Nano projet : Stéganographie

Buts : Allocation dynamique, tableau multidimensionnel dynamique, entrées/sorties fichier, maitrise de l'environnement de travail

Durée : 2 séances + travail personnel

Ce mini projet clôture le premier semestre. Il sera réalisé en binôme sur deux séances. Il nécessite un peu de travail personnel hors séance.

Introduction

La *stéganographie* [1] consiste à cacher un message « secret » dans un contenu « support ». Le message secret doit être invisible aux observateurs du contenu support, et son existence même doit idéalement être *indétectable* par un adversaire qui chercherait à décider si le contenu contient ou pas un message caché (on parle alors de stéganalyse). Le message caché doit bien sûr pouvoir être retrouvé par ceux qui connaissent son existence et savent décoder cette information. Ce TD s'intéresse à une des premières techniques de stéganographie, dite *Least Significant Bit* (LSB, « bit de poids faibles »), qui permet de cacher un message secret quelconque (du texte, une autre image, un programme exécutable, un fichier mp3, etc.) dans une image numérique. On ne considérera que les images en niveau de gris sur 8 bits dans ce projet. Cette technique illustre la différence entre l'imperceptibilité par un humain, et la détectabilité du message caché par un test (voir section facultative). En toute rigueur, cette technique est donc à proscrire pour une utilisation réelle. Si vous deviez utiliser un algorithme relativement indétectable (sous certaines conditions que nous passons sous silence), préférez par exemple l'algorithme HUGO [2]. L'indétectabilité mène à l'imperceptibilité, mais la réciproque est donc fausse ! Il ressort par ailleurs de l'état de l'art actuel que la stéganographie fonctionne (des algorithmes indétectables existent), mais que la stéganalyse est vaine (on ne saurait déterminer à coup sûr si un contenu contient ou pas un message caché).

Exemple : la photo 1 (Lena.pgm) est l'image d'origine, le message à cacher est un texte et la photo 2 (Tatoue.pgm) est l'image dans laquelle le message est caché. Visuellement, il n'y a pas de différences entre les photos 1 et 2 et le message est donc bien invisible.

Photo 1 : Lena.pgm

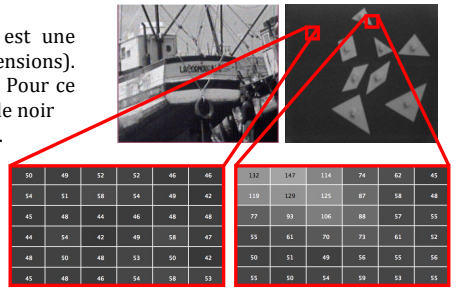
Phelma est une école merveilleuse
Phelma est une école merveilleuse
Phelma est une école merveilleuse
Phelma est une école merveilleuse
Phelma est une école merveilleuse
Phelma est une école merveilleuse
Phelma est une école merveilleuse

Message secret : ici, du texte

Photo 2 : Tatoue.pgm

Deux programmes complets seront réalisés, l'un capable d'encoder un message secret (du texte, une image, etc..) quelconque dans une image en niveau de gris, l'autre capable de décoder ce message.

Image numérique en niveau de gris



Une image numérique en niveau de gris est une fonction I(x,y) à support fini (ses dimensions). Chaque élément du tableau est appelé *pixel*. Pour ce TD, chaque pixel sera un scalaire sur 8 bits : le noir correspond à une valeur nulle, le blanc à 255.

Pour stocker une image en mémoire, on utilise un tableau d'octets à 2 dimensions. Vous avez ci-contre deux tableaux représentant les valeurs de l'image du tangram sur un voisinage de 6x6 pixels dans le fond et sur un sommet de triangle.

Les images seront lues et écrites sur disque au format de ficher PGM P5.

Principe de la technique de stéganographie LSB

L'œil est peu sensible aux variations minimes de teintes et d'intensité d'une image numérique. Ainsi, alors que toute image captée par un appareil photo comporte un aléa dû au bruit d'acquisition, ce bruit n'est pas perceptible si il est suffisamment faible. De même, si dans une image numérique on modifie arbitrairement la valeur d'un ou plusieurs pixels à une valeur « suffisamment proche », par exemple en passant d'une valeur `v`, $0 \leq v \leq 255$, à `v+1 ou v-1`, on obtient une nouvelle image qui est visuellement identique à la première. En termes plus techniques, on dit que les bits de poids faibles (*least significant bits*) des pixels contiennent peu ou pas d'information perceptible par l'homme.

La technique de stéganographie *Least Significant Bit* repose sur cette propriété de la perception. Pour cacher un message secret dans une image, on va modifier les pixels en modifiant les `n` bits de poids faible (`n`=1, 2, ou 4 bits par pixel pour simplifier).

Si on décide d'utiliser n=2 bits de poids faible par pixels pour cacher le message, alors pour cacher un octet qui comporte 8 bits, il faut le "disperser" sur 4 pixels de l'image ; et pour cacher un entier (4 octets, 32 bits) on utilise de même 16 pixels.

Exemple. Supposons qu'on veuille cacher dans une image l'octet de valeur 203, soit 0xCB en hexadécimal ou 11001011 en binaire, en utilisant 2 bits par pixel. Il faut modifier 4 pixels consécutifs dans l'image, de telle sorte que les bits de poids faible de ces pixels prennent les valeurs 11 00 10 11 de l'octet à cacher. Ils prennent donc respectivement les valeurs 11, 00, 10 et 11. Par exemple, si les 4 pixels de l'image ont les valeurs 202, 39, 186, 74 en décimal, on aura alors:

Octet à cacher 11001011, éclaté par paquet de 2 pixels	11	00	10	11
4 pixels de l'image initiale (binaire)	11001010	00100111	10111010	01001010
4 pixels de l'image initiale (décimal)	202	39	186	74
4 pixels de l'image modifiée (binaire)	11001011	00100100	10111010	01001011
4 pixels de l'image modifiée (décimal)	203	36	186	75

Pour décoder le message, il suffira d'extraire les deux bits de poids faible de chaque pixel puis de les regrouper dans l'ordre sur un octet.

Pour mettre en œuvre les processus d'encodage et de décodage LSB, il faut être capable de manipuler les bits d'un octet un à un. L'annexe technique explique comment s'y prendre.

Structure des informations cachées

Pour décoder un message caché dans une image, il est nécessaire que quelques informations soient cachées en plus du message lui même. Dans ce TD, pour cacher le contenu d'un fichier, on décide de cacher les informations suivantes :

- Le nom du fichier caché : c'est une chaine de 12 caractères, y compris la marque de fin de chaine qui est '\0'. Si on utilise 2 bits par pixels pour cacher le message, le nom du fichier contenant le message caché sera contenu dans les 12 caractères x 8 / 2 bits de poids faibles utilisés = 48 pixels utilisés pour cacher le nom du fichier message secret.
- Un entier T (4 octets, 32 bits à cacher) : la taille du message caché, c'est à dire le nombre d'octets du fichier caché. Avec 2 bits par pixels pour cacher le message, l'entier T est caché dans les 16 pixels suivants.
- Enfin, le contenu du fichier caché lui même, octet par octet (nombre d'octets spécifiés dans l'entier précédent). Avec 2 bits par pixels pour cacher le message, chaque octet du fichier secret est caché dans 4 pixels.

Pour cacher ces informations les unes après les autres dans les bits de poids faible des pixels de l'image, on commence par modifier le pixel en haut à gauche, puis on procède de gauche à droite et de haut en bas. Il faut donc que le nombre de pixels disponibles soit suffisant pour contenir tout le message caché.

Exemple d'encodage par stéganographie LSB

Pour encoder le fichier secret *message.txt*, qui contient la chaine de caractère ASCII « bonjour », dans une image initiale Lena.pgm, en utilisant n=2 bits de poids faible pour cacher l'information. On va donc cacher les trois informations suivantes, les unes après les autres :

- Le nom du fichier caché – ici, la chaine de caractères "message.txt" (12 octets à cacher).
- Un entier T (4 octets, 32 bits à cacher) : le nombre d'octets du message caché qui est aussi la taille du fichier message.txt – ici, T=7.
- Enfin, le contenu du fichier à cacher lui même, ici "bonjour" (7 octets à cacher).

Au total, il s'agit donc de cacher 12+4+7, soit 23 octets, dans l'image. Avec n=2 bits de poids faible par pixel, il faut altérer 23*4=92 pixels de l'image pour cacher toute l'information.

Considérons le processus permettant de cacher le nom du fichier « message.txt ». 0n détaille l'exemple pour les 4 premiers caractères soit `'m','e','s','s'`.

`'m'` vaut `109` en décimal, soit `0x6D` en hexadécimal et `01101101` en binaire. Il faut donc cacher `01` dans le premier pixel, `10` dans le deuxième pixel, `11` dans le troisième pixel, `01` dans le quatrième pixel,

`'e'` vaut `101` en décimal, soit `0x65` en hexadécimal et `01100101` en binaire. Il faut donc cacher `01` dans le cinquième pixel, `10` dans le sixième pixel, `01` dans le septième pixel, `01` dans le huitième pixel,

`'s'` vaut `115` en décimal, soit `0x73` en hexadécimal et `01110011` en binaire. Il faut donc cacher `01` dans le neuvième pixel, `11` dans le dixième pixel, `00` dans le onzième pixel, `11` dans le douzième pixel,

Pour le premier pixel qui vaut `162` (ou `0xA2`) initialement, donc `10100010` ; Si on change ses 2 bits de poids faibles en `01`, il prend donc la valeur `10100010` soit `161` ou `0xA1`.

Pour le douzième pixel qui vaut `159` (ou `0x9F`) initialement, donc `10011111` ; Si on change ses 2 bits de poids faibles en `11`, il prend donc la valeur `10011111` soit `159` ou `0x9F`.

Numéro de pixel	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
Valeur des pixels de l'image initiale Lena.pgm (décimal)	162	162	162	161	162	156	163	160	164	160	161	159	155	162	159	154
Valeur des pixels de l'image initiale Lena.pgm (hexadécimal)	A2	A2	A2	A1	A2	9C	A3	A0	A4	A0	A1	9F	9B	A2	9F	9A
Message à cacher (valeur voulue pour les 2 bits de poids faible)	01	10	11	01	01	10	01	01	01	11	00	11	01	10	11	00
Valeur des pixels de l'image tatouée Tatoue.pgm (héxadécimal)	A1	A2	A3	A1	A1	9E	A1	A1	A5	A3	A0	9F	99	A3	9C	98
Valeur des pixels de l'image tatouée Tatoue.pgm (décimal)	161	162	163	161	161	158	161	161	165	163	160	159	153	163	156	152

Le même procédé sera utilisé pour encoder la taille du message (32 bits, cachés dans les pixels 48 à 63 de l'image) puis le contenu du fichier « message.txt » lui même (7 octets, cachés dans les pixels 64 à 91 de l'image).

Algorithme

Pour cacher le message secret dans une image de dimension nl lignes et nc colonnes, il faut :

1. Définir le n, le nombre de bits par pixels
2. charger l'image. (voir annexe technique)
3. ouvrir le fichier à cacher contenant le message secret
4. calculer la taille du fichier à cacher (voir annexe technique).
5. Si la taille du fichier à cacher est compatible avec la taille de l'image
   - 5.1. Créer une nouvelle image de taille nlxnc pixels
   - 5.2. Recopier l'image source dans cette nouvelle image
   - 5.3. encoder le nom du fichier à cacher dans les 12x8/n premiers pixels de l'image
   - 5.4. encoder la taille du fichier à cacher dans les 4x8/n pixels suivants
   - 5.5. tant que le fichier à cacher n'est pas complètement lu
     - 5.5.1. lire en binaire un octet du fichier à cacher
     - 5.5.2. encoder cet octet dans 8/n pixels de l'image
   - 5.6. sauvegarder l'image dans un fichier

Exemple de décodage d'un message stéganographique LSB

Le décodage d'un message caché dans une image avec la technique LSB est simple si on connait le nombre de bits de poids faible utilisé ainsi que la structure des informations cachées. Dans l'exemple ci dessous, on utilise 2 bits par pixel.

Il suffit d'extraire de 4 pixels utilisés les 2 bits de poids faible, puis de les grouper dans un octet . On procèdera pas à pas en analysant les 4 premiers pixels, puis les 4 suivants etc…:

- Reconstitution du nom du fichier caché (réparti sur 12*4 pixels) et création en écriture de ce fichier.
- Reconstitution de l'entier T encodant la taille du contenu caché (réparti sur 16 pixels)
- Reconstitution octet par octet du contenu du fichier secret (réparti sur T*4 pixels) et écriture de ces octets au fur et à mesure dans le fichier résultat, ce qui permet de copier à l'identique sur disque le fichier qui avait été caché (même nom et contenu).

Travail à réaliser

Il vous est demandé de réaliser deux programmes.

- `encode` permet de cacher le contenu d'un fichier quelconque dans une image.
- `decode` permet d'extraire le message caché dans une image et recrée à l'identique le fichier qui avait été caché.

Méthode de travail

Vous réaliserez ce travail par groupe de deux. Il vous est demandé d'apporter une attention particulière à votre environnement de travail et à la propreté du code :

- Le code sera réparti dans plusieurs modules (fichiers .c et .h) [4].
- Vous réfléchirez au découpage en fonctions. Un découpage est proposé ci dessous ; vous pourrez le compléter en ajoutant des fonctions utilitaires à loisir.
- Le code sera proprement indenté [6] [5] et commenté
- Vous prendrez garde à compiler au fur et à mesure et vous testerez une à une vos fonctions au fur et à mesure de leur écriture (pas tout à la fin).

Découpage en modules et fonctions

On précise ci-dessous le prototype de chacune des fonctions à écrire. Chacune de ces fonctions est courte, typiquement 10 à 30 lignes de code.

Pour vous permettre de *compiler* votre programme, même si il est incomplet, vous commencerez par :

- Créer tous les fichiers source .c et .h
- Ecrire dans les .h les prototypes de toutes les fonctions
- Ecrire dans les .c des corps qui ne font rien pour chaque fonction

Ensuite, il faudra compléter et tester les fonctions une à une.

Rappel : vous pouvez ajouter d'autres fonctions utilitaires si cela facilite votre travail.

Fonctions de manipulation d'image

Les fonctions détaillées ci après, seront regroupées dans un module f_image.h / f_image.c,.

1) Fonction libérant une image en mémoire

```
/**
 * Libère la mémoire préalablement allouée pour une image de nbl*nbc pixels.
 * PARAMETRES :
 *    p_image : l'image à libérer
 */
void libere_image(unsigned char ** p_image) ;
```

2) Fonction allouant une image en mémoire

```
/**
 * Alloue une image de nbl*nbc pixels.
 * La zone de donnée de l'image, dans le champ img, est contiguë en mémoire.
 * La zone de donnée est initialisée à 0.
 * PARAMETRES :
 *    nbl : nombre de lignes de l'image à allouer
 *    nbc : nombre de colonnes de l'image à allouer
 * RETOUR : pointeur vers l'image allouée, ou NULL en cas d'erreur
 */
               unsigned char**  alloue_image(int nbl, int nbc) ;
```

Fonctions de décodage

Les fonctions détaillées ci après, seront regroupées dans un module f_decode.h / f_decode.c,.

3) Fonction qui extrait un octet caché dans une image à partir d'un pixel donné

```
/**
 * Extrait un octet caché dans l'image img, en démarrant au pixel « *p_k »
 * Cette fonction retourne l'octet du message qui est caché dans les
 * pixels d'indice *p_k , *p_k+1, *p_k+2, *p_k+3), etc.
 * selon le nombre de bits bitParPixel cachés dans un pixel.
 * *p_k est incrémenté pour référencer le prochain pixel à analyser
 * PARAMETRES :
 *    img : l'image dans laquelle l'octet est caché.
 *    p_k : pointeur vers l'indice du pixel a partir duquel on commence à extraire.
           *p_k est incrémenté par la fonction. Avec 2 bits par pixel
           *p_k sera augmenté de 4
 *    bitParPixel : nombre de bits de poids faible utilisé
          pour cacher le message dans chaque pixel
 * RETOUR : la valeur reconstruite.
 */
int extraitunoctet (unsigned char **img, int *p_k, char bitParPixel) ;
```

4) Fonction qui recrée un fichier caché dans une image

```
/**
 * Extrait le fichier caché dans l'image « img », à raison de « bitParPixel »
 * bits cachés dans un pixel, et recrée le fichier qui contenait le message.
 * Cette fonction commence par extraire de l'image le nom du fichier caché,
 * puis recrée pas à pas ce fichier à l'identique en extrayant son contenu
 * caché dans l'image.
 * PARAMETRES :
 *    img : l'image dans laquelle est caché le fichier.
 *    bitParPixel : nombre de bits de poids faible utilisé
          pour cacher le message dans chaque pixel
*     nl : nombre de lignes de l'image
*     nc : nombre de colonnes de l'image
 * RETOUR : 0 en l'absence d'erreur, -1 en cas d'erreur.
 */
     int imdecode (unsigned char** img, char bitParPixel, int nl, int nc) ;
```

Programme de decodage

Dans le fichier `decode.c`, écrire la fonction `main()` du programme `decode`..

Le programme `decode` s'utilise en ligne de commande dans le terminal de la façon suivante :

```
% decode <imageIn.pgm>  <nbbitsparpixel>
```

avec :

- `<imageIn.pgm>` : nom du fichier image tatouée qu'il s'agit de décoder.
- `<nbbitsparpixel>` : entier. Optionnel. Nombre de bits de poids faible utilisé par le message caché dans imageIn.pgm. Vaut 1, 2 ou 4.

La fiche [7] explique comment passer des paramètres à un programme depuis le terminal.

Fonctions de codage

Les fonctions détaillées ci après, seront regroupées dans un module f_decode.h / f_decode.c,

5) Fonction cachant un octet dans une image à partir d'un pixel donné

```
/
 * Cache l'octet « b » dans l'image img, en démarrant au pixel « k ».
 * Cette fonction cache l'octet « b » dans les pixels k,k+1,k+2,k+3), etc.
selon le nombre de bits
 * de poids faible bitParPixel cachés dans un pixel.
* PARAMETRES :
 *    img : l'image dans laquelle cacher l'octet « b ».
 *    b : l'octet à cacher, à raison de bitParPixel bits par pixel.
 *    k : indice du pixel ou on commence à cacher l'octet.
 * RETOUR : l'indice du prochain pixel pour cacher le prochain octet secret.
 *     Avec 2 bits par pixel, c'est k+4.
 */
            int cacheunoctet( unsigend char** img, unsigned char b,
                         int k, char bitParPixel) ;
```

6) *Fonction cachant un fichier dans une image*

```
/**
 * Cache le contenu du fichier de nom fileName dans l'image img,
 * à raison de bitParPixel bits de poids faible par pixels.
 * Cette fonction commence par cacher le nom  et la taille du fichier à cacher,
 * puis lit le contenu du fichier octet par octet et cache ce contenu
 * PARAMETRES :
 *    fic: nom du fichier contenant le message à cacher
 *    img : l'image dans laquelle cacher le fichier.
 *    b : l'octet à cacher, à raison de bitParPixel bits par pixel.
 *    bitParPixel : nombre de bits de poids faible utilisé
*     nl : nombre de lignes de l'image
*     nc : nombre de colonnes de l'image
 * RETOUR : 0 en l'absence d'erreur, un nombre non nul en cas d'erreur.
 */
int imencode(char *fic, unsigned char**img, char bitParPixel, int nl, int nc) ;
```

Programme de codage

Dans le fichier `encode.c`, écrire la fonction `main()` du programme `encode`.
Ce programme s'utilise en ligne de commande dans le Terminal de la façon suivante :

```
% encode <imageIn.pgm> <fileToHide> <imageOut.pgm> <nbbitsparpixel>
```

avec :

- `<imageIn.pgm>` : nom du fichier image à tatouer (le support)
- `<fileToHide>` : nom du fichier contenant les données à cacher (le message)
- `<imageOut.pgm>` : nom du fichier image tatoué, qui est créé par le programme
- `<nbbitsparpixel>` : entier. Nombre de bits de poids faible utilisé pour cacher le message. Il vaut 1, 2 ou 4.

Annexe technique

Binaire, décimal et hexadécimal.

Un même nombre entier peut être codé dans des bases différentes. Le binaire est la représentation interne utilisée par une machine numérique et correspond à la base 2, le décimal est notre système habituel et l'hexadécimal est la base 16, dont les éléments de base sont 0,1,2...9,A,B,C,D,E,F.
Notez que par convention, un nombre exprimé en hexadécimal est précédé de '0x'. Un octet est alors représenté par 2 digits (chiffres) représentant 2 groupes de 4 bits. Par exemple :

`255` s'écrit `1111 1111` en binaire et `0xFF` en hexadécimal.
`157` s'écrit `1001 1101` en binaire et `0x9D` en hexadécimal
`128` s'écrit `1000 0000` en binaire et `0x80` en hexadécimal.
`1` s'écrit `0000 0001` en binaire et `0x01` en hexadécimal.

Notez qu'une variable dans un programme est toujours représentée en binaire en machine et que seul l'affichage nous le présente dans une base quelconque. Ainsi, le programme suivant affiche :

```
c en base 10 :255 ;  c en base 16 :0xFF.
c en base 10 :157;  c en base 16 :0x9D.
c en base 10 :1;  c en base 16 :0x1.

int main() {  unsigned char c ;
  c= 255 ; printf("c en base 10 : %d ;  c en base 16 : %x \n",c,c) ;
  c= 157; printf("c en base 10 : %d ;  c en base 16 : %x \n",c,c) ;
  c= 1 ; printf("c en base 10 : %d ;  c en base 16 : %x \n",c,c) ;
}
```

Manipulation des images : parcours, lecture, sauvegarde

Les images seront allouées de manière contiguë, comme présenté en cours. Cela permet de pouvoir, dans cette application, parcourir l'image facilement soit avec un indice de ligne et un indice de colonne (matrice), soit avec un unique indice de pixel (linéairement). Cette dernière solution est plus simple pour ce projet.
Exemple : afficher tous les pixels d'une image avec 2 indices ligne et colonne

```
void affiche( unsigned char** im, int nl, int nc) { int i,j ;
  for (i=0 ; i<nl ; i++) {
    for (j=0 ; j<nc ; j++) printf("%d" ,im[i][j]) ;
   printf("\n ") ;
}
```

Exemple : afficher tous les pixels d'une image avec 1 indice pixel

```
void affiche( unsigned char** im, int nl, int nc) { int k;
   unsigned char* p ;
  p = im[0] ;
  for (k=0 ; k<nl*nc ; k++) {
    printf("%d" ,p[k]) ; // On peut aussi utiliser printf("%d" ,*im[k]) ;
    if (k%nc==nc-1) printf("\n ") ;
}
```

Les fonctions permettant de lire/sauvegarder une image dans un fichier de format pgm sont incluses dans la bilbiothèque SDL_Phelma.

```
unsigned char ** lectureimage8(char* fic, int *pnl, int *pnc) ;
```

Elle lit le fichier dont le nom est donné par `fic`, met a jour la taille de l'image `*pnl` et `*pnc`, crée une matrice dynamique de `*pnl` lignes et `*pnc` colonnes et retourne cette image. Elle retourne `NULL` si la lecture de l'image est impossible (fichier inexistant, mauvais format).

```
void ecritureimagepgm(char* fic, unsigned char** im, int nl, int nc) ;
```

Elle écrit l'image (matrice dynamique) `im` de `nl` lignes et `nc` colonnes dans le fichier dont le nom est donné par `fic`.

Taille d'un fichier.

Pour connaitre la taille d'un fichier en octet, on peut simplement ouvrir le fichier (fonction `fopen`), se positionner à la fin (fonction `fseek`), récupérer la position actuelle (fonction `ftell`).

Manipulation de bits en C

Les opérateurs bit à bit & (et), | (ou), ^ (ou exclusif), ~(inversion), >> (rotation à droite), <<(rotation à gauche) réalisent des opérations bit à bit et permettent d'isoler des bits ou groupes de bits.
Si i est un **octet**, prenons par exemple i=101(i.e. 0x65 en hexadécimal et 0110 0101 en binaire) :

- i & 0x2 (2 en binaire 00000010) permet d'isoler le bit 2 de i : i&0x2 donne dans cet exemple 0
- i & 0xF0 (soit 11110000 en binaire) permet d'isoler les 4 bits de poids forts de i. i & 0xF0 donne 01100000 en binaire ou 0x60 en hexadécimal ou 96 en décimal.
- i & 0x03 (soit 00000011 en binaire) permet d'isoler les 2 bits de poids faibles de i. i & 0x3 donne 00000001 en binaire ou 1 en hexadécimal ou 1 en décimal.
- i >> 3 décale tous les bits de i de 3 crans vers la droite. On obtient alors 00001100 ou 12 en décimal et 0xC en hexadécimal.
- i << 2 décale tous les bits de i de 2 crans vers la gauche, soit 10010100 ou 148 en décimal et 0x94 en hexadécimal.

Avec ces opérateurs, il est par exemple possible d'afficher un à un dans le Terminal les bits d'un octet. C'est ce que fait la fonction suivante, en commençant par les bits de poids fort :

```
void affichebin(unsigned char n) { char i;
      unsigned char bit = 0 ;
      unsigned char mask = 0x80 ; // 1000 0000 en binaire
      for ( i = 7 ; i >=0 ; i--) {
            bit = (n & mask) ;
            bit = bit >> i ;        // on decale le bit i crans a droite
            printf("%d", bit) ;
            mask >>= 1 ;
      }
}
```

Facultatif

Estimation du nombre plan de bit

Dans une image non tatouée, la différence entre les bits de poids faible de 2 pixels voisins suit en général une répartition statistique uniforme. En "modifiant" le bit de poids faible pour cacher un message, on rompt cette répartition statistique et on peut donc détecter que le bit de poids faible a été utilisé pour cacher un message. Cette stratégie peut s'étendre au 2ieme bit, puis au 3ieme, etc… Mieux, on peut estimer la taille du message caché. La méthode proposée par [3] peut être résumée de la manière suivante :

Soit u la valeur du pixel de coordonnées i,j et v celle du pixel voisin i, j+1.

On utilise 4 compteurs X, W, V et Z :

- Z est le nombre de couples u,v tels que u=v
- X est le nombre de couples u,v tels que v est pair et u < v ou v est impair est u>v
- Y est le nombre de couples u,v tels que v est pair et u > v ou v est impair est u<v
- W est le nombre de couples u,v de Y tels que |u-v|=1
- V = Y-W.
- P est le nombre total de couples et P = X+W+V+Z.

Si un message est caché, sa longueur relative (proportion de bits modifiée) est donnée par la plus petite racine de l'équation du second degré : $(W+Z)/2\ x^2 + (2*X-P)\ x - X = 0$.

Pour le bit de poids faible, la longueur du message est donc racine*nbre_pixel_image/8 octets.

Si le discriminant est négatif ou nul, c'est -(2*X-P)/(W+X) qui est la racine estimée.

Pour connaître le nombre de bits qui ont été utilisés, on applique la même procédure pour chaque plan de bits (il suffit de décaler la valeur des pixels u et v vers la droite de p bits et d'appliquer la même résolution).

On dispose alors de 8 valeurs (1 pour chaque plan de bits) de racine obtenues par cette méthode. Les valeurs des racines sont proches de 0 quand il n'y a pas de message caché, et sensiblement identiques et non nulles sinon pour les bits 0 à 3 (valeurs de 0,01 à 0,04). Pour détecter automatiquement le nombre p de bits utilisés, on peut choisir la valeur de p pour laquelle la différence entre 2 racines est la plus forte, ou mieux encore choisir p tel que :

$$\widehat{p} = \underset{p=0..7}{\arg\max}\left(r(p) - \frac{1}{p} * \sum_{k=0}^{p-1} r(k) \right) \quad avec \quad r(k) : racine \ \ estimée$$

Références

[1] Francois Cayre : Cryptographie, stéganographie et tatouage : des secrets partagés ,Interstices, 2008, http://interstices.info/jcms/c_32093/cryptographiestéganographie-et-tatouage-des-secrets-partages

[2] Tomas Pevny, Tomas Filler and Patrick Bas, Using High-Dimensional Image Models to Perform Highly Undetectable Steganography. Proc. Information Hiding Workshop, 2010, Calgary, Canada, pp. 161–177, https://hal.archives-ouvertes.fr/hal-00541353/document

[3] Sorina Dumitrescu and Xiaolin Wu and Nasir Memon : On steganalysis of random lsb embedding in continuous-tone images, IEEE International Conference on Image Processing, 2002

[4] Fiche informatique Phelma : *Notion de module en C et fichier header .h*. http://tdinfo.phelma.grenoble-inp.fr/2Aproj/fiches/modules_fichierHeader.pdf

[5] Fiche informatique Phelma : *Coding Style - conventions de codage*. http://tdinfo.phelma.grenoble-inp.fr/2Aproj/fiches/coding_styles.pdf

[6] Linus Torvalds. *Linux Kernel Coding Style*. http://computing.llnl.gov/linux/slurm/coding_style.pdf

[7] Fiche informatique Phelma : *Le prototype de la fonction main()*. http://tdinfo.phelma.grenoble-inp.fr/2Aproj/fiches/prototype_main.pdf