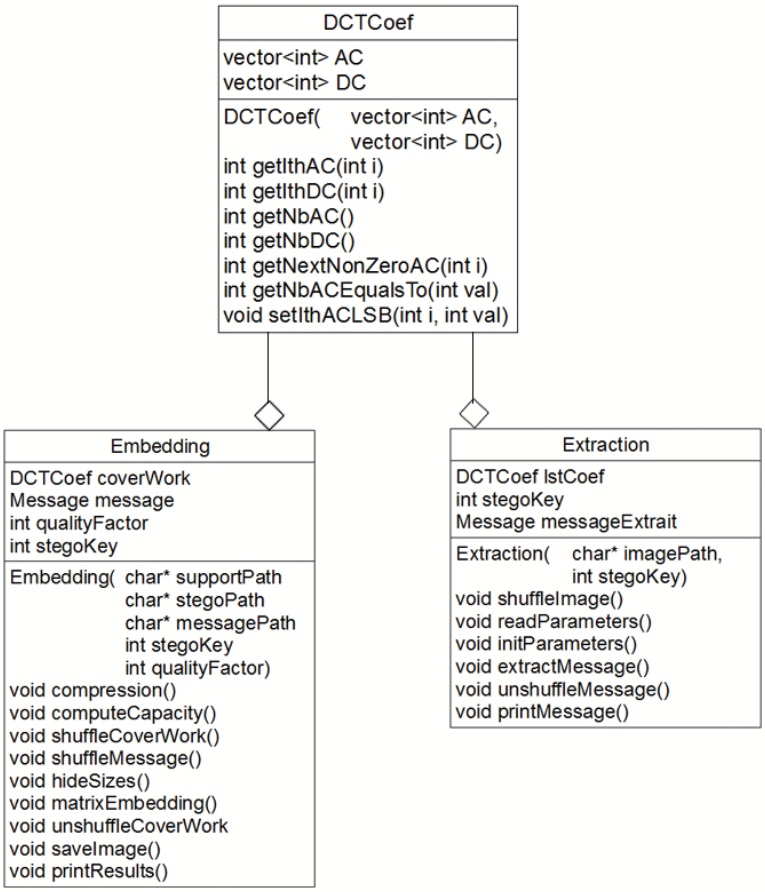

FIG. 12 – Diagramme de classe des classes Embedding et Extraction

En sortie, le programme génère un fichier .pgm et un fichier.jpegLike portant le nom de l'image de sortie, qui elle est l'image stéganographiée décompréssée afin de voir le résultat.

Le programme principal pour l'extraction est le fichier *mainExtract.cpp*. Il génère un exécutable *extract* dont l'usage est le suivant :

./extract [imageIn] [password]

Où :
imageIn est l'image stéganographiée décompressée
password est le mot de passe nécessaire.

NB : L'image décompressée au format pgm doit être accompagnée (dans le même dossier) du fichier .jpegLike compressée.

Par exemple :

./extract images/stego/lena.pgm 007

Le programme ne génère rien en sortie, il se contente d'afficher le message extrait.

Ces deux programmes utilisent la classe *DCTCoef* qui est une structure permettant de stocker une suite de coefficients DCT (cf figure 12 pour le diagramme UML). La structure *Message* est en fait un *vector<int>*.

Résultats

La figure 13 présente l'image Goldhill.pgm après une simple compression (facteur de qualité 70), et après une compression puis insertion d'un message de 10 kbits, ce qui représente environ 25% de la capacité du support. On remarque que le changement n'est pas perceptible pour l'oeil humain.

La figure 14 représente la différence des histogrammes des coefficients DCT des deux images précédentes. Malgrès un fort pic au niveau des coefficients 0, cela reste convenable car le nombre de différences est d'environ 2000, ce qui ne représente pas grand chose sur les 512x512 = 262144 coefficients de l'image.

FIG. 13 – image goldhill après une compression simple (a) et après insertion d'un message (b).

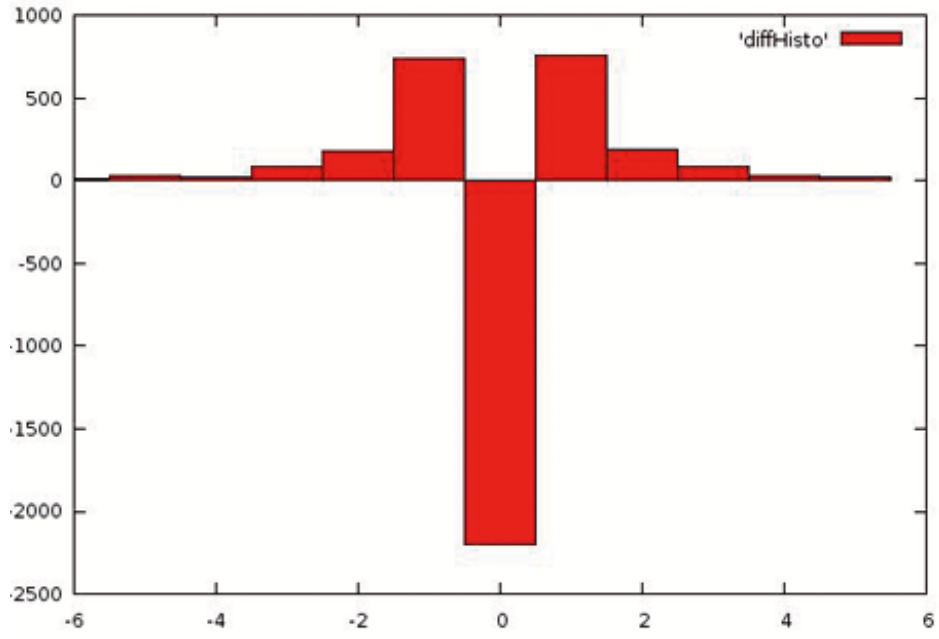

FIG. 14 – Différence des histogrammes des deux images précédentes.

4.7 Implémentation et résultats de l'attaque de l'algorithme F5

L'attaque de l'algorithme F5 est implémentée dans le fichier *attack.cpp* et génère un exécutable *attack* dont l'usage est le suivant :

./attack [imageIn] [qualityFactor]

Où

imageIn est l'image stéganographiée décompressée

qualityFactor est le facteur de qualité utilisé lors de l'insertion.

Par exemple :

./attack images/stego/lena.pgm 80

L'attaque a été testée sur plusieurs images pgm en niveaux de gris, de résolution 512x512, avec un facteur de qualité entre 80 et 90. Le tableau 15 récapitule les résultats, en mettant en parallèle les valeurs de β et la longueur du message inséré obtenus lors de l'insertion, et ceux obtenus après estimation.

Image	β	$\hat{\beta}$	n	$\hat{n}$
baboon	0.11	0.12	33456	30831
goldhill	0.26	0.24	33744	30847
house	0.16	0.16	5648	5597
lena	0	0.03	0	8755
sailboat	0.29	0.17	45072	34341
tiffany	0	-0.002	0	0

FIG. 15 – Résultats obtenus pour l'attaque de l'algorithme F5 sur différentes images.

5 Conclusion

En conclusion, nous avons pu voir que la construction d'un schéma de stéganographie n'était pas une chose aisée, avec la nécessité de modifier *peu* et surtout d'*une bonne manière* le support, afin de cacher l'existence d'un message. Nous avons également pu voir que l'utilisation des codes correcteurs en stéganographie, et notemment des codes linéaires, était un bon moyen de réduire le nombre de changements effectués dans un support, et pouvait s'adapter à la règle de sélection non partagée. D'autre part, les images JPEG, avec la liste des coefficients DCT comme support, sont beaucoup utilisées en stéganographie, du fait du nombre important d'échanges d'images sous ce format. Ainsi, un bon algorithme de stéganographie se doit de préserver les statistiques des coefficients DCT. L'algorithme F5 est un bon exemple de combinaison des techniques de matrix embedding et de préservation de l'histogramme de l'image. Cependant, il entraîne certaines modifications spécifiques, utilisées pour l'attaque de celui-ci.

Une amélioration possible des schémas de stéganographie réside dans la robustesse que l'on peut y apporter : garder le message inséré intact après altération de l'image par un attaquant : compression, redimensionnement, ...etc. Cette robustesse est inexistante dans les schémas vus dans ce projet, ceux-ci utilisant les LSB des coefficients DCT, ils sont extrêmement sensibles à la moindre modification.

Enfin, ce projet a été me concernant quelque chose de positif, qui m'a permis d'apprendre des choses intéressantes, notemment sur la théorie de l'information, les codes correcteurs, la compression jpeg, et le traitement des images, où j'ai pu approfondir et poursuivre certaines notions vues dans mon cursus. J'ai également été confronté à l'étude d'articles de recherche plutôt récents, et à la rédaction de documents avec LaTeX, ce qui est un plus concernant mon projet professionnel qui est la recherche informatique.

6 Références bibliographiques

Références

[1] J. Barbier : Analyse de canneaux de communication dans un contexte non coopératif, 2007 134-172.

[2] I.J. Cox, M.L. Miller, J.A. Bloom, J. Fridrich, T. Kalker : Digital Watermarking and Steganography (second edition) 425-467.

[3] J. Fridrich, M. Goljan, D. Hogea : Steganalysis of JPEG Images : Breaking the F5 Algorithm.

[4] C. Cachin, An Information-Theoretic Model for Steganography.

[5] J. Fridrich, P. Lisonek, D. Soukal : On Steganographic Embedding Efficiency.

[6] A. Westfeld : F5 - A Steganographic Algorithm. High Capacity Despite Better Steganalysis, 1999.

[7] J. Fridrich, M. Goljan, D. Soukal : Efficient Wet Paper Codes.

[8] http ://www.w3.org/Graphics/JPEG/itu-t81.pdf.

A Fichiers source

A.1 DCTCoef.h

```
/**
        * Name : DCTCoef
        * author : R mi Watrigant
        * date : 04/05/09
        * description : Classe contenant les coefficients DCT quantifi s d'une imag
        *
**/

#ifndef DCTCoef_H
#define DCTCoef_H

#include <vector>
#include <iostream>
        using namespace std;

class DCTCoef {

        private :

                vector<int>     AC;             //liste des coef AC
                vector<int>     DC;             //liste des coef DC

                /**
                        * lsb
                        * retourne le lsb d'un entier
                        * \param n l'entier
                        *
                **/
                int lsb(int n);

        public :

                /**
                        * DCTCoef
                        * constructeur par d faut
                        *
                **/
                DCTCoef();


                /**
                        * DCTCoef
                        * constructeur par valeur
                        * \param AC vecteur des coef AC
                        * \param DC vecteur des coef DC
                        *
                **/
                DCTCoef(vector<int> AC, vector<int> DC);

                /**
```

```
         *
         * accesseur vecteur AC
         *
**/
vector<int> getAC();

/**
         *
         * accesseur vecteur DC
         *
**/
vector<int> getDC();

/**
         *
         * mutateur vecteur AC
         * \param AC le vecteur des coef AC
         *
**/
void setAC(vector<int> AC);

/**
         *
         * mutateur vecteur DC
         * \param DC le vecteur des coef DC
         *
**/
void setDC(vector<int> DC);

/**
         *
         * getNextNonZeroAC
         * retourne le prochain coef ac non nul
         * \param i l'indice    partir d'o   l'on recherche
         *
**/
int getNextNonZeroAC(int i);

/**
         *
         * getNextNonZeroAC
         * retourne l'indice du prochain coef ac non nul
         * \param i l'indice    partir d'o   l'on recherche
         *
**/
int getIthNextNonZeroAC(int i);

/**
         *
         * getNbDC
         * retourne le nombre de coefficients DC
         *
**/
int getNbDC();

/**
```

```
        *
        * getNbAC
        * retourne le nombre de coef AC
        *
**/
int getNbAC();

/**
        *
        * getNbCoef
        * retourne le nombre de coef
        *
**/
int getNbCoef();

/**
        *
        * getNbACEqualsTo
        * retourne le nombre de coefficients AC gaux la valeur
        * \param val valeur tester
        *
**/
int getNbACEqualsTo(int val);

/**
        *
        * getIthAC
        * retourne le ieme coef AC
        * \param i indice du coef retourner
        *
**/
int getIthAC(int i);

/**
        *
        * setIthAC
        * modifie le ieme coef AC
        * \param i l'indice du coef a modifier
        * \param val la valeur mettre la place
        *
**/
void setIthAC(int i, int val);

/**
        *
        * setIthACLSB
        * modifie le lsb de ieme coef AC
        * \param i l'indice du coef a modifier
        * \param val la valeur du lsb mettre la place
        *
**/
void setIthACLSB(int i, int val);

/**
        *
        * getIthDC
```

```
                * retourne le ieme coef DC
                * \param i l'index du coef DC que l'on recherche
                *
            **/
            int getIthDC(int i);
};

#endif
```

A.2 DCTCoef.cpp

```
/**
 * Name : DCTCoef
 * author : R mi Watrigant
 * date : 04/05/09
 * description : Classe contenant les coefficients DCT quantifi s d'une imag
 *
**/

#include "DCTCoef.h"
#include <cstdlib>
#include <stdexcept>
#include <math.h>


/**
 * DCTCoef
 * constructeur par d faut
 *
**/
DCTCoef::DCTCoef() {

}


/**
 * DCTCoef
 * constructeur par valeur
 * \param AC vecteur des coef AC
 * \param DC vecteur des coef DC
 *
**/
DCTCoef::DCTCoef(vector<int> AC, vector<int> DC) {

    for (int i = 0 ; i < AC.size() ; i++)
    {
        (this->AC).push_back(AC[i]);
    }

    for (int i = 0 ; i < DC.size() ; i++)
    {
        (this->DC).push_back(DC[i]);
    }
}

/**
 *
 * accesseur vecteur AC
 *
**/
vector<int> DCTCoef::getAC() {

    return AC;
}
```

```
/**
        *
        * accesseur vecteur DC
        *
**/
vector<int> DCTCoef::getDC() {

        return DC;
}

/**
        *
        * mutateur vecteur AC
        * \param AC le vecteur des coef AC
        *
**/
void DCTCoef::setAC(vector<int> AC) {

        (this->AC).clear();

        for (int i = 0 ; i < AC.size() ; i++)
        {
                (this->AC).push_back(AC[i]);
        }
}

/**
        *
        * mutateur vecteur DC
        * \param DC le vecteur des coef DC
        *
**/
void DCTCoef::setDC(vector<int> DC) {

        (this->DC).clear();

        for (int i = 0 ; i < DC.size() ; i++)
        {
                (this->DC).push_back(DC[i]);
        }
}

/**
        *
        * getNextNonZeroAC
        * retourne le prochain coef ac non nul
        * \param depart l'indice    partir d'o   l'on recherche
        *
**/
int DCTCoef::getNextNonZeroAC(int depart) {

        if ((depart<0) || (depart>=AC.size()))
        {
                //envoie d'exception
                cout << "i=" << depart << endl;
                throw std::length_error("DCTCoef::getNextNonZeroAC :
```

```
                exit(-1);
        }

        int i = depart;

        do
        {
                i++;

                if (i >= AC.size() )
                {
                        //envoie d'exception
                        throw std::overflow_error("DCTCoef::getNextNo
                }
        }while (AC[i] == 0);

        return AC[i];

}

/**
        *
        * getNextNonZeroAC
        * retourne l'indice du prochain coef ac non nul
        * \param depart l'indice     partir d'o   l'on recherche
        *
**/
int DCTCoef::getIthNextNonZeroAC(int depart) {

        if ((depart<0) || (depart>=AC.size()))
        {
                //envoie d'exception
                cout << "i=" << depart << endl;
                throw std::length_error("DCTCoef::getNextNonZeroAC :
        }

        int i = depart;

        do
        {
                i++;

                if (i >= AC.size() )
                {
                        //envoie d'exception
                        throw std::overflow_error("DCTCoef::getNextNo
                }
        }while (AC[i] == 0);

        return i;

}


/**
        *
```

```
        * getNbDC
        * retourne le nombre de coefficients DC
        *
**/
int DCTCoef::getNbDC() {

        return DC.size();
}

/**
        *
        * getNbAC
        * retourne le nombre de coef AC
        *
**/
int DCTCoef::getNbAC() {

        return AC.size();
}


/**
        *
        * getNbCoef
        * retourne le nombre de coef
        *
**/
int DCTCoef::getNbCoef() {

        return getNbAC()+getNbDC();
}


/**
        *
        * getNbACEqualsTo
        * retourne le nombre de coefficients AC  gaux     la valeur
        * \param val valeur    tester
        *
**/
int DCTCoef::getNbACEqualsTo(int val) {

        int compteur = 0;

        for (int i = 0 ; i < AC.size() ; i++)
        {
                if (abs(AC[i]) == val)
                {
                        compteur++;
                }
        }

        return compteur;
}

/**
```

```
        *
        * getIthAC
        * retourne le ieme coef AC
        * \param i indice du coef    retourner
        *
**/
int DCTCoef::getIthAC(int i) {

        return AC[i];
}

/**
        *
        * setIthAC
        * modifie le ieme coef AC
        * \param i l'indice du coef a modifier
        * \param val la valeur    mettre    la place
        *
**/
void DCTCoef::setIthAC(int i, int val) {

        AC[i] = val;
}

/**
        *
        * setIthACLSB
        * modifie le lsb de ieme coef AC
        * \param i l'indice du coef a modifier
        * \param val la valeur du lsb    mettre    la place
        *
**/
void DCTCoef::setIthACLSB(int i, int val) {

        if ((val != 0) && (val != 1) )
        {
                throw runtime_error("DCTCoef::setIthACLSB : la valeu
        }

        if (val != lsb( getIthAC(i)) )
        {
                int newVal = (getIthAC(i) >= 0) ? getIthAC(i)+1 : get
                setIthAC(i, newVal);
        }


        if (val != lsb( getIthAC(i)))
        {
                throw( std::runtime_error("DCTCoef::setIthACLSB : mo
exit(-1);
        }
}


/**
        *
```

```
        * getIthDC
        * retourne le ieme coef DC
        * \param i l'index du coef DC que l'on recherche
        *
**/
int DCTCoef::getIthDC(int i) {
        return DC[i];
}


/**
        * lsb
        * retourne le lsb d'un entier
        * \param n l'entier
        *
**/
int DCTCoef::lsb(int n) {

        //positif + impair OU negatif + pair => 1
        //positif + pair OU negatif + impair => 0
        if (((n > 0) && (n%2 != 0)) || ((n < 0 ) && (n%2 == 0)))
        {
                return 1;
        }
        else
        {
                return 0;
        }

}
```

A.3 compress.cpp

```
#include "../image/image.H"
#include "../image/imageio.H"
#include "../image/imagetools.H"
#include "../image/dct.h"


#include <cstdlib>
#include <stdio.h>
#include <cstring>
#include <fstream>
#include <math.h>
#include <string>
#include <iostream>
    using namespace std;


char* IMAGE_IN;
char* IMAGE_OUT;
int QUALITY_FACTOR;

/**
        * proccessOptionsInLine
        * traite les options d'ex cution
        * \param argc
        * \param argv
        *
**/
void processOptionsInLine(int argc, char** argv)
{
        if ((argc != 4))
        {
                cout << "\nUsage : " << argv[0] << " imageIn.pgm imageOut.pgm Quality
                exit(-1);
        }

        IMAGE_IN = new char[200];
        IMAGE_OUT = new char[200];

        //recopie des param tres dans les variables
        strcpy(IMAGE_IN, argv[1]);

        strcpy(IMAGE_OUT, argv[2]);

        char* qualityFactor = new char[3]; strcpy(qualityFactor, argv[3]);
        QUALITY_FACTOR = atoi(qualityFactor);

        if ((QUALITY_FACTOR < 0 ) || (QUALITY_FACTOR > 100))
        {
                cout << "\nError : le facteur de qualit  doit  tre  compris entre 0
                exit(-1);
        }

}
```

```
/**
        * compress
        * simule une compression jpeg
        * \param imageIn image d'entr e
        * \param QF facteur de qualit . 0 < QF < 100
        *
**/
Image<octet>* compress(Image<octet> imageIn, int QF)
{
        Image<octet>* imageOut = new Image<octet>(imageIn.NbRow(), imageIn.NbCol());


        int height = imageIn.NbRow();
        int width = imageIn.NbCol();

        //construction de la table de quantification
        int tabRef[8*8] = {              16, 11, 10, 16,  24,  40,  51,
61,
                                            12, 12, 14, 19,  26,  58,
60,  55,
                                    14, 13, 16, 24,  40,  57,  69,  56,
                                            14, 17, 22, 29,  51,  87,
80,  62,
                                            18, 22, 37, 56,  68,  109, 103, 77,
                                            24, 35, 55, 64,  81,  104, 113, 92,
                                    49, 64, 78, 87,  103, 121, 120, 101,
                                    72, 92, 95, 98,  112, 100, 103, 99  };


        //param tre c(Q)
        double cQ;

        if (QF < 50)
        {
                cQ = 50000/(QF+1);
        }
        else
        {
            cQ = 200 - 2*QF;
        }

    double* tabQuant = new double[8*8];


        for (int i = 0 ; i < 8*8 ; i++)
        {
                tabQuant[i] = tabRef[i]*cQ/100 ;
        }

        for (int row = 0 ; row < height/8 ; row++)
        {

                for (int col = 0 ; col < width/8 ; col++)
                {
                        //premier   lement   du bloc
```

```
                        int i0 = row*8*width+col*8;

                        //remplissage des coef

                        double* bloc = new double[8*8];
                        for (int x = 0; x < 8 ; x++)
                        {
                                for (int y = 0 ; y < 8 ; y++)
                                {
                                        bloc[8*y+x] = imageIn[i0+y*width+x];
                                }
                        }

                        //dct conversion
                        dct::Data2DCT88(bloc);


                        //Division
                        for (int i = 0 ; i < 8*8 ; i++)
                        {
                                bloc[i] = round(bloc[i]/tabQuant[i]);
                                bloc[i] *= tabQuant[i];
                        }

                        //inverse dct
                        dct::DCT882Data(bloc);

                        //insertion des valeurs dans l'image de sortie

                        for (int x = 0; x < 8 ; x++)
                        {
                                for (int y = 0 ; y < 8 ; y++)
                                {
                                        //recadrage des valeurs entre 0 et 255
                                        (*imageOut)[i0+y*width+x] = (bloc[8*y+x] < 0
                                }
                        }
                }
        }
        return imageOut;
}


/**
        * main
        * Programme principal...
        *
**/
int main(int argc, char** argv)
{

        /*TRAITE LES OPTIONS*/
                processOptionsInLine(argc, argv);


                //LECTURE DE L'IMAGE D'ENTREE
```

```
                Image<octet> imageIn;
                ImageIO<octet>::ReadPGM(IMAGE_IN, imageIn);
                Image<octet>* imageOut = new Image<octet>(imageIn.NbRow(), imageIn.N

                //COMPRESSION
                imageOut = compress(imageIn, QUALITY_FACTOR);

                //SAUVEGARDE DE L'IMAGE
                ImageIO<octet>::WritePGM(IMAGE_OUT, *imageOut);

                cout << "\nFichier compress dans : " ; for (int i = 0 ; i < 200 ;

    return 0;
}
```

A.4 Embedding.h

```
/**
    * Name : Embedding
    * author : R mi Watrigant
    * date : 30/04/09
    * Description : ins re un message dans une image en utilisant l'algorithme F5
       * de Westfeld
**/

#ifndef EMBEDDING_H
#define EMBEDDING_H

#include "../image/image.H"
#include "../image/imageio.H"
#include "../image/imagetools.H"
#include "../image/dct.h"
#include "../image/DCTCoef.h"


#include <fstream>
#include <math.h>
#include <iostream>
        using namespace std;

typedef vector<int> Message;



class Embedding {

        private :

                /**
                        * param tres d'entr e/sorties de l'algo
                **/
                Image<octet> imageIn;
                Image<octet> imageOut;
                int qualityFactor;
                int stegoKey;

                /**
                        * stats
                **/
                int sizeMessage;
                int sizeImage;
                double relative_length;
                int nb_usable_coef;
                int nb_of_bits_embedded;
                int NOM_ME;             // Number Of Modifications due to Matrix Emb
                int NOM_SK;     // Number Of Modifications due to SKrinkage
                int NOM_all;    // NOM_ME + NOM_SK
                double embedding_efficiency ;
                double relative_nb_of_modifications     ;               // Relative
```

```
/**
        * le message et le coverWork dans l'algo
**/
Message message;
DCTCoef coverWork;

/**
        * table de quantification pour la compression
**/
double* tabQuant;


/**
        * les param tres techniques de l'algo
**/
int parameter_S;
int parameter_N;
int nbChangements;
int** H;

/**
        * stockage des bijections pour m langer le message et le co
**/
int* coverWorkBijection;
int* coverWorkBijectionInverse;
int* messageBijection;
int* messageBijectionInverse;


/**
        * initH
        * initialise la matrice H utilis e dans l'algo
        *    partir du param tre n et s
        *
**/
void initH();

/**
        * convertIntToBin
        * convertit un nombre en un nombre en base 2
        * \param number le nombre    convertir
        * \param binary param tre d'entr e/sortie
        * \param size le nombre de bits sur lesquel coder le nombre
        *
**/

void convertIntToBin(int number, int* binary, int size);

/**
        * convertCharToBin
        * convertit un caract re en un nombre en base 2
        * \param mess le caract re    convertir
        * \param binary param tre d'entr e/sortie
        *
**/
```

```
        void convertCharToBin(char mess, int* binary);

        /**
                * buildBijection
                * cr     une bijection de [1, n] dans lui m me
                * \param n borne max de l'ensemble
                *
        **/
        int* buildBijection(int n);

        /**
                * getRandomInt
                * retourne un entier al atoire entre [0, bSup]
                * \param bSup la borne sup de l'intervalle
                *
        **/
        int getRandomInt(int bSup);

        /**
                * lsb
                * retourne le lsb d'un entier
                * \param n l'entier
                *
        **/
        int lsb(int n);


public :

        /**
                * Constructeur
                * initialisation des param tres du programme
                * \param supportPath chemin de l'image du support
                * \param stegoPath chemin de l'image de sortie du programme
                * \param messagePath chemin du fichier contenant le message
                * \param stegoKey mot de passe necessaire pour d finir les
                * \param qualityFactor facteur de qualit  de compression jp
                *
        **/
        Embedding(      char* supportPath,
                                char* stegoPath,
                                char* messagePath,
                                int stegoKey,
                                int qualityFactor );

        /**
                * destructeur
        **/
        ~Embedding();

        /**
                * printResults
                * Affiche des information sur l'insertion
        **/
        void printResults();
```

```
/**
        * setIsMessageTooLong
        * indique que la taille du message est trop longue
**/
void setIsMessageTooLong(bool isTooLong);

/**
        * compress
        * simule une compression jpeg jusqu'  la quantification sur
        *
**/
void compression();


/**
        * computeCapacity
        * calcule la capacit  d'insertion sur l'image d'entr e,
        * le param tre d'insertion, la taille du message dans le su
        *
**/
void computeCapacity();


/**
        * shuffleCoverWork
        * m lange al atoirement le support    l'aide du mot de pas
        *
**/
void shuffleCoverWork();


/**
        * shuffleMessage
        * m lange le message    l'aide du mot de passe
        *
**/
void shuffleMessage();


/**
        * hideSizes
        * cache le param tre de l'algo et la taille du message dans
        * dans les premiers bits du support, en utilisant la techniq
        *
**/
void hideSizes();


/**
        * matrixEmbedding
        * ins re le message en utilisant la technique de matrixEmbe
        *
**/
void matrixEmbedding();
```

```
        /**
                * unShuffleCoverWork
                * r arrange le coverWork dans l'ordre
                *
        **/
        void unShuffleCoverWork();


        /**
                * saveStegoImage
                * sauvegarde l'image .jpegLike et l'image decompressee .pgm
                * \param stegoPath le chemin de l'image de sortie
                *
        **/
        void saveImage(char* stegoPath);

        /**
                * saveHistogram
                * enregistre l'histogramme des coef AC
                * \para histoPath le chemin du fichier    etre enregistr
        **/
        void saveHistogram(char* histoPath);


        /**
                * fonction de test des m thodes
                *
        **/
        void test();


};


#endif
```

A.5 Embedding.cpp

```
/**
     * Name : Embedding
     * author : R mi Watrigant
     * date : 30/04/09
     * Description : ins re un message dans une image en utilisant l'algorithme F5
         * de Westfeld
**/

#include "Embedding.h"
#include <math.h>
#include <stdexcept>


/**
         * Constructeur
         * initialisation des param tres du programme
         * \param supportPath chemin de l'image du support
         * \param stegoPath chemin de l'image de sortie du programme
         * \param messagePath chemin du fichier contenant le message    ins rer
         * \param stegoKey mot de passe necessaire pour d finir les marches al atoi
         * \param qualityFactor facteur de qualit  de compression jpeg
         *
**/
Embedding::Embedding(    char* supportPath,
                                                    char* stegoPath,
                                                    char* messagePath,
                                                    int stegoKey,
                                                    int qualityFactor ) {

        this->stegoKey = stegoKey;
        this->qualityFactor = qualityFactor;

        //LECTURE DE L'IMAGE D'ENTREE
        ImageIO<octet>::ReadPGM(supportPath, imageIn);

        //LECTURE DU MESSAGE
        FILE* fichier = fopen(messagePath, "r");
    char currCar;
    vector<char> messageString;

    if (fichier != NULL)
    {
        do
        {
            currCar = fgetc(fichier); // On lit le caract re
                        messageString.push_back(currCar);

        } while (currCar != EOF); // On continue tant que fgetc n'a pas retourn  EO

        fclose(fichier);
    }
        else
        {
```

```
                cout << "Probl me d'ouverture du message\n";
                exit(-1);
        }

        //TRANSFORMATION DU MESSAGE EN BINAIRE
        int* binary = new int[8];
        int b;
        for (int i=0, b=0 ; i < messageString.size() ; i++, b+=8)
        {
                convertCharToBin(messageString[i], binary);      //on convertit le me
                for (int j = 0 ; j < 8 ; j++) message.push_back(binary[j]) ;     //on
        }


}

/**
        * destructeur
**/
Embedding::~Embedding() {

        delete tabQuant;
        delete coverWorkBijection;
        delete coverWorkBijectionInverse;
        delete messageBijection;
        delete messageBijectionInverse;

        for (int i = 0 ; i < parameter_N ; i++)
        {
                delete H[i];
        }
        delete H;

}


/**
        * compress
        * simule une compression jpeg jusqu'  la quantification sur l'image d'entr
        * sauvegarde les coefficients DCT dans une structure DCTCoef
        *
**/
void Embedding::compression() {

        int tabRef[8*8] = {              16, 11, 10, 16,  24,  40,  51,
61,
                                         12, 12, 14, 19,  26,  58,  60,
55,
                           14, 13, 16, 24,  40,  57,  69,  56,
                                         14, 17, 22, 29,  51,  87,  80,
62,
                                         18, 22, 37, 56,  68,  109, 103, 77,
                                         24, 35, 55, 64,  81,  104, 113, 92,
                           49, 64, 78, 87,  103, 121, 120, 101,
                           72, 92, 95, 98,  112, 100, 103, 99  };
        vector<int> DC;
```

```
    vector<int> AC;

    int height = imageIn.NbRow();
    int width = imageIn.NbCol();

    //construction de la table de quantification

    //param tre c(Q)
    double cQ;

    if (qualityFactor < 50)
    {
            cQ = 50000/(qualityFactor+1);
    }
    else
    {
        cQ = 200 - 2*qualityFactor;
    }

tabQuant = new double[8*8];


    for (int i = 0 ; i < 8*8 ; i++)
    {
            tabQuant[i] = tabRef[i]*cQ/100 ;
    }

    for (int row = 0 ; row < height/8 ; row++)
    {

            for (int col = 0 ; col < width/8 ; col++)
            {
                    //premier   lement   du bloc
                    int i0 = row*8*width+col*8;

                    //remplissage des coef

                    double* bloc = new double[8*8];
                    for (int x = 0; x < 8 ; x++)
                    {
                            for (int y = 0 ; y < 8 ; y++)
                            {
                                    bloc[8*y+x] = imageIn[i0+y*width+x];
                            }
                    }

                    //dct conversion
                    dct::Data2DCT88(bloc);

                    //Division
                    for (int i = 0 ; i < 8*8 ; i++)
                    {
                            bloc[i] = round(bloc[i]/tabQuant[i]);

                    }
```

```
                        //insertion des valeurs dans les vecteurs temporaires DC et A
                        DC.push_back(bloc[0]);

                        for (int i = 1 ; i < 64 ; i++)
                        {
                                AC.push_back(bloc[i]);
                        }
                }
        }

        //initialisation de l'attribut coverWork avec les deux vecteurs cr   s
        coverWork.setAC(AC);
        coverWork.setDC(DC);

}


/**
        * computeCapacity
        * calcule la capacit  d'insertion sur l'image d'entr e,
        * le param tre d'insertion, la taille du message dans le support
        *
**/
void Embedding::computeCapacity() {

        //capacit  estim e sans matrixEmbedding

        int capacity = round(coverWork.getNbCoef() - coverWork.getNbDC() - coverWork

        //si la taille relative est sup rieure  2/3 cela ne sert  rien d'ins re

        if (message.size() >= (2./3)*capacity)
        {
                cout << endl << endl << "Attention : tout le message n'a pas pu  tre
                message.resize( round( (2./3)*capacity ) - ( (int)round( (2./3)*capa

        }


        //il faut chercher le nombre de changements  effectuer
    //i.e le nombre de fois qu'il faut diviser le message et le coverWork
    bool trouve = false;

    nbChangements = 1;


    //inutile de devoir calculer 2^n pour des valeurs trops grandes
    while ( ( message.size() / nbChangements ) > 28 )
    {
        nbChangements++;
    }
    // nbChangements * 2^( message.size() / nbChangements ) est la taille du coverWo
    // si c'est sup rieur au coverWork il faut alors encore diviser
    while (!trouve)
    {
       trouve = nbChangements * ( pow( 2, ceil( (float) message.size() / nbChangemen
```

```
        //on prend l'arrondi sup rieur de la division flottante parce que
        //par exemple si l'on veut d couper un message de 17 en 2 il faut des "morce
        //autrement si l'on prend 8 le message ne sera pas transmis completement (2*8=
        //on ins re alors 2*9=18 bits (bourrage)

        nbChangements++;

    }

    nbChangements--; // (on a incr m nt  pour rien avant de sortir)

    //recherche de la nouvelle taille du message s'il y a bourrage
    int newSize = message.size();

    while ( (newSize % nbChangements ) != 0 )
    {
         newSize++;
    }

    int bourrage = newSize - message.size();
/** DEBUG **/ /*
        if (newSize != message.size())
        {
                cout << "Bourrage : " << bourrage << endl ;
        }
/** DEBUG **/


     sizeMessage = message.size();

     //il faut decaler message de (newSize - message.size()) bits vers la droite pou
     for (int i = 0 ; i < bourrage ; i++)
     {
        //on ins re un z ro au debut
        message.insert(message.begin(), 0);
     }

     //calcul des param tres S et N de l'algo
     parameter_S = message.size() / nbChangements;

     parameter_N = (int) pow( 2, parameter_S ) - 1;

/**DEBUG**/  /*
        cout << endl << endl << "\tinfos sur les capacites :" << endl << endl;
        cout << "\t\tnbCoef : " <<       coverWork.getNbCoef() << endl;
        cout << "\t\tnbDC : " << coverWork.getNbDC() << endl;
        cout << "\t\tcoef a 0 : " << coverWork.getNbACEqualsTo(0) << endl;
        cout << "\t\tcoef a 1 : " << coverWork.getNbACEqualsTo(1) << endl;
        cout << "\t\tcapacity : " << capacity << endl ;
        cout << "\t\tparametre de l'algo : " << parameter_S << endl << endl;
/**DEBUG**/

                nb_usable_coef = coverWork.getNbAC() - coverWork.getNbACEqualsTo(0);
        NOM_ME = nbChangements;
                NOM_SK = 0; //(initialisation)
                embedding_efficiency = parameter_S / (1-pow(2, -parameter_S));
```

```
                sizeImage = coverWork.getNbCoef();
                nb_of_bits_embedded = message.size();    //apr s bourrage
                relative_length = (float)sizeMessage/sizeImage;
}



/**
        * shuffleCoverWork
        * m lange al atoirement le support    l'aide du mot de passe
        *
**/
void Embedding::shuffleCoverWork() {

        srand(stegoKey);

        coverWorkBijection = new int[coverWork.getNbAC()];
        coverWorkBijectionInverse = new int[coverWork.getNbAC()];
        coverWorkBijection = buildBijection(coverWork.getNbAC());

        vector<int> randomACCoef;
        randomACCoef.resize(coverWork.getNbAC());


        for (int i = 0 ; i < coverWork.getNbAC() ; i++)
        {
                coverWorkBijectionInverse[coverWorkBijection[i]] = i;
                randomACCoef[i] = coverWork.getIthAC(coverWorkBijection[i]);
        }

        coverWork.setAC(randomACCoef);


}


/**
        * shuffleMessage
        * m lange le message    l'aide du mot de passe
        *
**/
void Embedding::shuffleMessage() {

/**DEBUG**/ /*
        cout << "Message non m lang  : " << endl;
        for (int i = 0 ; i < message.size() ; i++)
        {
                cout << message[i] << " ";
        }
        cout << endl;
/**DEBUG**/

        srand(stegoKey);

        messageBijection = new int[message.size()];
        messageBijectionInverse = new int[message.size()];
```

```
        messageBijection = buildBijection(message.size());

        Message randomMessage;
        randomMessage.resize(message.size());

        for (int i = 0 ; i < message.size() ; i++)
        {
                messageBijectionInverse[messageBijection[i]] = i;
                randomMessage[i] = message[messageBijection[i]];
        }

        message.swap(randomMessage);

}


/**
        * hideSizes
        * cache le param tre de l'algo et la taille du message dans l'image
        * dans les premiers bits du support, en utilisant la technique LSB
        * (codage de chaque entier sur 32 bits)
        *
**/
void Embedding::hideSizes() {

        int NBBITS = 32;

        int* s_binary = new int[NBBITS];
        int* size_binary = new int[NBBITS];

        for (int i = 0 ; i < NBBITS ; i++)
        {
                s_binary[i] = 0;
                size_binary[i] = 0;
        }

        convertIntToBin(parameter_S, s_binary, 32);
        convertIntToBin(message.size(), size_binary, 32);
        for (int i = 0 ; i < NBBITS ; i++)
        {
                coverWork.setIthACLSB(i, s_binary[i]);
                coverWork.setIthACLSB(i+32, size_binary[i]);
        }

/**DEBUG**/       /*
        cout << "Parametres inseres :" << endl;
        cout << "\t Parametre s : " << parameter_S << endl;
        cout << "\t Taille : " << message.size() << endl << endl;
        cout << "\t Parametre n : " << parameter_N << endl;
/**DEBUG**/

}



/**
```

```
        * matrixEmbedding
        * ins re le message en utilisant la technique de matrixEmbedding
        *
        **/
void Embedding::matrixEmbedding() {

        //initialisation de la matrice H
        initH();

        int compteur = 64; //incr ment  sur les AC du coverWork
                                                //commencement    64 car on    ins
        ;

        for (int changement = 0 ; changement < nbChangements ; changement++)
        {
                //buffer de s    lments    du message
                int* buffMess = new int[parameter_S];

                for (int i = 0 ; i < parameter_S ; i++)
                 {
                        buffMess[i] = message[changement*parameter_S + i];
                 }


                 //buffer de n    lments    du coverWork
                 //on m morise les index des    lments
                 int* buffCoverWorkId = new int[parameter_N];

                 for (int i = 0 ; i < parameter_N ; i++)
                 {
                        buffCoverWorkId[i] = compteur;
                        compteur = coverWork.getIthNextNonZeroAC(compteur);

                 }

                 int* Hx = new int[parameter_S];
                 int* He = new int[parameter_S];
                 int e_B10;

                 EFFONDREMENT:

                 //calcul de Hx

                 for (int i = 0 ;  i < parameter_S ; i++)
                        Hx[i] = 0;

                 for (int i = 0 ; i < parameter_S ; i++)
                 {
                        for (int j = 0 ; j < parameter_N ; j++)
                                Hx[i] += ( lsb( coverWork.getIthAC( buffCoverWorkId

                        Hx[i] %= 2 ;
                 }

                 //calcul de m-Hx = He
```

```
                for (int i = 0 ; i < parameter_S ; i++)
                        He[i] = ( buffMess[i] + Hx[i] ) % 2 ;


                //calcul de e
                //e_B10 est la repr sentation en base 10 de e
                e_B10 = 0;

                for (int i = 0 ; i < parameter_S ; i++)
                {
                        e_B10 += He[i]*(int)pow(2, parameter_S-i-1);
                }

                e_B10--;

        if (e_B10 >=0 )
        {
                int newAC = (coverWork.getIthAC(buffCoverWorkId[e_B10]) > 0 ?
coverWork.getIthAC(buffCoverWorkId[e_B10])-1 : coverWork.getIthAC(buffCoverWorkId[e_
                        coverWork.setIthAC(buffCoverWorkId[e_B10], newAC )
;


                        //effondrement?
                        if (coverWork.getIthAC(buffCoverWorkId[e_B10]) == 0)
                        {
                                NOM_SK++;
                                for (int i = e_B10 ; i < parameter_N-1 ; i++)
                                {
                                        buffCoverWorkId[i] = buffCoverWorkId[i+1];
                                }

                                buffCoverWorkId[parameter_N-1] = compteur;
                                compteur = coverWork.getIthNextNonZeroAC(compteur);

                                goto EFFONDREMENT;
                        }
                }

        }

        NOM_all = NOM_ME + NOM_SK;
        relative_nb_of_modifications = (float)NOM_all / nb_usable_coef;

}

/**
        * unShuffleCoverWork
        * r arrange le coverWork dans l'ordre
        *
**/
void Embedding::unShuffleCoverWork() {

        vector<int> newCoverWork;
        newCoverWork.resize(coverWork.getNbAC());
```

```
        for (int i = 0 ; i < coverWork.getNbAC() ; i++)
        {
                newCoverWork[i] = coverWork.getIthAC(coverWorkBijectionInverse[i]);
        }

        coverWork.setAC(newCoverWork);
}


/**
        * saveStegoImage
        * sauvegarde l'image .jpegLike et l'image decompressee .pgm
        * \param stegoPath le chemin de l'image de sortie
        *
**/
void Embedding::saveImage(char* stegoPath) {

        int height = imageIn.NbRow();
        int width = imageIn.NbCol();

        //ECRITURE DU FICHIER .jpegLike
        string strStegoPath(stegoPath);

        strStegoPath.erase(strStegoPath.end()-4, strStegoPath.end());

        strStegoPath += ".jpegLike";

        ofstream fichier(strStegoPath.c_str(), ios::out | ios::trunc);
// ouverture en   criture   avec effacement du fichier ouvert


        if(fichier)
    {
                int cptDC = 0;
                int cptAC = 0;
                int i = 0;

                while (i < coverWork.getNbAC() + coverWork.getNbDC() )
                {
                        fichier << coverWork.getIthDC(cptDC) << endl;
                        cptDC++; i++;
                        for (int k = 0 ; k < 63 ; k++)
                        {
                                fichier << coverWork.getIthAC(cptAC) << endl;
                                cptAC++;
                                i++;
                        }
                }

        fichier.close();
    }
    else
         cerr << "Impossible d'ouvrir le fichier !" << endl;


    //ECRITURE DU FICHIER .PGM
```

```
        Image<octet> temp(height, width);


        int cptDC = 0;
        int cptAC = 0;
       for (int row = 0 ; row < height/8 ; row++)
            {
                    for (int col = 0 ; col < width/8 ; col++)
                    {

                            //premier   lement   du bloc
                            int i0 = row*8*width+col*8;

                            double* bloc = new double[8*8];


                            //on remet notre DC
                            bloc[0] = coverWork.getIthDC(cptDC)*tabQuant[0];
                            cptDC++;


                            //puis on met les AC
                            for (int j = 1 ; j < 8*8 ; j++)
                            {
                                    bloc[j] = coverWork.getIthAC(cptAC)*tabQuant[j];
                                    cptAC++;

                            }

                    dct::DCT882Data(bloc);


                            for (int x = 0; x < 8 ; x++)
                            {
                                    for (int y = 0 ; y < 8 ; y++)
                                    {
                        int coef =  (int)bloc[8*y+x];

                        temp[i0+y*width+x] = (coef > 255) ? 255 : ( (coef < 0) ? 0 : coe
                                    }
                            }
                    }
            }

            imageOut = temp;

            ImageIO<octet>::WritePGM(stegoPath, imageOut);

}



/**
        * saveHistogram
        * enregistre l'histogramme des coef AC
        * \para histoPath le chemin du fichier     etre enregistr
```

```
**/
void Embedding::saveHistogram(char* histoPath) {


        int borne = 6;

        vector<int> histogram;

        for (int i = 0 ; i < 2*borne ; i++)
        {
                histogram.push_back(0);
        }

        int milieu = borne;

        for (int i = 0 ; i < coverWork.getNbAC() ; i++)
        {
                if ((coverWork.getIthAC(i) > -borne) && (coverWork.getIthAC(i) < bor
                {
                        histogram[borne + coverWork.getIthAC(i)]++;
                }
        }

        ofstream file(histoPath, ios::out|ios::trunc);  // ouverture en   criture   av

        if(file)
    {

        for (int i = (-1)*borne ; i < borne ; i++)
                {
                        file << i << " " << histogram[borne + i] << endl;
                }

        file.close();
    }
    else
         cerr << "Impossible d'ouvrir le fichier !" << endl;


}

/**
        * printResults
        * Affiche des information sur l'insertion
**/
void Embedding::printResults() {


        cout << endl <<  "\tR sultats :" << endl << endl;
        cout << "\t\tTaille du message : " << sizeMessage << " bits" << endl;
        cout << "\t\tTaille de l'image : " << sizeImage << " bits" << endl;
        cout << "\t\tTaille relative : " << relative_length << endl;
        cout << "\t\t" << nb_of_bits_embedded << " bits ins r s " << endl;
        cout << "\t\tEmbedding efficiency : " << embedding_efficiency << endl;
        cout << "\t\t" << NOM_ME << " modifications dues au matrix embedding" << end
        cout << "\t\t" << NOM_SK << " modifications dues aux effondrements" << endl;
```

```
        cout << "\t\t" << NOM_all << " modifications au total " << endl;
        cout << "\t\t" << "Nombre de modifications relatives : " << relative_nb_of_r

}



/**
        * initH
        * initialise la matrice H utilis e dans l'algo
        *    partir du param tre n et s
        *
**/
void Embedding::initH() {

        H = new int*[parameter_N];                // matrice H de 2^s -1 lignes et s c

    for (int i = 0 ; i < parameter_N ; i++)       // contient tous les entiers de 1
    {
        H[i] = new int[parameter_S];
    }

    //calcul de la matrice H
    for (int i = 0 ; i < parameter_N ; i++)
    {
        convertIntToBin(i+1, H[i], parameter_S);
    }
}


/**
        * convertIntToBin
        * convertit un nombre en un nombre en base 2
        * \param number le nombre    convertir
        * \param binary param tre d'entr e/sortie
        * \param size le nombre de bits sur lesquel coder le nombre
        *
**/

void Embedding::convertIntToBin(int number, int* binary, int size) {

        //On stocke dans le vecteur des poids faibles vers les poids forts.
        for (int i=0; i<size; i++)
    {
                binary[i] = ((number>>i)&1);

                if ((binary[i] != 0) && (binary[i] != 1)) {
                  cout<<"convertIntToBin : Convertion en binaire n'est pas bonne\n";
                  exit(1);
                }
    }

        //on inverse le tableau
        for (int j = 0 ; j < size/2 ; j++)
        {
                int tmp = binary[j];
```

```
                binary[j] = binary[size-j-1];
                binary[size-j-1] = tmp;
        }
}

/**
        * convertCharToBin
        * convertit un caract re en un nombre en base 2
        * \param mess le caract re    convertir
        * \param binary param tre d'entr e/sortie
        *
**/
void Embedding::convertCharToBin(char mess, int* binary) {

        //On stocke dans le vecteur des poids faibles vers les poids forts.
        for (int i=0; i<8; i++)
    {
                binary[i] = ((mess>>i)&1);

                if ((binary[i] != 0) && (binary[i] != 1)) {
                  cout<<"convertIntToBin: Convertion en binaire n'est pas bonne\n";
                  exit(1);
                }
    }
        //on inverse le tableau
        for (int j = 0 ; j < 4 ; j++)
        {
                int tmp = binary[j];
                binary[j] = binary[7-j];
                binary[7-j] = tmp;
        }
}


/**
        * buildBijection
        * cr    une bijection de [1, n] dans lui m me
        * \param n borne max de l'ensemble
        *
**/
int* Embedding::buildBijection(int n) {

        int i, j, nbelem;

        int* f = new int[n];


        vector<unsigned long int> lst_integer(n);

        for (i = 0 ; i < n ; i++)
        {
                lst_integer[i] = i;
        }

        i = 0;
        for (nbelem = n ; nbelem > 0 ; nbelem--)
```

```
        {
                j = getRandomInt(nbelem-1);

                f[i] = lst_integer[j];

                lst_integer[j] = lst_integer[nbelem-1];
                lst_integer.pop_back();

                i++;
        }

        return f;
}


/**
        * getRandomInt
        * retourne un entier al atoire entre [0, bSup]
        * \param bSup la borne sup de l'intervalle
        *
**/
int Embedding::getRandomInt(int bSup) {

        if (bSup <= 0)
                return 0;
        else
        return (int) (int)rint(bSup*(rand()/(double)RAND_MAX));
}


/**
        * fonction de test des m thodes
        *
**/
void Embedding::test() {

        int n = 1024;

        int* binary = new int[32];

        int i = 0;
     int j = 32-1;

     while (i <= 32)
     {
          if ( (n >> i) & 1)
          {
               binary[j]=1;
          }
          else
          {
               binary[j]=0;
          }

          i++;
          j--;
```

```
    }


        //convertIntToBin(n, binary, 32);

        cout << "repr sentation de " << n << " en binaire : " ;
        for (int k = 0 ; k < 32 ; k++)
        {
                cout << binary[k] << " ";
        }
        cout << endl;
}


/**
        * lsb
        * retourne le lsb d'un entier
        * \param n l'entier
        *
**/
int Embedding::lsb(int n) {

        //positif + impair OU negatif + pair => 1
        //positif + pair OU negatif + impair => 0
        if (((n > 0) && (n%2 != 0)) || ((n < 0 ) && (n%2 == 0)))
    {
                return 1;
        }
        else
        {
                return 0;
        }
}
```

A.6 Extraction.h

```
     /**
    * Name : Extraction
    * author : R mi Watrigant
    * date : 30/04/09
    * Description : extrait un message ins r  dans une image en utilisant l'algorit
        * de Westfeld
**/

#ifndef EXTRACTION_H
#define EXTRACTION_H

#include "../image/image.H"
#include "../image/imageio.H"
#include "../image/imagetools.H"

#include "../image/DCTCoef.h"

#include <iostream>
#include <fstream>
#include <math.h>
#include <iostream>
         using namespace std;

typedef vector<int> Message;

class Extraction {

        private :

                /**
                        * param tres d'entr e de l'algo
                **/
                char* jpeglikePath;
                int qualityFactor;
                int stegoKey;

                /**
                        * le message et les coef de l'image dans l'algo
                **/
                Message message;
                DCTCoef lstCoef;

                /**
                        * les param tres techniques de l'algo
                **/
                int parameter_S;
                int sizeMessage;
                int parameter_N;
                int** H;

                /**
                        * stockage des bijections pour m langer le message et le co
                **/
                int* imageBijection;
```

```
int* imageBijectionInverse;
int* messageBijection;
int* messageBijectionInverse;

/**
        * readFileAndFillLstDCT
        * lit le fichier et remplit la liste des coef DCT
        * \param fichier flux istream
**/
void readFileAndFillLstDCT(ifstream * fichier);


/**
        * forEachBlocExtraction
        * calcule le syndrome d'un bloc, ie le message qu'il contien
        * \param supportRecu le morceau de support
**/
int* forEachBlocExtraction(int* supportRecu);


/**
        * initH
        * initialise la matrice H utilis e dans l'algo
        *    partir du param tre n et s
        *
**/
void initH();

/**
        * convertIntToBin
        * convertit un nombre en un nombre en base 2
        * \param number le nombre    convertir
        * \param binary param tre d'entr e/sortie
        * \param size le nombre de bits sur lesquel coder le nombre
        *
**/

void convertIntToBin(int number, int* binary, int size);

/**
        * buildBijection
        * cr    une bijection de [1, n] dans lui m me
        * \param n borne max de l'ensemble
        *
**/
int* buildBijection(int n);

/**
        * getRandomInt
        * retourne un entier al atoire entre [0, bSup]
        * \param bSup la borne sup de l'intervalle
        *
**/
int getRandomInt(int bSup);

/**
```

```
                * lsb
                * retourne le lsb d'un entier
                * \param n l'entier
                *
        **/
        int lsb(int n);

public :

        /**
                * Constructeur
                * lecture du fichier
                * \param IMAGE_PATH le chemin de l'image steganographiee
                * \param STEGO_KEY mot de passe necessaire pour d finir les
                *
        **/
        Extraction (char* IMAGE_PATH, int STEGO_KEY);

        /**
                * Destructeur
        **/
        ~Extraction();

        /**
                * shuffleImage
                * m lange les coef de l'image
        **/
        void shuffleImage();

        /**
                * shuffleMessage
                * creation de la bijection et de son inverse pour le message
        **/
        void shuffleMessage();

        /**
                * readParameters
                * lecture du param tre de l'algo et de la taille du message
        **/
        void readParameters();

        /**
                * initParameters
                * initialise les parametres de l'algo
        **/
        void initParameters();


        /**
                * extractMessage
                * extrait le message
        **/
        void extractMessage();

        /**
                * unShuffleMessage
```

```
                * remet le message dans le bon ordre
        **/
        void unShuffleMessage();

        /**
                * printMessage
                * affiche le message extrait
        **/
        void printMessage();

};


#endif
```

A.7 Extraction.cpp

```
     /**
    * Name : Extraction
    * author : R mi Watrigant
    * date : 30/04/09
    * Description : extrait un message ins r  dans une image en utilisant l'algorit
        * de Westfeld
**/


#include "Extraction.h"

#include <math.h>
#include <stdexcept>


/**
        * Constructeur
        * lecture du fichier
        * \param IMAGE_PATH le chemin de l'image steganographiee
        * \param STEGO_KEY mot de passe necessaire pour d finir les marches al ato
        *
**/
Extraction::Extraction (char* IMAGE_PATH, int STEGO_KEY) {


        this->stegoKey = STEGO_KEY;

        //lecture du fichier .jpegLike accompagnant le fichier .pgm
        string strJpeglikePath(IMAGE_PATH);
        strJpeglikePath.erase(strJpeglikePath.end()-4, strJpeglikePath.end());
        strJpeglikePath += ".jpegLike";

        ifstream fichier(strJpeglikePath.c_str(), ios::in);  // ouverture en lecture

        //si le fichier .jpeglike est pr sent on le lit,
        //sinon on affiche un message d'erreur et on quitte
        if (fichier)
        {
                readFileAndFillLstDCT(&fichier);
        }
        else
        {
                cerr << "Le fichier.jpegLike doit accompagner le fichier .pgm \n \tA
                exit(-1);
        }

}

/**
        * Destructeur
**/
Extraction::~Extraction() {
        delete imageBijection;
        delete imageBijectionInverse;
```

```
        delete messageBijection;
        delete messageBijectionInverse;

        for (int i = 0 ; i < parameter_N ; i++)
        {
                delete H[i];
        }
        delete H;
}

/**
        * readFileAndFillLstDCT
        * lit le fichier et remplit la liste des coef DCT ainsi que
        * les param tres de l'algo se trouvant au debut du fichier
        * \param fichier flux istream
**/
void Extraction::readFileAndFillLstDCT(ifstream * fichier) {

        int compteur = 0;

        vector<int> AC;
        vector<int> DC;

        string ligne;

        while(getline(*fichier, ligne))  // tant que l'on peut mettre la ligne dans
    {
        if ((compteur % 64) == 0)
        {
                DC.push_back(atoi(ligne.c_str()));
                compteur++;
        }
        else
        {
                AC.push_back(atoi(ligne.c_str()));
                        compteur++;
           }
        }

        fichier->close();

        lstCoef.setAC(AC);
        lstCoef.setDC(DC);
}

/**
        * shuffleMessage
        * m lange les coef de l'image
**/
void Extraction::shuffleImage() {

        srand(stegoKey);

        imageBijection = new int[lstCoef.getNbAC()];
        imageBijectionInverse = new int[lstCoef.getNbAC()];
        imageBijection = buildBijection(lstCoef.getNbAC());
```

```
        vector<int> randomACCoef;
        randomACCoef.resize(lstCoef.getNbAC());


        for (int i = 0 ; i < lstCoef.getNbAC() ; i++)
        {
                imageBijectionInverse[imageBijection[i]] = i;
                randomACCoef[i] = lstCoef.getIthAC(imageBijection[i]);
        }

        lstCoef.setAC(randomACCoef);

}


/**
        * shuffleMessage
        * creation de la bijection et de son inverse pour le message
**/
void Extraction::shuffleMessage() {

        //cr ation de la bijection du message et son inverse
        srand(stegoKey);

        messageBijection = new int[message.size()];
        messageBijectionInverse = new int[message.size()];
        messageBijection = buildBijection(message.size());

        for (int i = 0 ; i < message.size() ; i++)
        {
                messageBijectionInverse[messageBijection[i]] = i;
        }


}


/**
        * readParameters
        * lecture du param tre de l'algo et de la taille du message
**/
void Extraction::readParameters() {

        vector<int> s_binary;
        vector<int> size_binary;

        for (int i = 0 ; i < 32 ; i++)
        {
                s_binary.push_back( lsb( lstCoef.getIthAC(i) ) );
                size_binary.push_back( lsb( lstCoef.getIthAC(i+32) ) );
        }

        parameter_S = 0;
        sizeMessage = 0;
        for (int i = 0 ; i < 32 ; i++)
```

```
        {
                parameter_S += s_binary[i]*pow(2, 32-i-1);
                sizeMessage += size_binary[i]*pow(2, 32-i-1);
        }


        message.resize(sizeMessage);

/**DEBUG**/       /*
        cout << "Parametres inseres :" << endl;
        cout << "\t Parametre s : " << parameter_S << endl;
        cout << "\t Taille : " << message.size() << endl << endl;
/**DEBUG**/

}

/**
        * initParameters
        * initialise les parametres de l'algo
**/
void Extraction::initParameters() {

        parameter_N = (int) pow(2, parameter_S) - 1;

        initH();


}



/**
        * extractMessage
        * extrait le message
**/
void Extraction::extractMessage() {

        int cpt = 64;
        int cptMess = 0;

        int* buffCoef = new int[parameter_N];
        int* messageBloc = new int[parameter_S];
        while (cptMess < message.size())
        {
                for (int i = 0 ; i < parameter_N ; i++)
                {
                        buffCoef[i] = lsb( lstCoef.getIthAC(cpt) );
                        cpt = lstCoef.getIthNextNonZeroAC(cpt);
                }

                messageBloc = forEachBlocExtraction(buffCoef);

                for (int i = 0 ; i < parameter_S ; i++)
                {
                        message[cptMess] = messageBloc[i];
                        cptMess++;
```

```
                }
        }
}

/**
        * forEachBlocExtraction
        * calcule le syndrome d'un bloc, ie le message qu'il contient
        * \param supportRecu le morceau de support
**/
int* Extraction::forEachBlocExtraction(int* supportRecu)
{
    //calcul du syndrome
    int* syndrome = new int[parameter_S];
    for (int i = 0 ;  i < parameter_S ; i++) syndrome[i] = 0;

    for (int i = 0 ; i < parameter_S ; i++)
    {
        for (int j = 0 ; j < parameter_N ; j++)
        {
            syndrome[i] += (supportRecu[j]*H[j][i]) % 2 ;
        }
        syndrome[i] %= 2 ;
    }

    return syndrome;
}



/**
        * unShuffleMessage
        * remet le message dans le bon ordre
**/
void Extraction::unShuffleMessage() {



        vector<int> newMessage;
        newMessage.resize(message.size());

        for (int i = 0 ; i < message.size() ; i++)
        {
                newMessage[i] = message[messageBijectionInverse[i]];
        }

        for (int i = 0 ; i < message.size() ; i++)
        {
                message[i] = newMessage[i];
        }

/**DEBUG**/ /*
        cout << "Message non melange : " << endl;
        for (int i = 0 ; i < message.size() ; i++)
        {
                cout << message[i] << " " ;
        }
```

```
        cout << endl;
/**DEBUG**/

}

/**
        * printMessage
        * affiche le message extrait
**/
void Extraction::printMessage() {



        int antibourrage = message.size() - (message.size()/8)*8;

/** DEBUG **/ /*
        if (antibourrage > 0)
        {
                cout << "Bourrage : " << antibourrage << endl ;
        }
/** DEBUG **/

        char* messageText = new char[message.size() - antibourrage];

        cout << endl << "MESSAGE EXTRAIT " << endl << endl  ;
        for (int i = 0 ; i < message.size()/8; i++)
        {
                messageText[i] = 0;
                for (int j = 0 ; j < 8 ; j++)
                {
                        messageText[i] += message[antibourrage+i*8 + j]*pow(2, 7-j);

                }
                cout << messageText[i] ;
        }
        cout << endl << endl << endl;


}

/**
        * initH
        * initialise la matrice H utilis e dans l'algo
        *    partir du param tre n et s
        *
**/
void Extraction::initH() {

        H = new int*[parameter_N];                // matrice H de 2^s -1 lignes et s c
    for (int i = 0 ; i < parameter_N ; i++)       // contient tous les entiers de 1
    {
        H[i] = new int[parameter_S];
    }

    //calcul de la matrice H
    for (int i = 0 ; i < parameter_N ; i++)
```

```
    {
        convertIntToBin(i+1, H[i], parameter_S);
        //for (int j = 0 ; j < parameter_S ; j++) cout << H[i][j] << " " ; cout <<
    }
}


/**
        * convertIntToBin
        * convertit un nombre en un nombre en base 2
        * \param number le nombre    convertir
        * \param binary param tre d'entr e/sortie
        * \param size le nombre de bits sur lesquel coder le nombre
        *
**/

void Extraction::convertIntToBin(int number, int* binary, int size) {

        //On stocke dans le vecteur des poids faibles vers les poids forts.
        for (int i=0; i<size; i++)
    {
                binary[i] = ((number>>i)&1);

                if ((binary[i] != 0) && (binary[i] != 1)) {
                  cout<<"convertIntToBin: Convertion en binaire n'est pas bonne\n";
                  exit(1);
                }
    }

        //on inverse le tableau
        for (int j = 0 ; j < size/2 ; j++)
        {
                int tmp = binary[j];
                binary[j] = binary[size-j-1];
                binary[size-j-1] = tmp;
        }
}

/**
        * buildBijection
        * cr    une bijection de [1, n] dans lui m me
        * \param n borne max de l'ensemble
        *
**/
int* Extraction::buildBijection(int n) {

        int i, j, nbelem;

        int* f = new int[n];


        vector<unsigned long int> lst_integer(n);

        for (i = 0 ; i < n ; i++)
        {
                lst_integer[i] = i;
```

```
        }

        i = 0;
        for (nbelem = n ; nbelem > 0 ; nbelem--)
        {
                j = getRandomInt(nbelem-1);

                f[i] = lst_integer[j];

                lst_integer[j] = lst_integer[nbelem-1];
                lst_integer.pop_back();

                i++;
        }

        return f;
}


/**
        * getRandomInt
        * retourne un entier aleatoire entre [0, bSup]
        * \param bSup la borne sup de l'intervalle
        *
**/
int Extraction::getRandomInt(int bSup) {

        if (bSup <= 0)
                return 0;
        else
        return (int) (int)rint(bSup*(rand()/(double)RAND_MAX));
}


/**
        * lsb
        * retourne le lsb d'un entier
        * \param n l'entier
        *
**/
int Extraction::lsb(int n) {

        //positif + impair OU negatif + pair => 1
        //positif + pair OU negatif + impair => 0
        if (((n > 0) && (n%2 != 0)) || ((n < 0 ) && (n%2 == 0)))
    {
                return 1;
        }
        else
        {
                return 0;
        }
}
```

A.8 mainEmbed.cpp

```
    #include "Embedding.h"


#include <cstdlib>
#include <cstring>
#include <stdio.h>
#include <fstream>
#include <math.h>
#include <string>
#include <iostream>
    using namespace std;



/* PARAMETRES GLOBAUX DU PROGRAMME */
int     STEGO_KEY ;
int     QUALITY_FACTOR;
char*   STEGO_PATH;
char*   SUPPORT_PATH;
char*   MESSAGE_PATH;
char*   HISTO_PATH;

/* FONCTIONS PRIVEES NECESSAIRES */

void processOptionsInLine(int, char**);


/**
        * MAIN
        * Programme principal...
        *
**/
int main(int argc, char** argv)
{

        //traite les arguments du programme
                processOptionsInLine(argc, argv);

                //cout << "initialisation..." << endl;
                //construction d'une instance de l'algo
                Embedding embedding(SUPPORT_PATH, STEGO_PATH, MESSAGE_PATH, STEGO_KE

                //cout << "compression..." << endl;
                //compression jusqu'   quantification
                embedding.compression();

                //cout << "calcul capacite..." << endl;
                //calcul de la taille du param tre
                //calcul de la taille du message dans l'image steganographi
                embedding.computeCapacity();

                //cout << "m lange du cover work..." << endl;
                //m lange les bits du support
                embedding.shuffleCoverWork();
```

```
                //cout << "m lange du message..." << endl;
                //m lange les bits du message;
                embedding.shuffleMessage();

                //cout << "insertion de la taille..." << endl;
                //ins re la taille du param tre et du message dans les premiers pi
                embedding.hideSizes();

                //cout << "insertion du message..." << endl;
                //ins re le message avec la m thode de matrix embedding
                embedding.matrixEmbedding();

                //cout << "unshuffle coverwork..." << endl;
                //r arrange le coverWork
                embedding.unShuffleCoverWork();

                //cout << "decompresse et sauve l'image..." << endl;
                //sauve l'image steganographiee dans un .jpegLike
                //sauve l'image "d compress e" dans un .pgm
                embedding.saveImage(STEGO_PATH);

                if (argc == 7)
                {
                        embedding.saveHistogram(HISTO_PATH);
                }

                embedding.printResults();


                return 0;
}


/**
        * proccessOptionsInLine
        * traite les options d'ex cution
        * \param argc
        * \param argv
        *
**/
void processOptionsInLine(int argc, char** argv)
{
        if ((argc != 6) && (argc != 7))
        {
                cout << "\nUsage : " << argv[0] << " support.pgm stego.pgm message.t
                exit(-1);
        }

        STEGO_PATH = new char[200];
        SUPPORT_PATH = new char[200];
        MESSAGE_PATH = new char[200];


        //recopie des param tres dans les variables
        strcpy(SUPPORT_PATH, argv[1]);
```

```
        strcpy(STEGO_PATH, argv[2]);

        strcpy(MESSAGE_PATH, argv[3]);


        char* stegoKey = new char[200]; strcpy(stegoKey, argv[4]);
        STEGO_KEY = atoi(stegoKey);



        char* qualityFactor = new char[3]; strcpy(qualityFactor, argv[5]);
        QUALITY_FACTOR = atoi(qualityFactor);

        if (argc == 7)
        {
                HISTO_PATH = new char[200];
                strcpy(HISTO_PATH, argv[6]);
        }


}
```

A.9 mainExtract.cpp

```
    #include "Extraction.h"


#include <cstdlib>
#include <cstring>
#include <stdio.h>
#include <fstream>
#include <math.h>
#include <string>
#include <iostream>
    using namespace std;



/* PARAMETRES GLOBAUX DU PROGRAMME */
int             STEGO_KEY;
char*   IMAGE_PATH;

/* FONCTIONS PRIVEES NECESSAIRES */

void processOptionsInLine(int, char**);


/**
        * MAIN
        * Programme principal...
        *
**/
int main(int argc, char** argv)
{

        //traite les arguments du programme
                processOptionsInLine(argc, argv);

                //cout << "constructeur" << endl;
                //construction d'une instance de l'algo
                Extraction extraction(IMAGE_PATH, STEGO_KEY);

                //cout << "shuffleImage()" << endl;
                //m lange les coef de l'image
                extraction.shuffleImage();

                //cout << "readParameters()" << endl;
                //lecture du param tre de l'algo et de la taille du message
                extraction.readParameters();

                //cout << "shuffleMessage() " << endl;
                //creation de la bijection et de son inverse pour le message
                extraction.shuffleMessage();

                //cout << "initParameters()" << endl;
                //initialise les param tres de l'algo
                extraction.initParameters();
```

```
                //cout << "extractMessage()" << endl;
                //extrait le message
                extraction.extractMessage();

                //cout << "unshuffleMessage() " << endl;
                //remet le message dans le bon ordre
                extraction.unShuffleMessage();

                //cout << "printMessage() " << endl;
                extraction.printMessage();


                return 0;
}


/**
        * proccessOptionsInLine
        * traite les options d'ex cution
        * \param argc
        * \param argv
        *
**/
void processOptionsInLine(int argc, char** argv)
{
        if (argc != 3)
        {
                cout << "\nUsage : " << argv[0] << " image.pgm stegoKey\n";
                exit(-1);
        }

        IMAGE_PATH = new char[200];

        //recopie des param tres dans les variables
        strcpy(IMAGE_PATH, argv[1]);

        char* stegoKey = new char[200]; strcpy(stegoKey, argv[2]);
        STEGO_KEY = atoi(stegoKey);




}
```

A.10 attack.cpp

```
     /**
        * Remi Watrigant
        * 09/05/2009
        * attaque de l'algorithme F5
        * prend une image steganographiee en entree,
    * supprime les 4 premieres et dernieres lignes et colonnes,
        * lui applique un filtre moyenneur
        * la compresse
        * calcule son histogramme dct
        *
**/

#include "../image/image.H"
#include "../image/imageio.H"
#include "../image/imagetools.H"
#include "../image/dct.h"



#include <cstdlib>
#include <stdio.h>
#include <cstring>
#include <fstream>
#include <math.h>
#include <string>
#include <iostream>
    using namespace std;

char* IMAGE_PATH;
int QF;

/**
        * proccessOptionsInLine
        * traite les options d'execution
        * \param argc
        * \param argv
        *
**/
void processOptionsInLine(int argc, char** argv)
{
        if (argc != 3)
        {
                cout << "\nUsage : " << argv[0] << " imagePath.pgm qualityFactor \n"
                exit(-1);
        }

        IMAGE_PATH = new char[200];

        //recopie des parametres dans les variables
        strcpy(IMAGE_PATH, argv[1]);

        char* qualityFactor = new char[3]; strcpy(qualityFactor, argv[2]);
        QF = atoi(qualityFactor);
```

```
}



 /**
     * crop()
     * supprime les 4 premieres et dernieres lignes et colonnes d'une image
     *
     **/
Image<octet>* crop(Image<octet> *imageIn)
{
    int width = imageIn->NbCol();
    int height = imageIn->NbRow();

    Image<octet>* newImage = new Image<octet>(height-8, width-8);

    for (int i = 0 ; i < width-8 ; i++)
    {
        for (int j = 1 ; j < height-8 ; j++)
        {
            (*newImage)[j*(width-8) + i] = (int)(*imageIn)[(j+4)*width + (i+4)];
        }
    }

    /**TEST**/ /**
    ImageIO<octet>::WritePGM("images/cropped.pgm", *newImage);
    /**TEST**/


    return newImage;
}

/**
     * filtre()
     * effectue un filte moyenneur sur l'image en entr e
     *
     **/
Image<octet>* filtre(Image<octet>* imageIn)
{
    int width = imageIn->NbCol();
    int height = imageIn->NbRow();

    int tailleFiltre = 3;

    //transformation de l'image dans un tableau 2D
        int _imgIn[width][height];
        double _imgOut[width][height];


        for (int i = 0 ; i < width ; i++)
        {
                for (int j = 0 ; j < height ; j++)
                {
                        _imgIn[i][j] = (int)(*imageIn)[j*width + i];
                        _imgOut[i][j] = 0;
                }
```

```
        }

        double paramFiltre = 2;

        //filtrage
        for (int i = tailleFiltre/2 ; i < width-tailleFiltre/2 ; i++)
        {
                for (int j = tailleFiltre/2 ; j < height-tailleFiltre/2 ; j++)
                {

                        _imgOut[i][j] = (1/(4*paramFiltre))* (paramFiltre*_imgIn[i

                }
        }

        //transformation du tableau 2D en image de sortie
        Image<octet>* newImage = new Image<octet>(height, width);
        for (int i = 0 ; i < width ; i++)
        {
                for (int j = 0 ; j < height ; j++)
                {
                        (*newImage)[j*width+i] = (int)round(_imgOut[i][j]);
                }
        }

        /**TEST**/ /*
    ImageIO<octet>::WritePGM("images/test.pgm", *newImage);
    /**TEST**/

        return newImage;
}



/**
        * readJpegLikeAndFillHisto
        * lit le fichier jpegLike qui contient la liste des coef DCT de l'image stego
        * param histo l'histo    remplir
**/
void readJpegLikeAndFillHisto(map<int, int> histo[8][8])
{
        //on transforme le char* .pgm en un string .jpeglike
        string jpegLikePath(IMAGE_PATH);
        jpegLikePath.erase(jpegLikePath.end()-4, jpegLikePath.end());
        jpegLikePath += ".jpegLike";

        ifstream file(jpegLikePath.c_str(), ios::in);

        if (file)
        {

                char* car;
                int coef;
                int compteur = 0;
                while (!file.eof())
                {
```

```
                        file.getline(car, 256);
                        coef = atoi(car);
                        int numeroLigne = (compteur/8) % 8;
                        int numeroColonne = compteur % 8;

                        //cout << compteur << " => (" << numeroLigne << ", " << nume

                        histo[numeroLigne][numeroColonne][coef] = histo[numeroLigne]

                        compteur++;


                }
        }
        else
        {
                cerr << "Le fichier.jpegLike doit accompagner le fichier .pgm \n \tA
                exit(-1);
        }
}

/**
        * compressAndBuildEstimatedHisto
        * compresse l'image crop e filtr e et calcule les histogrammes individuels
        * \param imageIn l'image crop e filtr e
        * \param histo l'histo    remplir
**/
void compressAndBuildEstimatedHisto(Image<octet>* imageIn, map<int, int> histo[8][8])
{
        int height = imageIn->NbRow();
        int width = imageIn->NbCol();

        //construction de la table de quantification
        int tabRef[8*8] = {              16, 11, 10, 16,  24,  40,  51,
61,
                                            12, 12, 14, 19,  26,  58,
60,  55,
                                    14, 13, 16, 24,  40,  57,  69,  56,
                                            14, 17, 22, 29,  51,  87,
80,  62,
                                            18, 22, 37, 56,  68,  109, 103, 77,
                                            24, 35, 55, 64,  81,  104, 113, 92,
                                    49, 64, 78, 87,  103, 121, 120, 101,
                                    72, 92, 95, 98,  112, 100, 103, 99  };


        //param tre c(Q)
        double cQ;

        if (QF < 50)
        {
                cQ = 50000/(QF+1);
        }
        else
        {
            cQ = 200 - 2*QF;
```

```
        }

    double* tabQuant = new double[8*8];


        for (int i = 0 ; i < 8*8 ; i++)
        {
                tabQuant[i] = tabRef[i]*cQ/100 ;
        }

        for (int row = 0 ; row < height/8 ; row++)
        {

                for (int col = 0 ; col < width/8 ; col++)
                {
                        //premier   lement   du bloc
                        int i0 = row*8*width+col*8;

                        //remplissage des coef

                        double* bloc = new double[8*8];
                        for (int x = 0; x < 8 ; x++)
                        {
                                for (int y = 0 ; y < 8 ; y++)
                                {
                                        bloc[8*y+x] = (*imageIn)[i0+y*width+x];
                                }
                        }

                        //dct conversion
                        dct::Data2DCT88(bloc);



                        //Division
                        for (int x = 0; x < 8 ; x++)
                        {
                                for (int y = 0 ; y < 8 ; y++)
                                {
                                        bloc[8*y+x] = round(bloc[8*y+x]/tabQuant[8*y-
                                        histo[x][y][ round(bloc[8*y+x]) ] = histo[x]
                                }
                        }

                }
        }
}

/**
        * fillBeta_kl
        * rempli le tableau des beta
        * \param H l'histo de la stego image
        * \param hbarre l'histo de l'image estimee
**/
void fillBeta_kl(map<int, int> H[8][8], map<int, int> hbarre[8][8], double beta_kl[8
{
```

```
        for (int k = 0 ; k < 8 ; k++)
        {
                for (int l = 0 ; l < 8 ; l++)
                {
                        double hbarre_0 = hbarre[k][l][0];
                        double hbarre_1 = hbarre[k][l][1] + hbarre[k][l][-1];
                        double hbarre_2 = hbarre[k][l][2] + hbarre[k][l][-2];

                        double H_0 = H[k][l][0];
                        double H_1 = H[k][l][1] + H[k][l][-1];
                        double H_2 = H[k][l][2] + H[k][l][-2];

                        beta_kl[k][l] = ( hbarre_1 * ( H_0 - hbarre_0 ) + ( H_1 - hb
( hbarre_2 - hbarre_1 ) ) / ( ( hbarre_1 * hbarre_1 ) + ( hbarre_2 - hbarre_1 ) * (

                }
        }
}

/**
        * computeBeta
        * calcule beta a partir des beta_kl
        * \param beta_kl
**/
double computeBeta(double beta_kl[8][8])
{
        /*
        cout << "beta[0][1] = " << beta_kl[0][1] << endl;
        cout << "beta[1][0] = " << beta_kl[1][0] << endl;
        cout << "beta[1][1] = " << beta_kl[1][1] << endl;
        */
        return ((beta_kl[0][1] + beta_kl[1][0] + beta_kl[1][1] ) / 3 );
}


/**
        * computeLongueurMessage
        * calcule la longueur du message insere
        * \param beta probabilite de modification d'un coef AC != 0
        * \param estimatedHisto l'histogramme de l'image estimee
**/
int computeLongueurMessage(double beta, map<int, int> estimatedHisto[8][8] )
{
        int nb_usable_coef = 0;          //nombre de coef AC non nul
        int coefEgal1 = 0;
        int k = 0;                                      //parametre de l'algo - tail

        int longueur_message = 0;

        //CALCUL DE NB_USABLE_COEF et de CoefEgal1
        for (int k = 0 ; k < 8 ; k++)
    {
        for (int l = 0 ; l < 8 ; l++)
        {
                if ( k+l >= 1 )
```

```
                {
                        for (map<int, int>::iterator it = estimatedHisto[k][l].begin
                                {
                                        if ( ( (*it).first == 1) || ((*it).f
                                                coefEgal1 += (*it).second;

                                        if ((*it).first != 0)
                                        nb_usable_coef += (*it).second;
                                }
                        }
                }
        }


        //CALCUL DU PARAMETRE K
        k = round ( log( ( 1 / beta ) + 1 ) / log(2) );

        int capacity = round(nb_usable_coef - 0.51*coefEgal1);

        //CALCUL DE LA LONGUEUR DU MESSAGE
        longueur_message = (pow(2, k)/( pow(2, k) - 1 ) ) * k * beta * capacity;

        return longueur_message;
}

int main(int argc, char** argv)
{
                //traite les arguments du programme
                processOptionsInLine(argc, argv);


                //LECTURE DE L'IMAGE D'ENTREE
                Image<octet>* imageIn = new Image<octet>();
                ImageIO<octet>::ReadPGM(IMAGE_PATH, *imageIn);
                Image<octet>* cropped = crop(imageIn);
        Image<octet>* filtree = filtre(cropped);

        //les histogrammes individuels
        map<int, int> estimatedHisto[8][8];
        map<int, int> stegoHisto[8][8];


        double beta_kl[8][8];
        double beta;

        readJpegLikeAndFillHisto(stegoHisto);
        compressAndBuildEstimatedHisto(filtree, estimatedHisto);
        fillBeta_kl(stegoHisto, estimatedHisto, beta_kl);

                beta = computeBeta(beta_kl);

                cout << "beta = " << beta << endl;

                int longueurMessage = computeLongueurMessage(beta, estimatedHisto);

                cout << "longueur du message = " << longueurMessage << endl;
```

```
        return 0;
}
```

A.11 makefile

```
     # Indiquer quel compilateur est a utiliser
CPP=g++

# Specifier les options du compilateur
CFLAGS=
LDFLAGS=

#Nom de l'executable
EXEC1 = compress
EXEC2 = embed
EXEC3 = extract
EXEC4 = attack



# Liste de fichiers objets necessaires pour le programme final
#SRC = fichier.cc fichier1.cc fichier2.cc


OBJ1 = ./src/compression/compress.o ./src/image/dct.o
OBJ2 = ./src/F5/Embedding.o ./src/image/dct.o ./src/F5/mainEmbed.o ./src/image/DCTC
OBJ3 = ./src/F5/Extraction.o  ./src/F5/mainExtract.o ./src/image/DCTCoef.o
OBJ4 = ./src/F5/attack.o  ./src/image/dct.o

all: $(EXEC1) $(EXEC2) $(EXEC3) $(EXEC4)

# Creation des .o a partir des .cc
%.o : %.c
	$(CPP) $<

# Generation du fichier executable


$(EXEC1): $(OBJ1)
# Deuxieme possibilite ATTENTION : OBJ contenant plusieurs
# dependances, il faut utiliser $^ et non $<
	$(CPP) -o $@ $^

$(EXEC2): $(OBJ2)
# Deuxieme possibilite ATTENTION : OBJ contenant plusieurs
# dependances, il faut utiliser $^ et non $<
	$(CPP) -o $@ $^

$(EXEC3): $(OBJ3)
# Deuxieme possibilite ATTENTION : OBJ contenant plusieurs
# dependances, il faut utiliser $^ et non $<
	$(CPP) -o $@ $^

$(EXEC4): $(OBJ4)
# Deuxieme possibilite ATTENTION : OBJ contenant plusieurs
# dependances, il faut utiliser $^ et non $<
	$(CPP) -o $@ $^
```

```
.PHONY : clean mrproper

# Nettoyage
clean:
	rm -rf *.o
	rm -rf *.*~
	rm -rf *~

mrproper: clean
	rm -rf $(EXE1).exe
	rm -rf $(EXE2).exe
	rm -rf $(EXE3).exe
	rm -rf $(EXE4).exe
```