ALGORITHMES DE COMPRESSION

"L'algorithme JPEG en C++ Objet"

Antoine Chantalou
D.E.S.S Logiciels fondamentaux
Université Jussieu PARIS 7, France
E-mail : `antoine.chantalou@noos.fr`
Home-Page : `http ://mapage.noos.fr/antoine.chantalou/`

Grégory Cellier
D.E.S.S Logiciels fondamentaux
Université Jussieu PARIS 7, France
E-mail : `gregory.cellier@numericable.fr`

Ce document a été écrit dans le cadre du module "ALGORITHMIQUE POUR LA TRANSMISSION D'INFORMATIONS NUMÉRIQUES ET LE GRAPHIQUE" dispensé par Christian CHOFFRUT à l'Université Paris7. Nous présentons ici notre implémentation de l'algorithme de compression non conservative JPEG.
Nous présentons les algorithmes utilisés, leurs implementations, les divers programmes réalisées, ainsi qu'une batterie de tests.

Powered by LaTeX

Table des matières

1 Introduction

Dans le monde de l'Internet, ou plus généralement dans les réseaux, on a fréquemment recours à la compression des données, afin d'accroître les performances de transmission. On peut notamment distinguer deux modes de compressions : *conservative* et *non conservative*. La compression conservative regroupe les techniques qui permettent de générer une copie exacte des données, après un cycle de compression/décompression. En revanche, la compression non-conservative autorise une certaine perte de précision, en échange d'une compression considérablement accrue. Aussi avons-nous choisi d'implementer un algorithme de compression non-conservative, travaillant sur des images, qui permet d'obtenir des taux de compression inégalables : l'algorithme JPEG. Notre implémentation se base sur une image source BMP 24 bits, et permet de compresser des images d'une taille quelconque.

L'acronyme *JPEG* désigne le nom du groupe d'experts internationaux (**J**oint **P**hotographic **E**xpert **G**roup) qui a établi, en 1991, une méthode de compression d'images non conservative. L'algorithme repose intrinsèquement sur la *DCT* ou *Discret Cosin Transform*, initiée en 1974 par le professeur *Rao* de l'université du Texas.

Dans un premier temps, nous présentons les divers programmes que nous avons écrits. Ensuite, nous détaillons nôtre implémentation de la compression JPEG. Dans un troisième temps, nous expliquerons comment nous avons implanté la démarche inverse : la décompression. Et enfin, nous effectuons une batterie de tests, afin de pouvoir faire ressortir quelques caractéristiques des l'algorithmes implémentés.

2 Les programmes

Pour l'implémentation des algorithmes, nous avons choisi le langage C^{++}. Ce choix nous ait apparu judicieux, dans la mesure ou les algorithmes de compression et décompression JPEG requièrent un language de programmation puissant et vorace ; les langages interprétés, JAVA[1] et autre machines virtuelles sont donc à proscrire pour ce type d'executables. Il aurait été possible d'utiliser uniquement le language C, mais nous avons jugé opportin d'utiliser un language Objet, qui permet en général d'obtenir un code plus clair et plus structuré, grâce à la notion d'objets. D'autre part, la spécification en classes utilisant des méthodes publiques ou privées, permet de simplifier la répartition des taches, au sein du projet.

L'interface de "visionnage" (ou *viewer*) à été conçue en TCL/TK, pour ses qualités de portabilité, qui combiné à un code C^{++}, permet de proposer un programme de compression/décompression capable de fonctionner à la foi sous MacOS, Windows, et Unix (voir figure 1 page 4).

[1] Les languages interprétés à garbage collector sont en moyène de 10 à 15% moins rapide qu'un programme C...

FIG. 1 – **L'interface : divers skins**

Nous avons réalisé en tout quatre programmes, dont une interface TCL/TK, qui permet de manipuler des images BMP (format standard), et JPEG (notre format propriétaire). En outre, notre interface permet de compresser ou décompresser des images, et de les afficher à l'écran. Le programme TK fait appel à trois commande UNIX distinctes :

- La commande `bmp2jpg`, permet de compresser une image BMP vers notre format JPEG,
- La commande `jpg2bmp`, permet de convertir une image JPEG en une image BMP standard,
- La commande `draw`, permet d'afficher à l'écran les images dont le format est décrit précédemment.

Il est donc possible d'utiliser ces commande soit par le biais de l'interface graphique, soit directement via un *shell* UNIX quelconque.

2.1 La commande bmp2jpg

La commande peut s'utiliser comme suit :

```
bmp2jpg toto.bmp toto.jpg 90
```

Ou le première argument désigne le fichier BMP source, le second argument, le fichier destination JPG[2], et le quatrième, le facteur de compression (ici 90%). Le fait d'avoir considéré une programmation Objet permet d'obtenir un code relativement simple. Pour compresser une image BMP à l'aide de nôtre algorithme, il suffit d'utiliser le code suivant :

```
ImgBmp  *bmp = new ImgBmp(srcBmp);             // load le fichier .bmp en memoire
ImgJpeg *jpg = bmp->bmpToJpeg(factor,dstJPEG); // facteur qualité, destination.jpg
```

Dans un premier temps, on teste la validité des argument de la commande, puis on créer un object $ImgBmp$, grâce au constructeur $ImgBmp(srcBmp)$. Une foi que le constructeur a chargé l'image BMP en mémoire, nous pouvons appeler la méthode $bmpToJpeg()$ de la classe $ImgBmp$, qui effectue la compression, et

[2]Ce paramètre peut être omis, et le fichier dest prend alors le même nom que le fichier src, avec l'extension "JPG".

produit en sortie un objet *ImgJpeg*, ainsi qu'un fichier binaire JPEG. L'image peut alors être afficher grâce à la commande *draw*, et convertie dans un format BMP standard grâce à la commande *jpg2bmp*.

2.2 La commande jpg2bmp

Cette commande permet d'obtenir un fichier BMP à partir d'un de nos fichier JPEG. Il s'agit en faite d'une décompression, et par conséquent, on produit un fichier de taille supérieur au fichier compressé.

```
jpg2bmp toto.jpg toto.bmp
```

Le code principale de la commande est présenté ci-dessous :

```
ImgJpeg *jpg;               // la jpg source
ImgBmp  *bmp;               // la bmp générée
jpg = new ImgJpeg(argv[1]); // load le jpeg
bmp = jpg->jpegToBmp();     // convertie en bmp
bmp->saveFile(dst);         // sauvegarde le bmp dans un fichier standard
```

Tout d'abord, on charge le fichier binaire JPEG en mémoire(dans une instance de la classe *ImgJpeg*), puis on applique la méthode *jpegToBmp*() à cet objet, ce qui produit un objet de la classe *ImgBmp*. Enfin, on appel la méthode $saveFile(char*dst)$ pour créer un fichier BMP. Il peut être intéressant d'appeler successivement les les commandes *bmpToJpeg* et *jpegToBmp* sur une image BMP, car cela permet d'apprécier que l'image originale n'a pas perdu "un" octet dans las manipulation ; ce qui laisse envisager que les algorithme implementers sont justes.

2.3 La commande draw

La commande *draw* s'appuie directement sur les classes *window*, *ImgBmp* et *ImgJpeg*. Nous avons utilisé la *Xlib*, et uniquement la fonction $put\_pixel(x, y)$. Nous n'avon pas jugé utile d'utiliser des bibliothèques plus évolués dans la mesure ou le fichier JPEG que nous produisons n'est pas un fichier standard du type *JFIF*. Ci-dessous figure l'essentiel du code de la commande *draw*.

```
window *win; // a window
ImgBmp  *bmp;
ImgBmp *bmp2;
ImgJpeg *jpg;
switch(extension) {
  case BMP:
  bmp = new ImgBmp(argv[1]);
  win = new window(argv[1],bmp->widthTrue,bmp->heightTrue);
  win->draw(bmp);
  break;

  case JPG:
  jpg  = new ImgJpeg(argv[1]);
  bmp2 = jpg->jpegToBmp();
  win  = new window(argv[1],jpg->widthTrue,jpg->heightTrue);
  win->draw(bmp2);
  break;
}
```

Naturellement, la classe *window* que nous avons écrite ne peut afficher que des images BMP. Il s'agit en effet d'un code trivial, puisqu'il suffit de parcourir les pixels de la matrice BMP[3], un à un, et de les afficher à l'écran grâce à la fonction $put\_pixel(int, int)$ de la Xlib. Ce code est effectué par la méthode $draw(ImgBmp*)$ de la classe *window*.

Il n'est évidement pas possible d'afficher une image JPEG à partir du fichier binaire, sans effecter l'algorithme de décompression. C'est pourquoi, nous créons une instance de la classe $ImgBmp$, et que nous appelons la méthode $jpegToBmp()$, avant de pouvoir afficher l'image. Pour résumer, afficher à l'écran une image JPEG équivaut à décompresser une image JPEG vers un format BMP.

3 Compression

Afin de produire une image JPEG, il faut partir d'un format non compressé, c'est pour quoi nous avons choisi de travailler avec le format BMP (**B**it**M**a**P**).

3.1 Chargement des fichiers BMP

3.1.1 Le format BMP

Introduction
Le format BMP est un format d'image créé par Microsoft pour ses interfaces graphiques Windows. Il s'agit d'un format volumineux, étant donné qu'il s'agit d'un format non-compressé. Un fichier BMP se compose de quatre segments. Le premier, "Header" (ou BitmapFileHeader), est constitué d'informations sur le fichier lui-même. Le second, "Info Header" (ou BitmapInfoHeader) est constitué d'informations propres à l'image, des informations qui permettront d'afficher correctement l'image. Le troisième segment est la table des couleurs, qui permettra de reconstituer la palette utilisée par l'image. Ce segment est parfois incorporé dans l'Info Header, bien qu'il n'en fasse pas réellement partie. Enfin, on trouvera l'image même dans le dernier bloc, "Raster Data". Ces données permettront de reconstituer l'image à partir de la palette de couleur décrite dans le troisième segment.

Structure du fichier

	Segment Header. 14 octets.
00-01	Signature. Ce sont toujours les deux valeurs $42et4$D correspondant aux caractères ASCII BP.
02-05	Taille du fichier. C'est la taille du fichier en octets.
06-09	Réservés. Ces octets sont conventionnellement égaux à 00.
0A-0E	Data Offset ou adresse du segment de données. Ici est écrite l'adresse du segment Raster Data. Cela permettra de se positionner directement à cette adresse si l'on ne désire pas lire toutes les informations.

[3]En étant conscient du fait que les lignes de la matrice BMP sont inversées...

	Segment Info Header. 40 octets.
0E-11	Taille du segment. C'est la taille de Info Header, toujours égale à 40d (28).
12-15	Largeur. Taille horizontale de l'image (nombre de pixels).
16-09	Hauteur. Taille verticale de l'image (nombre de pixels).
1A-1B	Plans. Nombre de plans dans l'image. Toujours égal à 1 pour le BMP.
1C-1D	BitCount. Représente le nombre de bits par pixel. On en déduit le nombre de couleurs de la palette utilisée. 18 si vingt-quatre bits représentent un pixel (16 millions de couleurs, RGB).
1E-21	Compression. Type de compression utilisée. Egal à : 00 si pas de compression.
22-25	Taille de l'image en pixels. Peut-être (et est le plus souvent) égal à 00 dans le cas de fichiers non compressés.
26-29	Xpixels/m. Résolution horizontale en pixels par mètre.
2A-2D	Ypixels/m. Résolution Verticale en pixels par mètre.
2E-31	Nombre de couleurs. Nombre de couleurs réellement utilisées par l'image.
32-35	Couleurs importantes. Nombre de couleur importantes. Ces couleurs sont les premières de la palette, par ordre d'importance.

	Table des couleurs. Nombre de couleurs fois 4 octets.
XX Rouge :	Intensité de rouge dans la couleur décrite (RedValue).
XX+1 Vert :	Intensité de vert dans la couleur décrite (GreenValue).
XX+2 Bleu :	Intensité de bleu dans la couleur décrite (BlueValue).
XX+3 Réservé :	Inutilisé et égal à 0. Répété Nombre de Couleurs fois.

Segment Raster Data (Segment de données).
De l'adresse indiquée par DataOffset à la fin du fichier.

Commentaires sur la structure du fichier.

Header : Les deux premiers octets servent à assurer que l'on est bien en présence d'un fichier bitmap BMP. Le dernier champ est essentiel, car il permet de repérer l'adresse du segment de données, et d'y accéder directement.

InfoHeader : la compression : on ne rencontre pratiquement jamais de fichiers BMP compressés. Certains logiciels permettent d'en écrire, mais aucun ne les utilise. Généralement, les octets Xpixels/m et Ypixels/m sont inutilisés.

La palette de couleurs : ce segment est absent dans les images que nous considérons (BMP de codage 24 bit).

Raster Data (données) Ol est plus efficace de coder la couleur directement selon ses composantes RGB. Pour BitCount égal à 16, on a $2^16 = 65536$ couleurs utilisables par l'image. Pour BitCount=24 (le maximum), on a plus de 16 millions de couleurs.

ATTENTION : la largeur X doit être paire. Si la résolution est par exemple de 179 avec un mode 8 bits, alors chaque ligne aura une taille de 180 octets, avec un octet vide à chaque fin de ligne pour combler le vide. Cela permet en

fait d'accéder au fichier mot par mot, et non plus octet par octet.

L'image est stockée ligne par ligne dans le segment de données. Chaque ligne commence à gauche et se termine à droite, mais c'est la dernière ligne qui code la première, de sorte que si on lit les données à la suite, on obtient l'image renversée. D'autre part, on peut dire que l'image BMP est en format BGR et non RGB, en effet les couleurs de chaque pixels sont écrites de droite à gauche et non de gauche à droite.

3.1.2 Implémentation

La première étape à consiste au chargement de l'image BMP, grâce à la méthode $loadFile(fileName)$ de la classe $ImgBmp$. Dans un premier temps nous nous occupons du chargement de l'entête du fichier, puis de l'entête Bitmap, dans les variables d'instance correspondantes. Ensuite, nous chargeons les composantes RGB de chaque pixel dans une matrice de pixel[4]. L'image BMP est alors chargée dans un objet $Matrix$ qui comporte une matrice de pointeurs sur des objets de la classe $Pixel$.

Or, pour le chargement de l'image, il est nécessaire de considérer une matrice dont les largeurs et hauteurs sont des multiples de 16 pixels (pour le mode YUV et la DCT).

Le problème suivant se pose alors : Comment completer la matrice si l'image n'est pas decomposable en blocs de $16 * 16$?

Nous avons décidé d'opter pour la solution suivante :
– Les lignes sont complétées à droite par le dernier pixel de cette ligne
– On recopie ensuite en bas la dernière ligne de pixel rencontrée

Exemple : considérons que chaque numéro correspond à un pixel,voici ce que l'on obtient (dans cet exemple, la matrice carrée est de dimension 8).

```
1 2 3 4 5 6 . .     1 2 3 4 5 6 6 6
1 2 3 4 5 6 . .     1 2 3 4 5 6 6 6
1 2 3 4 5 6 . .     1 2 3 4 5 6 6 6
1 2 3 4 5 6 . .     1 2 3 4 5 6 6 6
1 2 3 4 5 6 . .     1 2 3 4 5 6 6 6
4 4 4 4 4 5 . .     4 4 4 4 4 5 5 5
. . . . . . . .     4 4 4 4 4 5 5 5
. . . . . . . .     4 4 4 4 4 5 5 5
```

3.2 Les modes RGB et YUV

3.2.1 Le codage Luminance, Chrominance

D'une manière générale, notre oeil est plus sensible à la luminance (i.e. l'intensité de la lumière) qu'à la chrominance (i.e. les couleurs). La première technique pour réduire la taille d'une image est donc de donner plus d'importance aux informations de luminance que de chrominance qui seront codées sur un espace

[4]cf. les classes Pixel et Matrix dans l'annexe

moins important. D'ailleurs, c'est cette technique est également utilisée dans les systèmes de diffusion de télévision Hertzienne.

Ansi, à partir d'un codage de pixel RGB, nous devons obtenir obtient trois variables $Y(luminance)$, $Cr(chrominance)$ et $Cb(chrominance)$, et ce grâce aux formules suivantes[5] :

- $Y = 0.299 * R + 0.587 * G + 0.114 * B)$
- $Cb = -0.1687 * R - 0.3313 * G + 0.5 * B + 128)$
- $Cr = 0.5 * R - 0.4187 * G - 0.0813 * B + 128)$

Pour Y la valeur maximale que l'on peut avoir est 255. Pour Cb et Cr on peut obtenir -127.5 , qui ne correspond pas a une valeur correcte, pour cela on ajoute 128 afin de rester dans l'interval [0,255].

On peut noter qu'il est possible de retrouver les valeurs RGB grâce aux les formules suivantes [6] :
- $R = Y + 1.402 * Cr - 128$
- $G = Y - 0.34414 * (Cb - 128) - 0.71414 * (Cr - 128)$
- $B = Y + 1.772 * (Cb - 128)$

Ansi, si l'on considère une image 640x480 RGB, de 24 bits par pixel, alors la matrice Y aura nécessairement même taille. En revanche, les matrices de Cr et de Cb peuvent être réduites à des matrices 320x240, en prenant les moyennes des valeurs des pixels regroupés par carré de quatre. Cela ne nuit pas à la précision des informations sur l'image car l'oeil est moins sensible aux écarts de couleurs qu'aux différences d'intensités lumineuses. Comme chaque point de chaque matrice est une information codée sur 8 bits, il y a chaque fois 256 niveaux possibles (0-255).

3.2.2 Implémentation

Nous avons vu que l'on peut réduire la taille d'une image de 1/3 en appliquant la transformation RGB à YCrCb, et en effectuant pour les matrices Cr et Cb une moyenne sur quatre pixels. Notons qu'il s'agit pour l'instant d'une compression conservative, dans la mesure ou nous pouvons retrouver les valeurs initiales, si on ne considère pas les erreurs d'arrondi. Nous obtenons donc, trois matrice d'entiers, au lieu d'une matrice de pixels :
- matrixY
- matrixCr
- matrixCb

Cette opération étant effectuée par la méthode $rgbToYuv$:

```
void ImgBmp::rgbToYuv(ImgJpeg *jpeg){
   jpeg->makeMatrixY(matrix);
   jpeg->makeMatrixCb(matrix);
   jpeg->makeMatrixCr(matrix);
}
```

[5] YUV est une autre notation, dans laquelle U correspond a Cr et V à Cb.
[6] Ces formules interviendrons dans l'algorithme de décompression.

Ansi, lorsque l'on passe au mode Y Cr Cb on obtient des matrices matrixCr et matrixCb deux fois plus petites que la matrice BMP ; et puisque cette dernière est composée de blocs de 16×16, alors Cr et Cb sont composées de blocs de 8×8. Cette implémentation est du au fait que la DCT est optimale (rapport qualité/coût) lorsqu'elle travaille sur des blocs de 8×8.

3.3 Discret Cosin Transform

La DCT s'applique uniquement sur des matrices carrées de dimension N ; et dans le cas de de l'algorithme $JPEG$, il s'agit typiquement de matrice 8×8. La variable $img(x, y)$ représente la matrice de l'image et $dct(u, v)$ la matrice résultante. Ainsi, les indices x, y, u et v varient de 0 à $N - 1$, ou N est la dimension de la matrice. Si l'on ne tient pas compte des erreurs d'arrondis, on peut considérer que le calcul de la DCT est une transformation conservatrice (i.e. sans perte de qualité). La formule de la DCT est donnée ci-dessous, suivit plus loin, du fragment de code correspondant.

$$dct(u,v) = \frac{2}{N} c(u)c(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} img(x,y) cos\left[\frac{\pi}{N} u\left(x + \frac{1}{2}\right)\right] cos\left[\frac{\pi}{N} v\left(y + \frac{1}{2}\right)\right]$$

$$avec \begin{cases} c(0) = \frac{1}{\sqrt{2}} \\ c(w) = 1 pour w = 1, 2, \ldots, N-1 \end{cases}$$

```
// calcul de la DCT
for(u=0;u<8;u++) {
  for(v=0;v<8;v++) {
    sigma = 0.0f;
    for(x=0;x<8;x++)
      for(y=0;y<8;y++)
        sigma += image[x][y] * tabCos[u][x] * tabCos[v][y];
    if(u) cu = 1;                // cu = 1 si u!=0
    else cu = 0.70710678118654;  // cu = 1/sqrt(2)
    if(v) cv = 1;                // cv = 1 si v!=0
    else cv = 0.70710678118654;  // cv = 1/sqrt(2)
    dct[u][v] = (int)(0.25f * cu * cv * sigma);
  }
}
```

Le calcule de la DCT impose l'imbrication de quatres boucles de $0 = N - 1$, et c'est pourquoi l'algorithme $JPEG$ considère des matrices de dimension faible ($N = 8$), de façon à éviter une explosion combinatoire. En effet, ce calcul est très coûteux, d'autant plus que le calcul des *cosinus* en machine est loin d'être négligeable. Or d'une manière générale, plus le calcule de la DCT est basé sur une matrice de grande taille (idéalement l'image entière), plus les résultats sont pertinents. Mais le calcul de la DCT sur l'image entière n'est, dans l'état actuel des choses, pas envisageable.

Cependant, il est possible d'accélérer grandement le calcul de la DCT en considérant une matrice de *cosinus* qui contient l'ensemble des variables pré-calculées, qui entrent en jeu dans l'algorithme.

3.4 La quantification

3.4.1 Introduction

La quantification représente la phase non conservatrice du processus de compression JPEG. La transformée en cosinus nous donne des informations sur la fréquence ce qui va permettre de filtrer des informations. En effet, non seulement l'information effective de la plupart des images naturelles est concentrée dans les basses fréquences, mais de plus notre oeil est beaucoup moins sensible aux fréquences élevées (changements de couleur brusques) qu'aux fréquences basses (changements lents). La quantification consiste donc à diminuer la précision des fréquences élevées, en divisant chaque élément DCT par l'élément correspondant dans la table de quantification (matrice $8 * 8$ selon la norme), et en arrondissant à l'entier le plus proche. Ainsi beaucoup d'éléments deviendront nuls ou très faibles et occuperont donc moins de place.

Ultérieurement, lors de la restitution de l'image (décompression), il suffira de réaliser l'opération inverse (déquantification) en multipliant chaque valeur de la matrice quantifiée par le quantum correspondant, pour retrouver une matrice DCT déquantifiée, à partir de laquelle sera établie la matrice des pixels de sortie. La valeur du quantum peut être d'autant plus élevée que l'élément correspondant de la matrice DCT contribue peu à la qualité de l'image, donc qu'il se trouve éloigné du coin supérieur gauche (i=j=0).

C'est pourquoi les matrices de quantification comportent généralement des valeurs constantes selon des diagonales ascendantes (i, j, i-1, j+1), mais croissantes d'une diagonale à la suivante : cet accroissement constitue le pas du quantum, ou le " facteur de qualité ", car la perte de précision sera d'autant plus grande que ce pas sera élevé.

Pour des résultats optimums, deux formules distinctes doivent être appliquées pour le calcul les coefficients de la matrice de quantification pour les matrices de chrominance et la matrice de luminance :

$$\boxed{\text{Luminance} : 1 + (1 + i + j) * factor}$$

$$\boxed{\text{Chrominance} : 1 + (1 + i * i + j * j) * factor}$$

3.4.2 Implémentation

Nous avons créé une matrice de quantification pour chaque type de matrice, c'est à dire une matrice de quantification pour la matrice de luminance (matrixY) et une matrice de quantification pour celles de chrominance (Cr et Cb) afin d'obtenir de meilleurs résultats (qualitatif).
Voici les méthodes qui crées les matrices de quantification

```
void ImgJpeg::initQuantY(int factor) {
  int i,j;
  for(i=0;i<8;i++)
    for(j=0;j<8;j++)
      quantY[i][j] = 1 + ( 1 + i + j ) * factor;
}
```

```
void ImgJpeg::initQuantC(int factor) {
  int i,j;
  for(i=0;i<8;i++)
    for(j=0;j<8;j++)
      quantC[i][j] = 1 + ( 1 + i*i + j*j ) * factor;
}
```

Nous appliquons ensuite la phase de quantification sur chacune des matrices à l'aide de la matrice de quantification correspondante.
Voici les codes des méthodes réalisant cette opération :

```
void ImgJpeg::QuantY() {
  int i,j;
   for(i=0;i<8;i++)
    for(j=0;j<8;j++)
      dct[i][j]= dct[i][j]/quantY[i][j];
}

void ImgJpeg::QuantC() {
  int i,j;
   for(i=0;i<8;i++)
    for(j=0;j<8;j++)
      dct[i][j]= dct[i][j]/quantC[i][j];
}
```

3.5 La Linéarisation

3.5.1 La lecture *zigzag*

Afin de pouvoir écrire la matrice dans un fichier, il nous faut une suite d'octets, ie. effectuer une *linéarisation*. On va pour cela employer une méthode dite *zigzag*, de manière à regrouper ensemble les éléments les moins importants(voir figure 2 page 12).

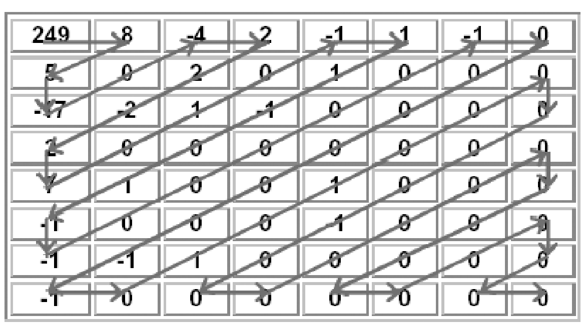

FIG. 2 – **La séquence zigzag**

On obtient, après la lecture zigzag, la suite :
{ 249, 8, 5, -17, 0, -4, 2, 2, -2, 2, 7 ,0, 1, 0, -1, 1, 1, -1, 0, 1, -1, -1, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, -1, -1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}.

On constate un nombre important de *zeros* qui vont permettent un codage efficaces par RLE.

3.5.2 Le codage RLE

La méthode de compression RLE *(Run Length Encoding)*, appelée aussi RLC, est une méthode relativement simple : le principe est de remplacer une séquence de n éléments v identiques par un couple *(n,v)*.

Par exemple, dans notre implémentation, la séquence 00003266 sera codé par la suite 403266 (i.e. on ne code que les *zeros* successifs).

3.5.3 Implémentation

Nous effectuons la lecture zigzag sur un bloc de $8*8$ puis la RLE avant de passer au bloc suivant. La séquence obtenue après la lecture zigzag est stockée dans un buffer, puis on applique la RLE sur ce buffer avant de passer au traitement du bloc suivant. Ci-dessous figure un fragment de code réalisant la lecture zigzag.

```
while((x!=7) || (y!=7)){
    // diagonale up
    while((x!=0)&&(y!=7)){
      buf[pos++] = dct[x][y];
      x--;
      y++;
    }
    buf[pos++] = dct[x][y];
    if(y==7)x++;
    else y++;

    // diagonale down
    while((x!=7)&&(y!=0)){
      buf[pos++] = dct[x][y];
      x++;
      y--;
    }
    buf[pos++] = dct[x][y];
    if(x==7)y++;
    else x++;
  }
  buf[pos] = dct[x][y];
```

3.6 Écriture du fichier JPEG

Nous avons crée un fichier JPEG propriétaire afin de stoquer la séquence *zigzag*, en utilisant le moins d'octets possibles. Pour cela, il est impératif de considérer une gestion de fichier bit à bit, et non octets par octets. Afin de manipuler aisément les lectures et écritures des fichiers JPEG, nous avons implémenté une classe $BFile$, qui permet d'effectuer des lectures et écritures *bit à bit*. Dans le cas de la création du fichier JPEG, il s'agit décrire le contenue du buffer RLE, *bit à bit*.

Pour se faire, la classe $BFile$ dispose des méthodes :

- `void writeBit(int bit) ;`
- `void writeNBytes(int nbBytes, unsigned char *bytes) ;`
- `void writeBitsEncode(int num) ;`

La première méthode permet d'écrire *un* bit, alors que la seconde permet d'écrire *n bits* consécutifs. Enfin, la méthode $writeBitsEncode(intnum)$ permet d'écrire un *entier* encodé en RLE, comme suit :
- on écrite la taille t de l'entier à écrire sur quatres bits (de 0 à 255),
- puis on écrite l'entier sur t bits (de 0 à 65535).

On minimise ainsi la place occupé par chaque entier.

Ci-dessou figure un un fragment de code qui assure l'écriture du fichier JPEG :

```
jpeg->bFile = new BFile(dst,"w"); // fichier destination
jpeg->bFile->writeNBytes(2,FileType);          // écrit l'entête
...
jpeg->bFile->writeInt4Bytes(jpeg->bufferSize); // écrit la taille du buffer RLE jpeg
jpeg->bFile->writeBitsEncode(jpeg->facteur);   // écrit facteur (0<x<100)
jpeg->bFile->writeBitsEncode(widthTrue);       // écrit taille
jpeg->bFile->writeBitsEncode(heightTrue);      // écrit taille
for(i=0;i<jpeg->bufferSize;i++)                // écrit le contenue de la RLE
  jpeg->bFile->writeBitsEncode(jpeg->buffer[i]);
jpeg->bFile->close();
```

La structure d'un fichier JPEG[7], est donc la suivante :

```
54 octets : entête
 4 octets : la taille du buffer RLE
 x octets : le facteur de compression
 x octets : la largeur de l'image
 x octets : la hauteur de l'image
 x octets : le buffer RLE
```

4 Décompréssion

4.1 Le chargement JPEG

Pour lire un fichier JPEG, on utilise le constructeur $ImgJpeg(char*fileName)$, dont on donne une aperçu du code :

```
ImgJpeg::ImgJpeg(char *fileName) {       // construit et load un fichier jpg
  bFile = new BFile(fileName,"r");       // ouvre le fichier jpg en lecture bit a bit
  ...
  bFile->readNBytes(2,&infoBmpFile[0]);  // Header du fichier bmp [0->13]
  bFile->readNBytes(4,&infoBmpFile[2]);
  ...
  facteur = bFile->readEncodedInt();     // facteur (sur 15 bits au plus)
  ...
  // read all JPEG buffer, encoded RLE
  for(i=0;i<bufferSize;i++) buffer[i] = bFile->readEncodedInt();
  ...
  bFile->close();
```

[7] Ou les valeurs "x" sont lues dynamiquements dans le fichier.

Tout d'abord, on créer une instance de la classe $BFile$; qui "ouvre" le fichier JPEG à charger en mode bit à bit. Ensuite, utilise tour à tour les méthodes :
- `void readNBytes(int nbBytes, unsigned char *bytes) ;`
- `int readEncodedInt() ;`

La première méthode permet de lire $nbBytes$ octets du fichier, et de les ranger à l'adresse $bytes$. Il s'agit donc d'une lecture classique octets par octets. En revanche la méthode $readEncodedInt()$ permet d'effectuer lecture bit à bit d'un *entier* encodé en RLE. Rappelons que le fichier est codé de la façon suivante :
- on lit quatres bits qui codent la taille t de l'entier à lire (de 0 à 15 bits),
- on lit t bits, qui codent la veleur de l'entier (de 0 à 32767).

Une foi que l'image JPEG est chargé en mémoire, on applique la méthode $jpegToBmp()$ à cette instance. Dans le corps de cette méthode, on créer tout d'abord un object BMP "vide" de taille $width \times height$, puis on rempli les champs de l'entête BMP grâce à ceux de l'objet JPEG. Ensuite, on parcour le buffer RLE (en ecrivant les $zero$ consequtifs), puis pour chaque blocs de 64,on appliqueles méthodes $zigzagInv()$, $invQuant()$, $invDCT()$. On obtient alors les trois matrices Y, C_b et C_r.

4.2 Dé-linéarisation

Il s'agit de repositionner les valeurs dans les matrices. On procéde pour cela matrice par matrice, tout comme lors de la linéarisation. cete fois on prend un buffer en entrée et l'on va replacer les valeurs à leur position dans la matrice.

```
void ImgJpeg::zigzagInv() {
  int x   = 0;
  int y   = 0;
  int pos = 0;

  while((x!=7) || (y!=7)){
    // diagonale up
    while((x!=0)&&(y!=7)){
      dct[x][y] = buf[pos++];
      x--;
      y++;
    }
    dct[x][y]= buf[pos++];
    if(y==7)x++;
    else y++;

    // diagonale down
    while((x!=7)&&(y!=0)){
      dct[x][y]= buf[pos++];
      x++;
      y--;
    }
    dct[x][y]= buf[pos++];
    if(x==7)y++;
    else x++;
  }
```

```
  dct[x][y]= buf[pos++];
}
```

4.3 Dé-quantification

Il nous faut maintenant appliquer l'opération inverse de la quantification, cette opération est réalisée en effectuant la multiplication de chaque coefficient, par le coefficient correspondant dans la matrice de quantification. On re-crée donc les matrices de quantification correspondante à l'aide du facteur de qualité sauvegardé dans l'entête de notre image jpeg.
voici donc les méthodes réalisant la demagnification :

```
void ImgJpeg::invQuantY() {
  int i,j;
   for(i=0;i<8;i++)
    for(j=0;j<8;j++)
      dct[i][j]= dct[i][j]*quantY[i][j];
}

void ImgJpeg::invQuantC() {
  int i,j;
   for(i=0;i<8;i++)
    for(j=0;j<8;j++)
      dct[i][j]= dct[i][j]*quantC[i][j];
}
```

4.4 Inv-DCT

La formule est tout a fait analogue à celle de la DCT :

$$img(x,y) = \frac{2}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1} c(u)c(v)F(u,v)cos\left[\frac{\pi}{N}u\left(x+\frac{1}{2}\right)\right]cos\left[\frac{\pi}{N}v\left(y+\frac{1}{2}\right)\right]$$

$$avec\begin{cases} c(0) = \frac{1}{\sqrt{2}} \\ c(w) = 1 pour w = 1,2,\ldots,N-1 \end{cases}$$

La portion de code correspondant figure ci-dessous :

```
for(x=0;x<8;x++) {
  for(y=0;y<8;y++) {
    // init le SIGMA a zero
    sigma = 0.0f;
    // calcul du SIGMA...
    for(u=0;u<8;u++)
      for(v=0;v<8;v++) {
        if(u) cu = 1;                // cu = 1 si u!=0
        else cu = 0.70710678118654;  // cu = 1/sqrt(2)
        if(v) cv = 1;                // cv = 1 si v!=0
        else cv = 0.70710678118654;  // cv = 1/sqrt(2)
        sigma += cu * cv * dct[u][v] * tabCos[u][x] * tabCos[v][y];
      }
      // calcul d'un coefficiant
      matrix[x+iStart][y+jStart] = (int)(0.25f * sigma)+128;
  }
}
```

4.5 Passage du mode YUV au mode RGB

4.5.1 Le codage Luminance, Chrominance

Il suffit cette fois d'appliquer les formules inverse. Voici les formules qui permettent de repasser au mode RGB.

- $R = Y + 1.402 * Cr - 128$
- $G = Y - 0.34414 * (Cb - 128) - 0.71414 * (Cr - 128)$
- $B = Y + 1.772 * (Cb - 128)$

4.5.2 Implémentation

Le retour au mode RGB s'effectue par appel de la méthode $yuvToRgb()$. Voici le fragment de code qui permet de repasser du mode YCrCb au mode RGB. La encore il faut penser a effectuer les tests sur les valeurs RGB afin de rester dans l'interval [0,255], sinon on peut générer des pixels erronés.

```
//on retire 128 ajouté lors de rgbtoyuv afin d'eliminer les valeures negatives
  for(row=0;row<heightTrue;row++){
    val1 = row/2;
    for(column=0;column<widthTrue;column++){
      val2 = column/2;
      // on calcul la valeur de R que l'on stocke dans C[0]
      tmp = (int)(matrixY[row][column] + 1.402*(matrixCr[val1][val2] -128));
      if(tmp<0) tmp=0;
      if(tmp>255) tmp = 255;
      c[0] = (unsigned char) tmp;
      // on calcul la valeur de G que l'on stocke dans C[1]
      tmp = (int)(matrixY[row][column] - 0.34414*(matrixCb[val1][val2]-128)
             -0.71414*(matrixCr[val1][val2]-128 ));
      if(tmp<0) tmp=0;
      if(tmp>255) tmp = 255;
      c[1] = (unsigned char)tmp;
      // on calcul la valeur de B que l'on stocke dans C[2]
      tmp = (int)(matrixY[row][column] + 1.772*(matrixCb[val1][val2] -128));
      if(tmp<0) tmp=0;
      if(tmp>255) tmp = 255;
      c[2] = (unsigned char) tmp;
      // rappel: on donne au pixel les parametres  RGB ds l'ordre  R G B
      matrix->matrix[row][column]->setRGB(c[0],c[1],c[2]);
    }
  }
```

5 L'interface

L'interface de "visionnage" (ou *viewer*) à été conçue en TCL/TK, pour ses qualités de portabilité. Nous proposons différentes *skins* (ou peaux), afin que chacun trouve son bonheur(voir figure 3 page 18). L'interface permet, en outre :

- d'ouvrir une image source (BMP ou JPG)
- de l'afficher à l'écran
- de choisir le niveau de compression (scroll barre)

- de compresser une image BMP en image JPG
- afficher le taux exacte de compression en pourcentage
- de décompresser une image JPG en image BMP
- afficher le taux exacte de dé-compression en pourcentage
- afficher l'image résultante
- changer d'apparence du viewer

FIG. 3 – **L'interface : divers skins**

6 Les tests

6.1 Compression

6.1.1 Rentabilité

Nous avons effectué plusieurs series de tests comparatif, entre notre algorithme, et l'algorithme JPG de *Photoshop* et *Gimp*. Les résultats sont très concluants ; car pour des images résultantes semblables (i.e. on ne fait pas la différence entre le JPG obtenu par notre algorithme, et celui obtenu par photoshop) les fichiers JPG sont de taille sensiblement égale. La ou notre fichier fait 320 Ko, celui de photoshop fait 302 Ko (pour une image BMP de a Mo 1280*1024). Cette sensible différence s'explique peut-être par la compression de Huffman que nous n'avon pas implémenté, et aussi parceque l'entête de nos fichiers JPG aurait put être diminuée.

Néanmoins, la figure 4 page 19 montre qu'une image BMP(1280*1024) de 3,8 Mo peut être compressée en une image JPG de 188 Ko, sans pertes de qualité apparente (soit 96% d'économie!!!).

FIG. 4 – **Facteur 75 :** 96% d'économie, pas de perte de qualité.

6.1.2 Vitesse

Notre algorithme est apparemment très rapide : immédiat pour les petite images, et ne dépasse pas 6 seconde pour les images de très haute résolution (1280*1024 et 2196*1746).

6.1.3 facteur/qualité

Si le facteur est important (voir maximum = 99%) on obtient un fichier JPG de l'ordre de 20%, par rapport à la taille du fichier BMP. D'autre par, lorsque le facteur est élevé (de 80 à 99%), les déperdition sont difficilement décelables, voir même inexistantes pour l'oeil humain (mais cela est subjectif!). D'autant plus que la nature de l'image influe sur la qualité obtenue par la compression JPEG. D'un manière générale, nous avons utilisé des images de très bonne qualité, afin de pouvoir observer facilement les déperditions (s'il y en a).

Nous avons d'autre part effectué des tests sur des images de taille diverses, et nous avons constaté qu'avec un facteur compris entre 70 et 80%, a compression offrait le meilleur rapport qualité/taille fichier. En effet, les défaut sont alors, à peine observables, et le fichier à une taille de l'ordre de 3% par rapport au fichier d'origine.

En deçà de 70%, les pertes sont réellement visibles, et la taille d'u fichier JPG n'est pas excessivement moins importante (de 2 à 3%). La figure 5 page 20 montre les différents rendus en fonction du facteur qualité.

6.1.4 Taille image / Temps

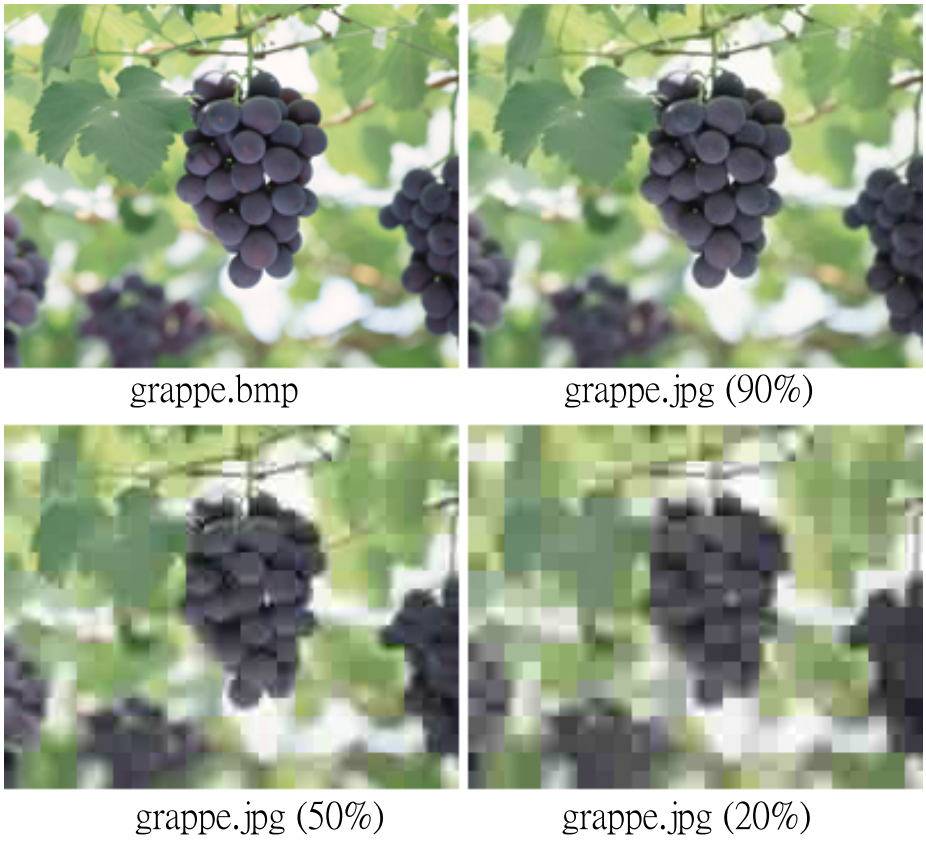

FIG. 5 – **Facteur/qualité**

Image	ovd.bmp	ffx_small.bmp	Grape.bmp	Apples.bmp	ffx.bmp	Apples_huge.bmp
Dim	100*100	640*480	1280*1024	1280*1024	1280*1024	2196*1746
Bmp_size	30 Ko	901 Ko	3 841 Ko	3 841 Ko	3841 Ko	11 234 Ko
Jpg_size	16 Ko	337 Ko	761 Ko	579 Ko	717 Ko	1 331 Ko
Time	0.0 sec	0.9 sec	2.1 sec	2.1 sec	2.7	6.3 sec

Pour ces tests, nous avons utilisé un facteur de compression 99%, la taille du fichier JPG est donc inférieur au fichier BMP, mais elle pourrai être largement diminuée. En revanche, ce facteur fait qu'il est impossible d'observer une quelconque différence, entre les images sources et destination (même si cela est subjectif !). Les mesures de temps ont étés faite grâce à la bibliothèque $< time.h >$.

On constate que les images de même taille demandent des temps de calcul du même ordre. Cependant, il faut prendre en compte le fait qu'un processus peut prendre la main plus souvent qu'un autre, ce qui fausse les mesures. Les mesures prennent en compte le chargement du fichier source, et l'écriture du fichier destination ; et donc, il se peut que la taille du buffer RLE influe sur le temps de calcul. Enfin, on constate que la taille du fichier JPG n'est pas forcement identique, même si les images BMP ont même taille. Ceci s'explique par le fait qu'une matrice *zigzag* peut contenir un grand nombre de *zeros*,

favorisant ainsi la compression RLE.

6.2 Décompression

Pour toutes les images nous retombons exactement sur la même dimensions et même taille (heureusement !). On constate que la décompression est légèrement plus lente que la compression.

Image	ovd.jpg	ffx_small.jpg	Grape.jpg	Apples.jpg	ffx.jpg	Apples_huge.jpg
Dim	100*100	640*480	1280*1024	1280*1024	1280*1024	2196*1746
Jpg_size	16 Ko	337 Ko	761 Ko	579 Ko	717 Ko	1 331 Ko
Bmp_size	30 Ko	901 Ko	3 841 Ko	3 841 Ko	3841 Ko	11 234 Ko
Time	0.0 sec	1 sec	3.0 sec	2.7 sec	2.9	6.9 sec

La figure 6 page 21 montre qu' on obtient une image BMP identique à l'image d'origine BMP, en effectuant un cycle de compression/decompression, avec un facteur de 90.

FIG. 6 – **bmp2jpg, puis jpg2bmp :** pas de pertes de qualité.

7 Annexe

7.1 Les chiffres

Le projet pèse 2914 lignes des codes compactes (Makefile et TCL inclus).

7.2 Les fichiers

L'archive du projet comporte un répertoire *Src* qui contient les source du projet (soit, en tout 16 fichiers). Chaque classe C++ possède un fichier entête (header) *.hh*, et le corps des méthodes est contenu dans un fichier *.cc*.

Voici la liste des classe :
- Pixel.hh
- Matrix.hh
- Window.hh

- BFile.hh
- ImgBmp.hh
- ImgJpeg.hh

Voici la liste des commandes UNIX :
- bmp2jpg.cc
- jpg2bmp.cc
- draw.cc

Les autres fichiers :
- viewer.tcl
- Makefile

Le répertoire *Data* contient des ressources : comme des images pour les *skins* de l'interface.

Le repertoire *Images* est en général utilisé pour stocker les images sur lesquelles nous travaillons.

7.3 Le Makefile

```
CC=g++ -c
LN=g++
LIB=-lm
XLIB=-L/usr/X11R6/lib -lX11

all: pixel matrix bfile imgbmp imgjpeg window draw bmp2jpg jpg2bmp

pixel: Src/Pixel.cc
    ${CC} Src/Pixel.cc
matrix: Src/Matrix.cc
    ${CC} Src/Matrix.cc
bfile: Src/BFile.cc
    ${CC} Src/BFile.cc
imgbmp: Src/ImgBmp.cc
    ${CC} Src/ImgBmp.cc
imgjpeg: Src/ImgJpeg.cc
    ${CC} Src/ImgJpeg.cc
window: Src/window.cc
    ${CC} Src/window.cc
draw: Src/draw.cc
    ${LN} Matrix.o Pixel.o BFile.o ImgBmp.o ImgJpeg.o window.o -o draw Src/draw.cc ${XLIB}
bmp2jpg: Src/bmp2jpg.cc
    ${LN} Matrix.o Pixel.o BFile.o ImgBmp.o ImgJpeg.o window.o -o bmp2jpg Src/bmp2jpg.cc ${XLIB}
jpg2bmp: Src/jpg2bmp.cc
    ${LN} Matrix.o Pixel.o BFile.o ImgBmp.o ImgJpeg.o window.o -o jpg2bmp Src/jpg2bmp.cc ${XLIB}
cleanBMP:
    rm -f Images/*.BMP
cleanJPG:
    rm -f Images/*.JPG
cleanimages:
    rm -f Images/*.BMP Images/*.JPG
clean:
    rm -f draw bmp2jpg jpg2bmp main *.o Src/*.o core *.stackdump  Src/*.hh~ Src/*.cc~ makefile~ *.tcl~
```