Ensimag — Printemps 2013

Projet Logiciel en C

Sujet : Conversion d'images JPEG

Auteurs : Des enseignants actuels et antérieurs du projet C

Table des matières

1 Objectif **5**
- 1.1 À propos du JPEG . . . 5
- 1.2 Principe . . . 6
- 1.3 Représentation des données . . . 6
- 1.4 Lecture et analyse du flux . . . 7
- 1.5 Décompression . . . 7
  - 1.5.1 Le codage de Huffman . . . 7
  - 1.5.2 Arbres DC et codage DPCM . . . 9
  - 1.5.3 Arbres AC et codage RLE . . . 9
- 1.6 Zigzag inverse et Quantification inverse . . . 10
- 1.7 Transformée en cosinus discrète inverse . . . 10
- 1.8 Reconstitution des MCUs . . . 10
- 1.9 Conversion vers des pixels RGB . . . 12

2 Principe et base de code **13**
- 2.1 Environnement logiciel et fonctionnement . . . 13
  - 2.1.1 Principe . . . 13
  - 2.1.2 Entrées/Sorties du programme . . . 13
- 2.2 Squelette de code fourni et interfaces . . . 13
  - 2.2.1 Présentation des types de données utilisés . . . 14
  - 2.2.2 Les fonctions à implémenter . . . 14
  - 2.2.3 Écriture du résultat . . . 17

3 Travail attendu **19**
- 3.1 Partie obligatoire : décodeur . . . 19
  - 3.1.1 Étapes . . . 19
  - 3.1.2 Minimum requis . . . 20
- 3.2 Extensions . . . 20
- 3.3 Outils . . . 20
- 3.4 Informations . . . 21

A Le format JPEG **23**
- A.1 Principe du format JPEG . . . 23
- A.2 Sections JPEG . . . 23
  - A.2.1 Petit point sur les indices . . . 23
  - A.2.2 APPx - Section Application . . . 23
  - A.2.3 SOFx - Start Of Frame . . . 24
  - A.2.4 DQT - Define Quantization Table . . . 24
  - A.2.5 DHT - Define Huffman Tables . . . 25
  - A.2.6 SOS - Start Of Scan . . . 25
  - A.2.7 COM - Commentaire . . . 26
- A.3 Récapitulatif . . . 26

B Le format TIFF **27**
B.1 Principe du format 27
B.2 Entête du fichier 27
B.3 Image File Directory 27
B.3.1 Types de données 28
B.4 Tags Utiles 28
B.4.1 Caractéristiques de l'image 28
B.4.2 Information de stockage 28
B.4.3 Divers 29
B.4.4 Récapitulatif 29
B.5 Exemple 29

Chapitre 1

Objectif

L'objectif de ce projet est de réaliser un convertisseur d'image au format JPEG en langage C, en implantant au minimum la partie décodeur. Ce décodeur traitera des images encodées en JPEG, pour les écrire dans un format TIFF non compressé (ce qui signifie que les pixels de l'image apparaissent en clair à un endroit dans le fichier, et seront affichés facilement et rapidement y compris par des logiciels de visualisation basiques).

Il vous sera également proposé de réaliser une extension qui consiste en un réencodage du fichier à partir du décodeur.

Le format JPEG est l'un des formats les plus utilisés en matière d'image numérique. Il est en particulier utilisé comme format compressé par la plupart des appareils photo numériques, étant donné que le coût de calcul est acceptable, et que la taille de l'image résultante reste petite.

Le but du projet étant d'écrire du langage C, les informations structurelles et algorithmiques vous seront fournies en totalité, et il est bien sûr possible d'aller chercher des compléments par ailleurs (site Web, etc).

1.1 À propos du JPEG

Le JPEG (*Joint Photographic Experts Group*), est un comité de standardisation pour la compression d'image dont le nom a été détourné pour désigner une norme en particulier, la norme JPEG, que l'on devrait en fait appeler ISO/IEC IS 10918-1 | ITU-T Recommendation T.81.

Cette norme spécifie plusieurs alternatives pour la compression des images en imposant des contraintes uniquement sur les algorithmes et les formats du décodage. Notez que c'est très souvent le cas pour le codage source (ou compression en langage courant), car les choix pris lors de l'encodage garantissent la qualité de la compression. La norme laisse donc la réalisation de l'encodage libre d'évoluer. Pour une image, la compression est évaluée par la réduction obtenue sur la taille de l'image, mais également par son impact sur la perception qu'en a l'œil humain. Par exemple, l'œil est plus sensible aux changements de luminosité qu'aux changements de couleur. On préférera donc compresser les changements de couleur que les changements de luminosité, même si cette dernière pourrait permettre de gagner encore plus en taille. C'est l'une des propriétés exploitées par la norme JPEG.

Parmi les choix proposés par la norme, on trouve des algorithmes de compression avec ou sans pertes (une compression avec pertes signifie que l'image décompressée n'est pas strictement identique à l'image d'origine) et différentes options d'affichage (séquentiel, l'image s'affiche en une passe pixel par pixel, ou progressif, l'image s'affiche en plusieurs passes en incrustant progressivement les détails, ce qui permet d'avoir rapidement un aperçu de l'image, quitte à attendre pour avoir l'image entière).

Dans son ensemble, il s'agit d'une norme plutôt complexe qui doit sa démocratisation à un format d'échange, le JFIF (JPEG File Interchange Format). En ne proposant au départ que le minimum essentiel pour le support de la norme, ce format s'est rapidement imposé, notamment sur Internet, amenant à la norme le succès qu'on lui connaît aujourd'hui. D'ailleurs, le format d'échange JFIF est également confondu avec la norme JPEG. Ainsi, un fichier possédant une extension .jpg ou .jpeg est en fait un fichier au format JFIF respectant la norme JPEG. Évidement, il existe d'autres formats d'échange supportant la norme JPEG comme les formats TIFF ou EXIF. La norme de compression JPEG peut aussi être utilisée pour encoder de la vidéo, dans un format appelé Motion-JPEG. Dans ce format, les images sont toutes enregistrées à la suite dans un flux. Cette stratégie permet d'éviter certains artefacts liés à la compression inter-images dans des

formats types MPEG.

Le décodeur JPEG demandé dans ce projet (ainsi que l'extension d'encodeur) doit supporter le mode dit "baseline" (compression avec pertes, séquentiel, Huffman). Ce mode est utilisé dans le format JFIF, et il est décrit dans la suite de ce document.

1.2 Principe

Bien que le projet s'attaque initialement au décodage, le principe s'explique plus facilement du point de vue du codeur.

Tout d'abord, l'image est partitionnée en blocs appelés macroblocs ou MCUs pour Minimum Coded Unit. Ceux-ci sont en général de taille 8 x 8 pixels. Ensuite, chaque MCU est traduite dans le domaine fréquentiel par transformation en cosinus discrète (DCT). Le résultat de ce traitement, appelé MCU fréquentielle, est encore un bloc 8×8, mais de fréquences et non plus de pixels. On y distingue une composante continue *DC* à la case $(0,0)$ et 63 composantes fréquentielles *AC*[1]. Les plus hautes fréquences se situent autour de la case $(7,7)$.

L'œil étant peu sensible aux hautes fréquences, il est plus facile de les filtrer avec cette représentation fréquentielle. Cette étape de filtrage, dite de quantification, détruit de l'information pour permettre d'améliorer la compression, au détriment de la qualité de l'image (d'où l'importance du choix du filtrage). Elle est réalisée MCU par MCU à l'aide d'un filtre de quantification spécifique à chaque image. La MCU fréquentielle filtrée est ensuite parcourue en zigzag (ZZ) afin de transformer la MCU en vecteur de 1×64 fréquences avec les hautes fréquences en fin. De la sorte, on obtient statistiquement plus de 0 en fin de MCU.

Cette MCU vectorisée est alors compressée en utilisant successivement deux codages sans perte : d'abord un codage RLE (voir section 1.5) pour exploiter les répétitions de 0, puis un codage entropique[2] dit codage de *Huffman* qui utilise un dictionnaire spécifique à l'image en cours de traitement.

Finalement, le bitstream est construit par concaténation de séquences de données brutes séparées par des marqueurs qui précisent la longueur et le contenu des données associées. Le format et les marqueurs sont spécifiés dans l'annexe A.

Le décodeur qui est le but de ce projet effectue les opérations dans l'ordre inverse, comme illustré sur la figure 1.1.

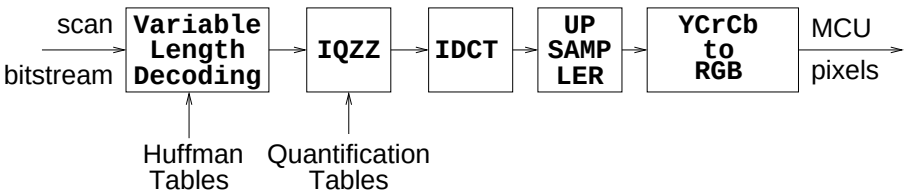


FIGURE 1.1 – Principe du décodage JPEG

1.3 Représentation des données

Il existe plusieurs manières de représenter une image. Une image numérique est en fait un tableau de pixels, chaque pixel ayant une couleur distincte. Dans le domaine spatial, le décodeur utilise deux types de représentation de l'image.

Le format ARGB, qu'on appellera également RGB dans ce document, qui est le format le plus courant, est le format utilisé en sortie. Il représente chaque couleur de pixel en donnant la proportion de 3 couleurs : le rouge (R), le vert (G), et le bleu (B). Une information de transparence (A) est également fournie, mais elle ne sera pas utilisée dans ce projet. Le format ARGB est le format utilisé en amont et en aval du décodeur.

Un deuxième format, appelé YCbCr, utilise une autre stratégie de représentation, en 3 composantes : une luminance dite Y, une différence de chrominance bleue dite Cb, et une différence de chrominance rouge dite Cr. Le format YCbCr est le format utilisé en interne pour le JPEG.

1. Il s'agit là de fréquences spatiales 2D avec une dimension verticale et une dimension horizontale
2. C'est à dire qui minimise la quantité d'information nécessaire pour représenter un message, aussi appelé entropie

Une confusion est souvent réalisée entre le format YCbCr, qui est celui utilisé par la norme JPEG, et le format YUV [3]. Ainsi, dans la suite du document, si on parle de YUV ou des composantes U et V, on se réfère en faite au YCbCr, et aux composantes Cb et Cr.

Cette stratégie est plus efficace que le RGB (*red, green, blue*) classique, car d'une part les différences sont codées sur moins de bits que les valeurs et d'autre part elle permet des approximations (ou de la perte) sur la chrominance à laquelle l'œil est moins sensible.

1.4 Lecture et analyse du flux

La phase de lecture du flux vidéo effectue une analyse grossière du *bitstream* considéré comme un flux d'octets, afin de déterminer les actions à effectuer. Le bitstream, comme évoqué précédemment, représente l'intégralité de l'image à traiter.

Ce flux de données est un flux ordonné, constitué d'une série de marqueurs et de données. Les marqueurs permettent d'identifier ce que représentent les données qui les suivent. Cette identification permet ainsi, en se référant à la norme, de connaître la sémantique des données, et leur signification (*i.e* les actions à effectuer pour les traiter). Un marqueur et ses données associées représentent une section.

Deux grands types de sections peuvent être distinguées :

– définition de l'environnement : ces sections contiennent des données permettant d'initialiser le décodage du flux. La plupart des informations du JPEG étant dépendantes de l'image, c'est une étape nécessaire. Les informations à récupérer concernent, par exemple, la taille de l'image, ou les tables de Huffman utilisées ;
– représentation de l'image : ce sont les données brutes qui contiennent l'image encodée.

La définition de l'environnement consiste généralement en une récupération des données propres à l'image. Ces données sont stockées pour les réutiliser ensuite lors du décodage de l'image. Elles peuvent nécessiter un traitement particulier avant d'être utilisable.

Une liste exhaustive des marqueurs est définie dans la norme JFIF. Les principaux vous sont donnés en annexe de ce document, avec la représentation des données utilisée, et la liste des actions à effectuer. On notera cependant ici 4 marqueurs importants :

SOI : le marqueur *Start Of Image* représente le début de l'image,

EOI : le marqueur *End Of Image* représente la fin de l'image,

SOF : le marqueur *Start Of Frame* marque le début d'une *frame* JPEG, c'est à dire le début de l'image effectivement encodée. Le marqueur SOF est associé à un numéro, qui permet de repérer le type d'encodage utilisé. Dans notre cas, ce sera toujours un SOF0. La section SOF contient la taille de l'image et les facteurs de sous-échantillonage utilisés.

SOS : le marqueur *Start Of Scan* indique le début de l'image encodée (données brutes).

1.5 Décompression

1.5.1 Le codage de Huffman

Les codes de Huffman sont appelés codes *préfixes*. C'est une technique de codage statistique à longueur variable.

Les codes de Huffman associent aux symboles les plus utilisés les codes les plus petits et aux symboles les moins utilisés les codes les plus longs. Si on prend comme exemple la langue française, avec comme symboles les lettres de l'alphabet, on coderait la lettre la plus utilisée (le '*e*') avec le code le plus court, alors que la lettre la moins utilisée (le '*w*' si on ne considère pas les accents) serait codée avec un code plus long. Notons qu'on travaille dans ce cas sur toute la langue française. Si on voulait être plus performant, on travaillerait avec un "dictionnaire" de Huffman propre à un texte. Le JPEG exploite cette remarque, les codes de Huffman utilisés sont propres à chaque frame JPEG.

Ces codes sont dits *préfixes* car par construction aucun code de symbole, considéré ici comme une suite de bits, n'est le préfixe d'un autre symbole. Autrement dit, si on trouve une certaine séquence de bits dans

3. l'utilisation du YUV vient de la télévision. La luminance seule permet d'obtenir une image en niveau de gris, le codage YUV permet donc d'avoir une image en noir et blanc ou en couleurs, en utilisant le même encodage. Le YCbCr est une version corrigée du YUV

un message, et que cette séquence correspond à un symbole qui lui est associé, cette séquence correspond forcément à ce symbole et ne peut pas être le début d'un autre code.

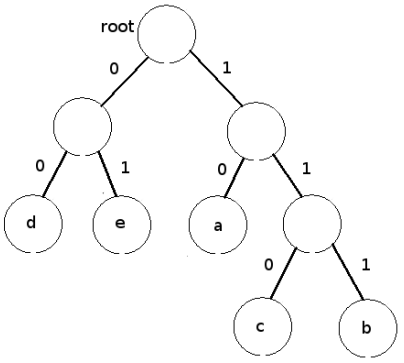
Ainsi, il n'est pas nécessaire d'avoir des « séparateurs » entre les symboles même s'ils n'ont pas tous la même taille, ce qui est ingénieux [4]. Par contre, le droit à l'erreur n'existe pas : si l'on perd un bit en route, tout le flux de données est perdu et l'on décodera n'importe quoi.

La construction des codes de Huffman n'entre pas dans le cadre initial de ce projet (pour la partie facultative "encodeur", vous pouvez utiliser des codes génériques). Par contre, il faut comprendre la représentation utilisée pour pouvoir les utiliser correctement, c'est l'objet de la suite de cette partie.

Un code de Huffman peut se représenter en utilisant un arbre binaire. Les feuilles de l'arbre représentent les symboles et à chaque noeud correspond un bit du code : à gauche, le '0', à droite, le '1'.

Le petit exemple suivant illustre ce principe :

Symbole	Code
a	10
b	111
c	110
d	00
e	01



Le décodage du bitstream `0001110100001` produit la suite de symboles `decade`. Il aurait fallut 3 bits par symbole pour distinguer 5 symboles pour un code de taille fixe (tous les codes sont alors équivalents), et donc la suite de 6 symboles aurait requis 18 bits, alors que 13 seulement sont nécessaires ici.

Cette représentation en arbre présente plusieurs avantages non négligeables, en particulier pour la recherche d'un symbole associé à un code. On remarquera que les feuilles de l'arbre représentent un code de longueur « la profondeur de la feuille ». Cette caractéristique est utilisée pour le stockage de l'arbre dans le fichier (cf. annexe A). Un autre avantage réside dans la recherche facilitée du symbole associé à un code : on parcourt l'arbre en prenant le sous arbre de gauche ou de droite en fonction du bit lu, et dès qu'on arrive à une feuille terminale, le symbole en découle immédiatement. Ce décodage n'est possible que parce que les codes de Huffman sont préfixes.

Dans le cas du JPEG, les tables de codage [5] sont fournies avec l'image. On notera que la norme requiert l'utilisation de plusieurs arbres pour compresser plus efficacement les différentes composantes de l'image. Ainsi, en mode baseline, le décodeur supporte deux tables pour les coefficients AC, et deux tables pour les coefficients DC. Les différentes tables sont caractérisées par un indice et par leur type (AC ou DC). Lors de la définition des tables (marqueur DHT) dans le fichier JPEG, l'indice et le type sont donnés. Lorsque l'on décode l'image encodée, la correspondance indice/composante (Y,Cb,Cr) est donnée au début, et permet ainsi le décodage. Attention donc à toujours utiliser le bon arbre pour la composante et le coefficient en cours de traitement.

Le format JPEG stocke les tables de Huffman d'une manière un peu particulière, pour gagner de la place. Le format de stockage est présenté plus en détail dans les annexes. Plutôt que de donner un tableau représentant les associations codes/valeurs de l'arbre pour l'image, les informations sont fournies en deux temps. D'abord, on donne le nombre de codes de chaque longueur comprise entre 1 et 16 bits. Ensuite, on donne les valeurs triées dans l'ordre des codes. Pour reconstruire la table ainsi stockée, on fonctionne donc profondeur par profondeur. Ainsi, on sait qu'il y a n_p codes de longueur p, $p = 1, \ldots, 16$. Notons que sauf pour les plus longs codes de l'arbre, on a toujours $n_p \leq 2^p - 1$. On va donc remplir l'arbre, à la profondeur 1, de gauche à droite, avec les n_1 valeurs. On remplit ensuite la profondeur 2 de la même manière, toujours de gauche à droite, et ainsi de suite pour chaque profondeur.

Pour illustrer, reprenons l'exemple précédent. On aurait le tableau suivant pour commencer :

4. Pour une fréquence d'apparition des symboles connue et un codage de chaque symbole indépendamment des autres, ces codes sont optimaux.

5. Chacune représentent un arbre.

Longueur	Nombre de codes
1	0
2	3
3	2

Ensuite, la seule information que l'on aurait serait l'ordre des valeurs :

d, e, a, c, b

soit au final la séquence suivante, qui représente complètement l'arbre :

0 3 2 d e a c b

1.5.2 Arbres DC et codage DPCM

Sauf en cas de changement brutal ou en cas de retour à la ligne, la composante DC d'une MCU (c'est à dire la moyenne de la MCU) a de grandes chances d'être proche de celle des MCU voisins. C'est pourquoi elle est codée comme la différence par rapport à celle de la MCU précédente (dit prédicateur). Pour la première MCU, on initialise le prédicateur à 0. Ce codage s'appelle DPCM (Differential Pulse Code Modulation).

La norme permet d'encoder une différence comprise entre -2047 et 2047. Si la distribution de ces valeurs était uniforme, on aurait recours à un codage sur 12 bits. Or les petites valeurs sont beaucoup plus probables que les grandes. C'est pourquoi la norme propose de classer les valeurs par ordre de magnitude, comme le montre le tableau ci-dessous.

Magnitude	valeurs possibles
0	0
1	$-1, 1$
2	$-3, -2, 2, 3$
3	$-7, \cdots, -4, 4, \cdots, 7$
$\vdots$	$\vdots$
11	$-2047, \cdots, -1024, 1024, \cdots, 2047$

TABLE 1.1 – Classes de magnitude de la composante DC

De la sorte, on n'a besoin que de n bits pour coder une valeur de magnitude n. Ce codage est effectué dans l'ordre croissant au sein d'une ligne du tableau. Par exemple, on codera -3 avec la séquence 00 (car c'est le premier élément de la ligne), -2 avec 01 et 7 avec 111.

Ces séquences permettent de trouver la valeur différentielle au sein d'une classe de magnitude. Et pour identifier la classe de magnitude, un codage de Huffman est utilisé.

Ainsi, une différence sera codée par le code associé à sa classe de magnitude dans la table de Huffman, suivi d'un nombre de bits variable correspondant au codage associé à sa valeur.

1.5.3 Arbres AC et codage RLE

Les algorithmes de type *run length encoding* ou RLE permettent de compresser sans perte en exploitant les répétitions successives de symboles au sein du flux. Par exemple, on pourrait coder la séquence (`bbbbce`) comme suit : `4bce`.

Dans le cas du JPEG, le symbole qui revient souvent dans une image est le 0. L'utilisation du zigzag (cf 1.6) dans le JPEG permet de ranger les coefficients des fréquences en créant de longues séquences de 0 à la fin. L'application d'un algorithme de type RLE permet donc de compresser encore plus l'image.

L'algorithme utilisé code uniquement les composantes AC non nulles. Le symbole codé sur un octet est composé de deux champs. Les 4 bits de la partie haute représentent le nombre de composantes AC à zéro entre la composante non nulle actuelle et la précédente. Les 4 bits de la partie basse représentent l'ordre de magnitude de la composante actuelle et peut prendre des valeurs entre 1 et 10 puisque le zéro n'est pas codé et que la norme prévoie des valeurs entre -1023 et 1023.

Néanmoins, le codage admet deux exceptions pour gérer les gros paquets de zéros. A ce stade, le codeur RLE ne peut sauter que 15 composantes AC nulles. Pour faire mieux, le codeur utilise le symbole ZRL (codé

0xF0) pour désigner un saut de 16 composantes AC nulles ainsi que le symbole EOB (End Of Block, codé 0x00) pour signaler que toutes les composantes AC à suivre dans la MCU sont nulles.

Les 162 symboles issues du RLE sont ensuite codés en utilisant un arbre de Huffman AC. Comme pour la compression de la composante DC, le flux compressé ajoute après chaque symbole de Huffman un nombre de bits variable contenant le codage associé à la composante AC non nulle dans sa classe de magnitude.

Pour ce qui est du décodage, il faut évidemment faire l'inverse et donc étendre les données compressées par les algorithmes RLE et de Huffman pour obtenir une MCU sous forme d'un vecteur de 64 entiers représentant des fréquences.

1.6 Zigzag inverse et Quantification inverse

Le vecteur MCU obtenu après décompression doit être réorganisé par l'opération zigzag inverse qui recopie les données aux coordonnées d'entrée linéaires dans un tableau de 8×8 aux coordonnées fournies par le zigzag de la figure 1.2. Ceci permet d'obtenir à nouveau une matrice 8×8.

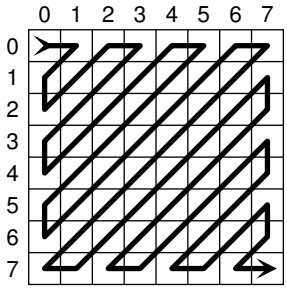


FIGURE 1.2 – Réordonnancement Zigzag.

La quantification inverse consiste à multiplier élément par élément le MCU fréquentiel et la table de quantification (de taille 8×8). Cette dernière est définie dans le bitstream au sein d'une section DQT.

1.7 Transformée en cosinus discrète inverse

Cette étape consiste à transformer les informations fréquentielles en informations spatiales. C'est une formule mathématique « classique » de transformée. Dans sa généralité, la formule de la transformée en cosinus discrète inverse pour les macroblocs de taille $n \times n$ est :

$$S(x,y) = \frac{1}{\sqrt{2n}} \sum_{\lambda=0}^{n-1} \sum_{\mu=0}^{n-1} C(\lambda)C(\mu) \cos\left(\frac{(2x+1)\lambda\pi}{2n}\right) \cos\left(\frac{(2y+1)\mu\pi}{2n}\right) \times \Phi(\lambda,\mu).$$

Dans cette formule, S est le macrobloc spatial et Φ le macrobloc fréquentiel. Les variables x et y sont les coordonnées des pixels dans le domaine spatial et les variables λ et μ sont les coordonnées des fréquences dans le domaine fréquentiel. Finalement, le coefficient C est tel que $C(\xi) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } \xi = 0, \\ 1 & \text{sinon.} \end{cases}$

Dans le cas qui nous concerne, $n = 8$ bien évidemment.

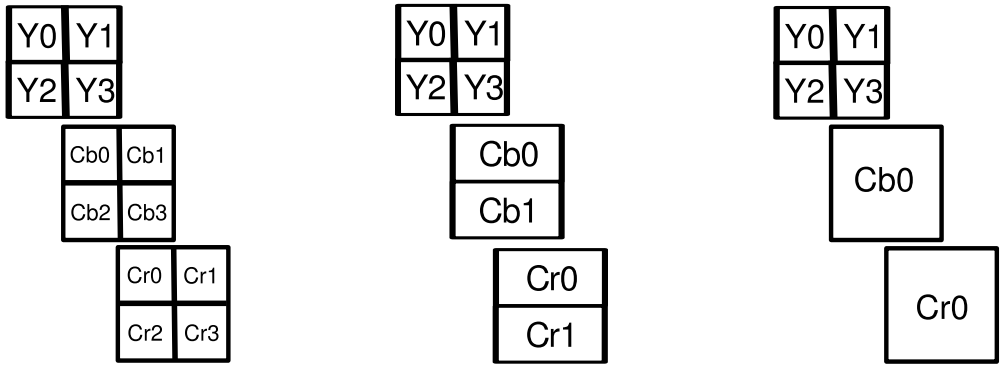
1.8 Reconstitution des MCUs

Finalement, pour terminer la décompression, il faut reconstruire les pixels à partir des informations disponibles. Le JPEG exploite la faible sensibilité de l'œil humain aux composantes de chrominance pour réaliser un sous-échantillonage (subsampling) de l'image.

Le sous-échantillonage est une technique de compression qui consiste en une diminution du nombre d'échantillons pour certaines composantes de l'image. Pour prendre un exemple, imaginons qu'on travaille sur un bloc de 2×2 pixels.

Ces 4 pixels ont chacun une valeur de chaque composante. Le stockage du bloc en YCbCr occupe donc $4 \times 3 = 12$ emplacements. On ne sous-échantillonne jamais la composante de luminance de l'image. En effet, l'œil humain est extrêmement sensible à cette information, et une modification impacterait trop la qualité perçue de l'image.

Cependant, comme on l'a dit, la chrominance contient moins d'information. On pourrait donc décider que sur ces 4 pixels, une seule valeur par composante de chrominance suffit. On obtiendrait alors un bloc stocké sur $4 + 2 \times 1 = 6$ emplacements, ce qui réduit notablement la place occupée !



(a) Non sous-échantillonné (b) Sous-échantillonné (c) Sous-échantillonné 4:2:0

FIGURE 1.3 – Composantes Y, Cb et Cr avec et sans sous-échantillonnage, pour un bloc de 4 pixels

On caractérise le sous-échantillonage par une notation de type $L : H : V$. Ces 3 valeurs ont une signification qui permet de connaître le facteur d'échantillonnage [6].

Voici les modes d'échantillonnage les plus courants qu'il vous est demandé de supporter:

4:4:4 pas de sous-échantillonnage, même nombre de blocs Y, Cb et Cr. Meilleure qualité d'image, taux de compression le plus faible ;

4:2:2 la moitié de la résolution horizontale de la chrominance est éliminée pour Cb et Cr. À l'encodage, cela se traduit par un calcul moyenné des valeurs de chrominance (même valeur Cb et Cr pour deux Y). La résolution complète est conservée verticalement. C'est un format très classique sur le Web et les caméras numériques ;

4:2:0 la moitié de la résolution horizontale et verticale de la chrominance est éliminée pour Cb et Cr. La qualité est visiblement moins bonne, mais sur un téléphone portable ou un timbre poste, c'est bien suffisant !

La diminution du nombre d'échantillons entraîne des MCU ne couvrant pas la même surface de l'image. Même s'ils représentent les mêmes pixels de l'image, les informations sur les composantes Y, Cb et Cr sont encodées dans des MCU séparées. S'il y a effectivement sous-échantillonnage, les MCU pour Cr et Cb contiennent 8×8 échantillons de chrominance, qui correspondent à un plus grand nombre de pixels de l'image. Dès lors, le nombre de blocs 8×8 par composante diffère selon le mode de sous-échantillonnage :

- pour le 4:4:4, un bloc de 8×8 pixels est nécessaire et permet de former un MCU pour chacune des composantes (Y, Cb et Cr) ;
- pour le 4:2:2, une MCU de 8×16 pixels est nécessaire et permet de former deux blocs pour Y (arrangés horizontalement $\boxed{bloc_1 \mid bloc_2}$) et un par composante de chrominance ;
- pour le 4:2:0, une MCU de 16×16 pixels est nécessaire et permet de former quatre blocs pour Y (2 blocs en horizontal, 2 en vertical $\begin{array}{|c|c|}\hline bloc_1 & bloc_2 \\ \hline bloc_3 & bloc_4 \\ \hline \end{array}$) et un par composante de chrominance ;

Ces nombres de blocs sont donnés uniquement à titre d'exemple, un 4:2:2 pourrait être contenu dans une MCU de 16×16 pixels, avec 4 blocs Y, et 2 blocs verticaux par composantes de chrominance $\begin{array}{|c|}\hline bloc_1 \\ \hline bloc_2 \\ \hline \end{array}$

Il convient alors de se poser la question de l'ordonnancement des blocs d'une image. L'image est balayée par blocs, de gauche à droite, de haut en bas, avec la taille de MCUs correspondant au sous-échantillonnage. L'ordonnancement des blocs à l'intérieur d'une MCU suit les séquences ci-dessous :

6. Pour être précis, ces valeurs représentent la fréquence d'échantillonnage, mais on ne s'en préoccupera pas ici

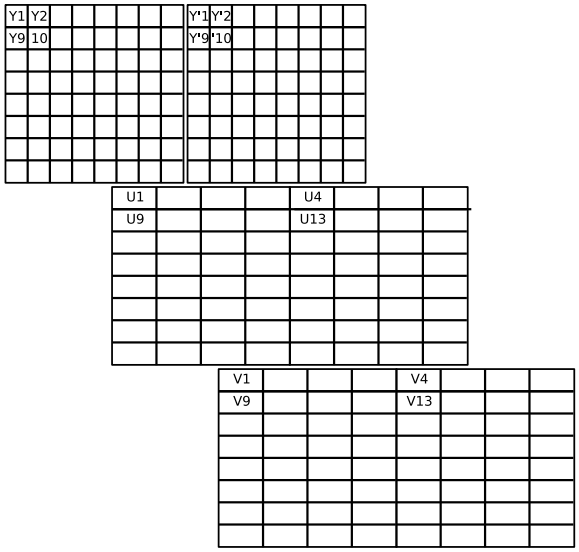


FIGURE 1.4 – Découpage des MCUs pour le sous échantillonage 4:2:2

4:4:4 Y,Cb,Cr

4:2:2 Y_1Y_2CbCr

4:2:0 $Y_1Y_2Y_3Y_4$CbCr

Au niveau du décodeur, il faut à l'inverse sur-échantillonner les blocs Cb et Cr pour qu'ils recouvrent les Y en totalité. Ceci signifie par exemple qu'en 4:2:2, la partie gauche de la chrominance couvre le premier bloc Y1, alors que la partie droite de la chrominance couvre le second bloc Y2. Il faut également fusionner les différents blocs, pour obtenir un tableau unifié et non plus plusieurs tableaux de 8×8. En effet, si on continue sur l'exemple du 4:2:2, il est nécessaire d'unifier les différents tableaux en 1 tableau de taille 16×8 par composante.

1.9 Conversion vers des pixels RGB

La conversion YCbCr vers RGB s'effectue à l'aide des 3 MCU successives (éventuellement issues d'un sur-échantillonnage). Ainsi, pour chaque pixel dans l'espace YCbCr, on effectue le calcul suivant pour obtenir le pixel RGB (donné dans la norme de JPEG).

$$\begin{aligned} R &= 1 \times Y - 0.0009267 \times (C_b - 128) + 1.4016868 \times (C_r - 128) \\ G &= 1 \times Y - 0.3436954 \times (C_b - 128) - 0.7141690 \times (C_r - 128) \\ B &= 1 \times Y + 1.7721604 \times (C_b - 128) + 0.0009902 \times (C_r - 128) \end{aligned}$$

Les formules simplifiées suivantes sont encore acceptables pour respecter les contraintes de rapport signal à bruit du standard [7].

$$\begin{aligned} R &= Y + 1.402 \times (C_r - 128) \\ G &= Y - 0.34414 \times (C_b - 128) - 0.71414 \times (C_r - 128) \\ B &= Y + 1.772 \times (C_b - 128) \end{aligned}$$

7. L'intérêt de ces formules n'est visible en terme de performance que si l'on effectue des opérations en point fixe ou lors d'implantation en matériel (elles aussi en point fixe).

Chapitre 2

Principe et base de code

2.1 Environnement logiciel et fonctionnement

2.1.1 Principe

Afin de permettre la validation de chaque étape indépendamment, nous avons préparé une version du programme de décodage découpé en de multiples morceaux.

Chaque étape implante une fonction, et est fournie sous la forme d'un objet compilé prêt à être lié à l'exécutable. Votre objectif, durant la première phase du projet sera de remplacer chacun de ces objets par vos propres modules, qui respecteront les conventions (prototype fourni dans un fichier d'entête) que nous avons édicté. Vous pourrez ainsi vérifier que votre fonction réalise effectivement la partie demandée.

Les types de données utilisés sont pour l'essentiel les types de base du langage C99. Que cela ne vous empêche pas dans vos fonctions d'utiliser des structures si cela clarifie le programme.

2.1.2 Entrées/Sorties du programme

Le programme de décodage JPEG prend en entrée un fichier au format JPEG supporté, décode l'image contenu dans ce fichier pour en faire une image non compressée sous forme de "bitmap", c'est à dire une représentation RGB de l'image, pixel par pixel.

On enregistre ensuite cette image sous la forme d'un fichier "raw" (brut). Plusieurs formats d'image bruts existent. On trouve par exemple le format "BitMaP" de Microsoft, aussi appelé BMP.

Dans le projet, nous vous proposons de travailler avec un format normalisé par Adobe, le TIFF, pour Tagged Image File Format. Ce format de fichier supporte de nombreux formats de stockage ou d'encodage de la partie image proprement dite (entre autres le JPEG, d'ailleurs), mais nous ne l'utiliserons que comme format non compressé brut.

Afin de ne pas surcharger cette section, le format TIFF est décrit en annexe B. La description proposée ne contient que les informations requises pour le format qui nous intéresse, toute personne intéressée pourra trouver gratuitement le standard TIFF sur le site web d'Adobe.

2.2 Squelette de code fourni et interfaces

Le squelette de code qui vous est fourni comprend les objets compilés des fonctions à implémenter, ainsi que les interfaces des fonctions à respecter lors de l'appel ou de l'implantation de ces fonctions. Il contient aussi la définition de certains types de données, et le code complet des fonctions permettant l'affichage de l'image décodée.

2.2.1 Présentation des types de données utilisés

• **scan_desc_t** :

```
typedef struct {
    uint8_t bit_count;
    uint8_t window;
    int32_t pred[3];
    huff_table_t *table[2][3];
} scan_desc_t;
```

Le type *scan_desc_t* permet de décrire l'image en cours de décodage. Pour chaque image décodée, il faut une structure de ce type, qui sera passée en paramètre à la fonction *unpack_block*.

La structure se compose de 4 champs. Les champs *bit_count*, *window* et *pred* doivent être initialisés à zéro. Vous pouvez bien évidemment les utiliser pour vos besoins personnels, mais ils sont utilisés et modifiés par la fonction *unpack_block*.

Le dernier champ, *table*, est un pointeur vers un tableau de tables de Huffman. Il faut deux tables par composante. *table*[0] doit correspondre aux tables pour le coefficient DC, *table*[1] sert pour les coefficients AC. L'ordre des composantes n'a pas d'importance.

• **huff_table_t** :

```
typedef struct _huff_table_t {
   int16_t code ;
   int8_t  value ;
   int8_t is_elt ;
   struct _huff_table_t *parent ;
   struct _huff_table_t *left ;
   struct _huff_table_t *right ;
} huff_table_t;
```

Le deuxième type de données qui vous est fourni représente les tables de Huffman. La représentation adoptée est l'arbre binaire.

Les éléments *parent*, *left*, *right* servent à la structure d'arbre et pointent soit vers *NULL* (l'élément est une feuille), soit vers le noeud correspondant (*left* vers le noeud fils de gauche, *right* vers le noeud fils de droite, *parent* vers le noeud père). Les autres champs servent au stockage des données. *is_elt* doit valoir 0 quand le noeud n'est pas une feuille, 1 dans le cas contraire (c'est à dire quand il est associé à une valeur). Une feuille n'a pas de fils. *value* doit contenir la valeur associée à la feuille. Elle n'a de sens que quand *is_elt* vaut 1. *code* est optionnel.

2.2.2 Les fonctions à implémenter

Les fonctions décrites ci-après sont toutes les fonctions fournies sous la forme d'objets compilés. Vous pouvez les implémenter dans l'ordre que vous voulez, une estimation de la difficulté vous est donnée dans la partie 3.1.1. Une dernière fonction, non présentée ici, est la fonction principale *main*.

• **Conversion YCbCr → RGB** :

```
extern void YCbCr_to_ARGB(uint8_t  *YCbCr_MCU[3],
                          uint32_t *RGB_MCU,
                          uint32_t nb_block_H,
                          uint32_t nb_block_V);
```

La fonction de conversion des composantes YCbCr en RGB s'applique sur les 3 composantes déjà décodées et dont les composantes Cb et Cr ont été sur-échantillonnées afin qu'elles aient la même taille que le (ou les) composante(s) Y. Elle requiert en paramètre :

– un tableau de 3 blocs, un par composante, dans l'ordre Y, Cb, puis Cr (attention à l'ordre). Les blocs sont de taille variable en fonction de l'échantillonnage. Les valeurs des composantes sont des entiers non signés sur 8 bits. Pour être précis, les cas possibles sont :

4 :4 :4 Y,Cb,Cr, menant à des MCUs de taille 8×8 ;

4 :2 :2 Y_1Y_2CbCr, menant à des MCUs de taille 16×8 ;

4 :2 :0 $Y_1Y_2Y_3Y_4$CbCr, menant à des MCUs de taille 16×16 ;

– un tableau déjà alloué ne contenant rien de significatif, qui contiendra en sortie les valeurs RGB des pixels. Une valeur RGB est codée sur 4 octets, non signés. Le premier octet n'est pas utilisé et doit être initialisé à 0, il correspond au A du ARGB. Les 3 octets suivants sont dans l'ordre : R ("red"), G ("green"), et B("blue"). L'ordre des octets, par rapport à l'endianness du processeur utilisé, est l'ordre standard des octets dans un entier. Le A est donc l'octet de poids fort de l'entier, alors que le B est l'octet de poids faible.

– le nombre de blocs dans une MCU, en horizontal (*nb_block_H*) et en vertical (*nb_block_V*).

Attention : les calculs incluent des opérations arithmétiques variées et doivent se faire avec des types signés et dont les résultats sont potentiellement hors de l'intervalle $[0 \dots 255]$. Il faut donc tronquer les valeurs de R, G et B afin qu'elles soient représentables sur un `uint8_t` (saturation).

• **Quantification inverse et Zigzag inverse** :

```
extern void iqzz_block(int32_t in[64], int32_t out[64],
                       uint8_t table[64]) ;
```

La quantification inverse et le zigzag inverse sont réalisés par une seule fonction, *iqzz_block*.

Les paramètres de la fonction *iqzz_block* représente, dans l'ordre : la MCU non décodée, la MCU décodée avec les données dans l'ordre issu du zigzag, et la table de quantification associée.

• **Calcul de la transformée en cosinus discrète inverse** :

```
extern void IDCT(int32_t *input, uint8_t *output);
```

La fonction de calcul de la transformée en cosinus discrète inverse ne prend que deux paramètres, la MCU en entrée et la MCU décodée en sortie. Les tableaux (pointeurs) passés en paramètres ont été alloués préalablement à l'appel de fonction, les pointeurs sont donc valides dans la fonction.

La version compilée fournie dans le squelette de code est une version optimisée, ne vous étonnez donc pas si les différences de performance paraissent importantes ! En revanche, la fonctionnalité doit être identique.

Outre la transformée, cette fonction passe d'une représentation signée à une représentation non signée. Il conviendra donc d'ajouter 128 à chaque coefficient obtenu.

• **Correction du sous échantillonnage** :

```
extern void upsampler(uint8_t *MCU_ds, uint8_t *MCU_us,
                      uint8_t h_factor, uint8_t v_factor,
                      uint16_t nb_blocks_H, uint16_t nb_blocks_V) ;
```

La correction du sous échantillonage permet d'étendre les blocs compressés, afin de les faire couvrir l'ensemble de la MCU qu'ils représentent.

Les paramètres de la fonction sont les suivants :

– *MCU_ds* est un tableau représentant le(s) blocs de la composante considérée, non corrigée(s). Dans le cas où il n'y a pas de sous échantillonnage dans l'image, ce paramètre est un tableau de 64 valeurs. Sinon, il contient plusieurs blocs qui sont rangés séquentiellement de gauche à droite et de haut en bas. Ainsi, la composante Y d'un sous échantillonnage 4 :2 :0 sera composée de 4 blocs. *MCU_ds* sera alors une succession de 4 tableaux de 64 valeurs ;

- *MCU_us*, après l'exécution de la fonction, contient les blocs corrigés, qui représente donc la MCU. Dans le cas précédemment cité de la composante Y dans un sous-échantillonnage 4 :2 :0, le bloc de sortie ne sera plus une suite de 4 tableaux de 8×8 pixels, mais un bloc unifié représentant un tableau de 16×16 pixels. Dans le cas où la composante est sous-échantillonnée, le résultat stocké dans *MCU_us* devra contenir un tableau unifié et étendu du bon nombre de pixels ;
- *h_factor* et *v_factor* sont les facteurs de sous-échantillonage, c'est à dire l'étalement requis des blocs en horizontal (*h_factor*) et en vertical (*v_factor*). La composante Cr d'un codage 4 :2 :2 aurait donc un *h_factor* de 2 (il faut étaler le bloc 2 fois en horizontal pour obtenir la MCU complète), et un *v_factor* de 1 ;
- les deux derniers paramètres représentent le nombre de blocs 8×8 dans une MCU (en horizontal et en vertical).

• **Construction/destruction des tables de Huffman** :

```
extern void free_huffman_tables(huff_table_t *root) ;

extern int load_huffman_table(FILE *image,
                              huff_table_t *ht) ;
```

La fonction *free_huffman_tables* sert à désallouer proprement une table de Huffman pointée par le pointeur *root*. Le "proprement" implique que les arbres fils seront eux aussi désalloués proprement. En sortie de la fonction, aucun des éléments de l'arbre ne doit rester alloué.

La fonction *load_huffman_tables* parcourt le fichier JPEG pour remplir la table de Huffman propre à l'image en cours de décodage. Le paramètre *image* est le fichier JPEG, ouvert, positionné 3 octets après le marqueur DHT concerné. Ainsi, si dans un fichier on trouve `ff c4 00 1a 00 01`, le prochain octet à lire dans *image* devra être `01`.

Attention cependant à bien prendre en compte le type des paramètres ! Le paramètre *ht* est un pointeur simple sur un *huff_table_t*, **il doit donc nécessairement être alloué avant l'appel à *load_huffman_table***. En effet, si ce n'est pas le cas, il ne sera pas possible en l'état de transmettre à la fonction appelante le pointeur alloué !

En fin de fonction, *ht* est le pointeur sur la racine de l'arbre de Huffman récupéré dans le fichier. Cet arbre est complètement initialisé, et a été rempli de gauche à droite, profondeur par profondeur.

La valeur de retour indique le nombre d'octets lus dans le fichier pendant la création de l'arbre en cas de succès, et une valeur négative en cas d'échec durant cette création.

• **Récupération d'une MCU dans le fichier** :

```
extern void unpack_block(FILE *image, scan_desc_t *scan_desc,
                         uint32_t index, int32_t T[64]) ;
```

La fonction *unpack_block* sert à récupérer des blocs 8×8 dans le fichier JPEG. *image* est le fichier JPEG en cours de lecture. *scan_desc* a déjà été présenté dans la partie réservée au type. *index* représente l'index dans les tables de Huffman de *scan_desc*, permettant de récupérer les tables associées à la composante. **Attention**, cet indice n'est pas forcément l'indice récupéré dans la section DHT, mais bien l'indice correspondant à la composante dans la structure ! A vous de gérer correctement cette structure, et de bien utiliser cette structure dans votre fonction ! *T* est le bloc en sortie.

À la fin de la fonction, *T* doit contenir un bloc 8×8 en fréquence de l'image.

Il est important de considérer les remarques qui suivent.

- Le décodage des blocs se fait bit par bit, et non pas octet par octet. Il est donc possible que le bloc de 64 pixels ne soit pas contenu sur un nombre entier d'octets, et donc que le bloc suivant, **décodé lors d'un prochain appel à la fonction**, commence au milieu d'un octet.
- Le coefficient DC d'un bloc est codé en différentiel, c'est à dire que sa valeur est calculée par rapport à la valeur du coefficient DC précédent. Il est donc nécessaire de connaître, lors de l'appel à la fonction, la valeur précédente de ce coefficient.

– Dans le flux des données brutes, il peut arriver qu'une valeur `0xff` apparaisse. Cependant, cette valeur est particulière, puisqu'elle marque le début d'une section. Afin de permettre aux décodeurs de faire un premier parcours du fichier en cherchant toutes les sections, ou de se rattraper lors que le flux est partiellement corrompu par une transmission peu fiable, la norme prévoit de distinguer ces valeurs "à décoder" des marqueurs de section. Une valeur `0xff` à décoder est donc toujours suivie de la valeur `0x00`, c'est ce qu'on appelle le *byte stuffing*. Pensez bien à ne pas décoder ce `0x00`.
– Comme précisé lors de la description de la structure *scan_desc_t*, les champs *bit_count*, *pred*, et *window* sont utilisés par notre version de *unpack_block*. Vous pouvez vous en servir si vous en voyez l'utilité, mais ce n'est pas une obligation. Cependant, si vous avez besoin d'appeler la version de référence (fournie au départ) de la fonction, n'oubliez pas d'initialiser ces champs à 0 lors du premier appel à la fonction. De la même manière, notez bien que ces champs seront modifiés lors de l'appel à la fonction, et qu'il ne faut pas les modifier entre deux appels.
– *T*, comme indiqué, doit contenir un bloc de 8×8 fréquences de l'image. Le décodage Huffman doit donc avoir été effectué, et le RLE doit avoir été décodé.

• **Évitement des segments non implantés** :

```
extern void skip_segment(FILE *image) ;
```

skip_segment est une fonction permettant de dérouler le fichier *image* pour le positionner sur le marqueur suivant, permettant ainsi en pratique de s'abstraire de l'interprétation d'un marqueur non implanté dans le décodeur, ou inutile au décodage comme par exemple un commentaire associé à l'image.

2.2.3 Écriture du résultat

• **Initialisation** :

```
extern int init_tiff_file(FILE *tiff_file,uint32_t width,
                          uint32_t length, uint32_t MCU_height) ;
```

init_tiff_file est la fonction qui permet d'initialiser le fichier de résultat. Elle écrit toutes les parties à écrire avant les données de l'image, comme par exemple l'entête du fichier TIFF, et initialise l'encapsuleur. Les paramètres de cette fonction sont :
– le fichier TIFF *tiff_file*, qui doit donc être ouvert,
– les dimensions de l'image *width* (largeur) et *length* (hauteur),
– la hauteur d'une MCU (dans le cas du JPEG, 8 ou 16, selon le sous échantillonnage).

La fonction renvoie 0 si tout se passe bien, et autre chose si une erreur a eu lieu. En particulier, elle renvoie -1 si *tiff_file* vaut *NULL*.

• **Ecriture de l'image** :

```
extern void write_tiff_file ( FILE * tifffile , uint32_t w, uint32_t *
    image ) ;
```

write_file permet d'écrire la MCU *image* dans le fichier initialisé *tiff_file*. Plus précisément, *image* est un tableau représentant une MCU décodée en ARGB (4 octets), dont l'élément A est à l'offset 0.

• **Finalisation de l'image** :

```
extern void finalize_tiff_file(FILE *tiff_file) ;
```

close_file permet de clore le fichier *tiff_file*, en nettoyant proprement le module, et en écrivant dans le fichier les informations que l'on ne peut écrire qu'à la fin, si il y en a.

Chapitre 3

Travail attendu

3.1 Partie obligatoire : décodeur

3.1.1 Étapes

Pour résumer, les étapes du décodage sont les suivantes :

1. Extraction de l'entête, récupération des facteurs de quantification et des tables de Huffman ;
2. Extraction et décompression de MCU ;
3. Multiplication par les facteurs issus des tables de quantification ;
4. Réorganisation zigzag des données des MCU ;
5. Calcul de la transformée en cosinus discrète inverse ;
6. Mise à l'échelle des composants CbCr ;
7. Conversion des YCbCr vers RGB de chaque pixel ;
8. Écriture du résultat dans le fichier TIFF (cf. 2.1.2).

Le tableau ci-dessous présente, du point de vue implantation, les différentes étapes à développer. Sans imposer d'ordre particulier, une idée de la difficulté de chaque étape, évaluée d'après nos estimations et les difficultés rencontrées par vos prédécesseurs, est précisée afin de vous aider à vous organiser. Nous ne pouvons évidemment que vous conseiller de commencer par les éléments les plus simples, afin de vous familiariser avec l'objectif, la philosophie, et l'environnement du projet.

Étapes algorithmiques :

étape	difficulté présentie
conversion YCbCr vers RGB	☆
quantification et zigzag	☆
IDCT	☆☆
upscaling	☆☆☆☆
construction de la table de Huffman	☆☆☆☆
exploitation de la table de Huffman	☆☆☆☆☆

Étapes de lecture et de manipulation :

étape	difficulté présentie
fonction d'évitement des sections	☆
détection et affichage des marqueurs	☆☆
encapsulation TIFF	☆☆☆☆
gestion complète du flux	☆☆☆☆☆

3.1.2 Minimum requis

Le travail minimum attendu dans le cadre de ce projet est le remplacement de tous les modules fournis par vos modules. Ceci signifie que toutes les fonctions décrites précédemment doivent être implantées.

Nous vous conseillons fortement de finir par les modules de gestion complète du flux (*main*) et d'encapsulation TIFF.

D'un point de vue pratique, votre projet devra être rendu sous la forme d'une archive comprenant toutes vos sources, aucun objet (sauf ceux que vous n'auriez pas eu le temps de remplacer, évidemment), et un Makefile, qui permettra, sans aucune autre action qu'un simple `make`, de compiler complètement le projet. Nous vous conseillons de considérer l'obligation du Makefile comme une aide, et pas comme une contrainte, et donc de l'utiliser tout au long du projet. En effet, le temps gagné est non négligeable, et les encadrants s'agaceront rapidement de devoir vous demander à chaque fois comment votre programme se compile si ce n'est pas fait. Et un encadrant agacé est un encadrant qui sera peu enclin à répondre à vos questions.

3.2 Extensions

Pour ceux d'entre vous qui finiront en avance, nous vous proposons plusieurs extensions du projet, à la carte :

- Nous pouvons vous conseiller d'optimiser les performances de votre programme en utilisant *gprof* par exemple. Vous serez certainement content d'apprendre qu'il existe des algorithmes optimisés pour mettre en œuvre une IDCT comme l'algorithme de Loeffler(☆☆☆). Cet algorithme, assez simple à implanter, même si complexe à comprendre, permet de réduire considérablement le temps de décodage d'une image JPEG.
- A partir du décodeur que vous avez réalisé, il est possible de réaliser "pas à pas" un encodeur JPEG. Il suffit pour cela d'utiliser un cycle de conception en V. Sachant déjà transformer un JPEG en TIFF en plusieurs étapes successives, on peut après la première étape du décodage réaliser la fonction réciproque pour l'encodage (permettant de recréer le JPEG d'entrée). De la même manière, on pourra développer et tester toutes les autres étapes d'encodage en suivant évidemment le séquencement des étapes [1]. Outre l'intérêt évident d'obtenir à la fin du cycle en V un encodeur JPEG capable de lire une image non compressée au format TIFF et de produire un JPEG au format JFIF, il faut noter qu'il sera possible en cours de ce cycle en V d'ajouter des options pour changer le JPEG de sortie en modifiant au passage certains paramètres (par exemple, réencoder l'image avec une nouvelle table de quantification).
- Le format TIFF peut également contenir une image compressée en JPEG. On peut donc envisager le support d'image JPEG dans un conteneur TIFF en entrée du décodeur.
- Si il vous reste encore du temps, sachez que la norme JPEG est encore très riche en extensions. Un exemplaire sera disponible en consultation auprès des enseignants.

3.3 Outils

En plus des outils présentés dans l'introduction générale du projet, vous pouvez utiliser :

- `display`, qui fait partie de la suite ImageMagick. C'est un utilitaire de visualisation et de conversion ou recompression des images qui gère entre autres JFIF/JPEG et TIFF. Il renvoie également quelques informations intéressantes quand le fichier est corrompu (utile pour corriger l'encapsuleur TIFF).
- `gimp`, qui est l'un des outils de traitement de l'image les plus connus. Il permet entre autre de générer des fichiers au format JFIF baseline, supporté par votre décodeur (normalement), de faire des modifications dans les fichiers, ou encore d'afficher des images JPEG ou TIFF. Pour l'affichage, préférez `display`, moins gourmand en ressources et plus rapide à invoquer, afin de ne pas trop surcharger le serveur.

1. Rien n'interdit évidemment de commencer le cycle en V par la dernière étape pour remonter à la première, mais l'intérêt d'une transformation TIFF vers TIFF est plutôt faible.

3.4 Informations

Voici quelques documents ou pages Web qui vous permettront d'aller un peu plus loin en cas de manque d'information. Au delà de la recherche d'information pour le projet, il peut aussi vous permettre d'approfondir votre compréhension du JPEG si vous êtes intéressés.

1. `http://www.impulseadventure.com/photo`
   Ce site fournit une approche par l'exemple pour qui veut construire un décodeur JPEG baseline. On y retrouve des illustrations des différentes étapes présentées dans ce document. Les détails des tables de Huffman, la gestion du sous-échantillonnage, etc, sont expliqués avec des schémas et force détail, ce qui permet de ne pas galérer sur les aspects algorithmiques.
2. Pour une compréhension plus poussée du sous-échantillonnage des chrominances, consulter `http://dougkerr.net/pumpkin/articles/Subsampling.pdf`
3. Pour les informations relatives au format JFIF, aller voir du côté de `http://www.ijg.org/`
4. Enfin, toute l'information sur la norme est évidemment disponible dans le document ISO/IEC IS 10918-1 | ITU-T Recommendation T.81. N'hésitez pas à nous demander pour le consulter (pour des raisons légales, il ne nous est pas possible de vous fournir ce document, ce qui ne gênera en rien votre progression dans le projet).

Parmi les informations que vous pourrez trouver sur le Web, il y aura du code, mais il aura du mal à rentrer dans le moule que nous vous imposons. L'examen du code lors de la soutenance sera sans pitié pour toute forme de plagiat.

Annexe A

Le format JPEG

A.1 Principe du format JPEG

Le format JPEG est un format basé sur des sections. Chaque section permet de représenter une partie du format. Afin de se repèrer dans le flux JPEG, on utilise des marqueurs, ayant la forme `0xffxx`, avec le `xx` qui permet de distinguer les marqueurs entre eux (*cf.* A.3).

Chaque section d'un flux JPEG a un role spécifique, et la plupart sont indispensables pour permettre le décodage de l'image. Nous vous donnons dans la suite de cette annexe une liste des marqueurs JPEG que vous pouvez rencontrer.

Deux marqueurs font exception dans le JPEG, les marqueurs de début (SOI, Start Of Image) et de fin (EOI, End Of Image) d'image. Ces marqueurs sont utilisés sans aucune autre information, et servent de repères.

Dans les autres cas, le format classique d'une section JPEG s'applique :

Offset	Taille (octets)	Description
`0x00`	2	Marqueur pour identifier la section
`0x02`	2	Longueur de la section, y compris les 2 octets de taille
`0x04`	X	Données associées à la section (dépend de la section)

A.2 Sections JPEG

A.2.1 Petit point sur les indices

Afin de faire les associations entre éléments, le JPEG utilise différents types d'indices. On en distingue trois :

- l'indice de composante "couleur", qu'on notera i_C,
- l'indice de table de Huffman, qui est en fait la concaténation de deux indices $(i_{AC/DC}, i_H)$,
- et l'indice de table de quantification, qu'on notera i_Q.

Une table de Huffman se repère par le type de coefficients qu'elle code, à savoir les constantes DC, ou les coefficients fréquentiels AC, et par l'indice de la table dans ce type, i_H.

Afin de pouvoir décoder chaque composante de l'image, l'entête JPEG donne les informations nécessaires pour :

- associer une table de quantification i_Q à chaque i_C,
- associer une table de Huffman $(i_{AC/DC}, i_H)$ pour chaque couple $(i_{AC/DC}, i_C)$.

A.2.2 APPx - Section Application

Cette section permet d'enregistrer des informations propres à chaque application, application signifiant ici format d'encapsulage. Dans notre cas, on ne s'intéressera qu'au marqueur APP0, qui sert pour l'encapsulage JFIF. On ne s'intéresse pas aux différentes informations dans ce marqueur. La seule chose qui nous intéresse est la séquence des 4 premiers octets de la section, qui doit contenir la phrase "JFIF".

Offset	Taille (octets)	Description
`0x00`	2	Marqueur APP0 (`0xffe0`)
`0x02`	2	Longueur en octets de la section
`0x04`	4	'J' 'F' 'I' 'F'
`0x09`	x	Données spécifiques au JFIF, non traitées

A.2.3 SOFx - Start Of Frame

Le marqueur SOF permet de marquer le début effectif d'une image, et de donner les informations générales rattachées à cette image. Il existe plusieurs marqueurs SOF, qui permettent de donner le type d'encodage JPEG utilisé. Dans le cadre de ce projet, nous ne nous intéressons qu'au JPEG "Baseline DCT (Huffman)", soit le SOF0 (`0xffc0`). Nous vous donnons pour information les autres types dans le récapitulatif des sections JPEG A.3.

Les informations générales associées à une image sont la précision des données, qui donnent le nombre de bits par coefficient, les dimensions de l'image, et le nombre de composantes couleurs utilisées. Pour pouvoir repérer ces composantes, on leur associe également l'indice i_C. Pour chaque composante, on donne les facteurs d'échantillonnage (*cf* 1.8), et la table de quantification associée à la composante (i_Q).

Dans le JFIF, l'ordre des composantes est toujours le même : Y, Cb et Cr. Toujours dans le JFIF, les indices sont normalement fixés pour ces composantes à 1, 2 et 3. Cependant, certains encodeurs ne suivent pas cette obligation d'indice. **Vous devrez donc traiter les cas où les indices ne sont pas fixés.** De manière générale, la section SOF est la seule dans laquelle vous avez la garantie que les composantes seront indiquées dans l'ordre Y, Cb, puis Cr. Ensuite, seuls les indices vous permettront de connaître la composante dont on parle.

Une section SOF a donc le format suivant :

Offset	Taille (octets)	Description
`0x00`	2	Marqueur SOFx : `0xffc0` pour le SOF0
`0x02`	2	Longueur de la section
`0x04`	1	Précision en bits par composante, généralement 8 pour le baseline
`0x05`	2	Hauteur en pixels de l'image
`0x07`	2	Largeur en pixels de l'image
`0x09`	1	Nombre de composantes N (Ex : 3 pour le YCbCr, 1 pour les niveaux de gris)
`0x0a`	3N	N fois : - 1 octet : Indice de composante i_C - 4 bits : Facteur d'échantillonage horizontal - 4 bits : Facteur d'échantillonage vertical - 1 octet : Table de quantification i_Q associée

A.2.4 DQT - Define Quantization Table

Cette section permet de définir une table de quantification. Il y a généralement plusieurs tables de quantification dans un fichier JPEG (de manière générale, deux). Ces tables sont repérées à l'aide de l'indice i_Q défini plus haut. C'est ce même indice, défini dans une section DQT, qui est utilisé dans la section SOF pour l'association avec une composante.

Offset	Taille (octets)	Description
`0x00`	2	Marqueur DQT (0xFFDB)
`0x02`	2	Longueur en octets de la section
`0x04`	4 bits 4 bits	précision (0 : 8 bits, 1 : 16 bits) Indice i_Q de la table de quantification
`0x05`	64	Valeurs de la table de quantification, **stockées au format zigzag**

A.2.5 DHT - Define Huffman Tables

La section DHT permet de définir une (ou plusieurs) table(s) de Huffman, avec leurs indices. On ne revient pas sur la représentation des tables dans le format JPEG, qui a été longuement expliquée en 1.5.1. En plus de la table, la section DHT définit les méthodes de repérage de la table, à savoir les indices $i_{AC/DC}$ et i_H.

Dans le cas où plusieurs tables sont définies dans une unique section DHT, il y a en fait répétition des 3 dernières cases du tableau suivant. La taille de la section représente la taille nécessaires pour stocker toutes les tables (d'où l'importance de la valeur de retour de la fonction `load_huffman_table`).

Dans un fichier, il ne peut pas y avoir plus de 4 tables de Huffman par type AC ou DC définies (sinon, le flus JPEG est corrompu).

Offset	Taille (octets)	Description
`0x00`	2	Marqueur DHT (FFC4)
`0x02`	2	Longueur en octets de la section
`0x04`	1	Information sur la table de Huffman : bit 0..3 : indice (0..3, ou erreur) bit 4 : type (0=DC, 1=AC) bit 5..7 : non utilisé, doit valoir 0 (sinon erreur)
`0x05`	16	Nombres de symboles avec des codes de longueur 1 à 16 La somme de ces valeurs représente le nombre total de codes et doit être inférieure à 256
`0x15`	x	Table contenant les symboles, triés par longueur (*cf* 1.5.1)

A.2.6 SOS - Start Of Scan

La section SOS marque le début du décodage effectif du flux JPEG, c'est-à-dire le début des données brutes de l'image JPEG. Elle contient les associations des composantes et des tables de Huffman, ainsi que des informations d'approximation et de sélection qui ne sont pas utilisées dans le cadre de ce projet. Elle donne également l'ordre dans lequel sont exprimées les composantes (comme précisé précédemment).

Les données brutes sont ensuite données par bloc de 8×8, dans l'ordre des composantes indiqué dans la section, et avec suffisament de blocs pour chaque composante pour reconstruire une MCU (en fonction du sous échantillonage). La taille de la MCU se déduit de la section SOF. Ainsi, si on a une MCU de 64 pixels, et un ordre classique des composantes, on lira d'abord un bloc pour Y, puis un bloc pour Cb, puis enfin un bloc pour Cr. Si on a un sous-échantillonage vertical de type 422, et une MCU de 128 pixels (16 de large, 8 de haut), dans l'ordre classique des composantes, on lira 2 blocs pour Y, 1 bloc pour Cb et 1 bloc pour Cr.

Offset	Taille (octets)	Description
`0x00`	2	Marqueur SOS (FFDA)
`0x02`	2	Longueur en octets de la section (données brutes non comprises)
`0x04`	1	N = Nombre de composantes La longueur de la section vaut 2N + 6
`0x05`	2N	N fois : 1 octet : Indice du composant i_C 4 bits : Indice de la table de Huffman (i_H) pour les coefficients DC ($i_{AC/DC} = DC$) 4 bits : Indice de la table de Huffman (i_H) pour les coefficients AC ($i_{AC/DC} = AC$)
...	1	Ss : Début de la sélection (non utilisé)
...	1	Se : Fin de la sélection (non utilisé)
...	1	Ah : 4 bits, Approximation successive, poids fort Al : 4 bits, Approximation successive, poids faible

A.2.7 COM - Commentaire

Afin de rajouter des informations textuelles supplémentaires, il est possible d'ajouter des commentaires dans le fichier. On remarquera que celà nuit à l'objectif de compression (les commentaires sont finalement des informations inutiles, généralement le nom de l'encodeur, par exemple).

Offset	Tallle (octets)	Description
0x00	2	Marqueur COM (0xFFFE)
0x02	2	Longueur de la section
0x04	x	Données

A.3 Récapitulatif

Code	ID	Description
0x00		Byte stuffing (ce n'est pas un marqueur !)
0x01	TEM	
0x02 ... 0xbf	Réservés (not used)	
0xc0	SOF0	Baseline DCT (Huffman)
0xc1	SOF1	DCT séquentielle etendue (Huffman)
0xc2	SOF2	DCT Progressive (Huffman)
0xc3	SOF3	DCT spatiale sans perte (Huffman)
0xc4	DHT	Define Huffman Tables
0xc5	SOF5	DCT séquentielle différentielle (Huffman)
0xc6	SOF6	DCT séquentielle progressive (Huffman)
0xc7	SOF7	DCT différentielle spatiale(Huffman)
0xc8	JPG	Réservé pour les extensions du JPG
0xc9	SOF9	DCT séquentielle étendue (arithmétique)
0xca	SOF10	DCT progressive (arithmétique)
0xcb	SOF11	DCT spatiale (sans perte) (arithmétique)
0xcc	DAC	Information de conditionnement arithmétique
0xcd	SOF13	DCT Séquentielle Différentielle(arithmétique)
0xce	SOF14	DCT Différentielle Progressive(arithmétique)
0xcf	SOF15	Progressive sans pertes (arithmétique)
0xd0 ... 0xd7	RST0 ... RST7	Restart Interval Termination
0xd8	SOI	Start Of Image (Début de flux)
0xd9	EOI	End Of Image (Fin du flux)
0xda	SOS	Start Of Scan (Début de l'image compressée)
0xdb	DQT	Define Quantization tables
0xdc	DNL	
0xdd	DRI	Define Restart Interval
0xde	DHP	
0xdf	EXP	
0xe0 ... 0xef	APP0 ... APP15	Marqueur d'application
0xf0 ... 0xfd	JPG0 ... JPG13	
0xfe	COM	Commentaire

Annexe B

Le format TIFF

B.1 Principe du format

Le format TIFF est un format basé sur des champs, chaque champ étant constitué d'un *tag* et de valeurs. Les *tags*, prédéfinis, permettent de donner de la signification aux valeurs qui suivent.

Un fichier TIFF est structuré à l'aide de "répertoires", appelés Image File Directory (IFD), qui servent à représenter une image. Plus précisément, le fichier TIFF est structuré comme une entête qui indique le format du fichier courant, et qui se termine par un pointeur sur le premier IFD (le pointeur est un offset dans le fichier). A la fin de chaque IFD, un pointeur vers l'IFD suivant est donné. Pour le dernier IFD du fichier, ce pointeur vaut 0. On se retrouve donc avec une liste chaînée d'IFD, chacun de ces répertoires permettant de représenter une image.

B.2 Entête du fichier

L'entête d'un fichier TIFF est constituée de 8 octets.

Offset	Size(bytes)	Description
`0x00`	2	Endianness du fichier ("II",0x4949 = LE, "MM",0x4D4D = BE)
`0x02`	2	Identification du TIFF (doit valoir 42)
`0x04`	4	Pointeur (Offset en nombre d'octets) sur le premier IFD

L'endianness permet de donner l'ordre des écritures des octets pour les valeurs de taille supérieure à 1 octet. Si la valeur est "II", l'écriture est Little Endian, et dans le fichier, le premier octet écrit est donc l'octet de poids faible. Si la valeur est "MM", l'écriture est Big Endian, et dans le fichier, le premier octet sera l'octet de poids fort. Vous êtes libres de choisir l'implantation qui vous convient le mieux concernant l'endianness.

L'identification du TIFF permet de vérifier qu'il s'agit bien d'un fichier TIFF. Il s'agit d'un code, valeur constante partagée entre tous les fichiers de type TIFF. Cette valeur est extrêmement bien choisie, puisqu'il s'agit de 42.

Le dernier champ permet de trouver l'emplacement du premier descripteur d'image, à l'aide d'un offset. Les offsets TIFF sont toujours calculés à partir du premier octet du fichier. Attention, l'IFD peut se trouver n'importe où dans le fichier (même après les données qu'il décrit), mais se trouve forcément à la limite d'un mot de 4 octets.

B.3 Image File Directory

Un IFD est composé de une ou plusieurs entrées de 12 octets chacune.

Offset	Size(bytes)	Description
`0x00`	2	Nombre d'entrées N
`0x02`	12N	N entrées IFD
`0x??`	4	Pointeur sur l'IFD suivant, (0x00000000 si dernier IFD)

Les entrées IFD sont des associations tag/valeurs (d'où le nom du format), qui permettent soit de définir des paramètres de l'image (hauteur, largeur, ...), soit de stocker l'image elle-même. Une entrée est une séquence de 12 octets avec le format suivant :

Offset	Size(bytes)	Description
`0x00`	2	Tag d'identification
`0x02`	2	Type de données
`0x04`	4	Nombre de valeurs associées à l'entrée
`0x08`	4	Pointeur (offset) sur les valeurs associées à l'entrée ou valeur directement.

Dans le cas où les données associées à l'entrée courante tiennent dans un espace de 4 octets (toutes les données avec 1 ou 2 SHORT, ou les données avec un seul LONG), le pointeur est remplacé par la valeur directement. Ce remplacement n'est pas optionnel (on ne peut pas choisir entre offset ou valeur)

Les entrées d'un IFD doivent être triées par ordre croissant de tag.

B.3.1 Types de données

Dans notre version limitée du TIFF, il existe 5 types de données possible.

Type	Valeur	Taille(bytes)	Description
BYTE	1	1	Représente un octet
ASCII	2	1	Représente un caractère sur 8 bits. Le dernier octet d'une chaîne composée de N ce ces caractères ASCII doit toujours être 0
SHORT	3	2	Représente un entier sur 16 bits
LONG	4	4	Représente un entier sur 32 bits
RATIONAL	5	8	Représente un nombre rationnel, en deux entiers. Le premier est le numérateur, le deuxième est le dénominateur

B.4 Tags Utiles

Il existe de nombreux tags définis dans la spécification du TIFF. Dans le cadre de ce projet, vous n'aurez besoin que des tags définis dans le tableau qui suit, et qui permettent de représenter une image couleur RGB **sans le A**.

B.4.1 Caractéristiques de l'image

Tout d'abord, voici une liste de tags permettant de définir les caractéristiques de l'image.

ImageWidth et *ImageLength* permettent de donner les dimensions en pixels de l'image, respectivement la largeur et la hauteur. Ces dimensions sont données soit sous la forme de SHORT, soit sous la forme de LONG.

3 autres champs doivent être renseignés dans une image, mais nous ne les utilisons pas. Ce sont les champs de résolution, c'est à dire le nombre de pixels par unité de mesure dans l'image.

XResolution et *YResolution* permettent de donner cette résolution, alors que *ResolutionUnit* permet de donner l'unité utilisée. Ces deux valeurs permettent de calculer les dimensions de l'image, cependant, elles n'influent pas lors de la visualisation "numérique" des images. Dans le cadre du projet, nous vous proposons de choisir la valeur 100 pixels par centimètre.

B.4.2 Information de stockage

Dans un fichier TIFF, les images sont découpées en bandes horizontales, qui s'étalent sur toute la largeur de l'image. La valeur associée au tag *RowsPerStrip* donne la hauteur de ces bandes en pixel. Pour chacune de ces bandes, on enregistre, ligne par ligne, les pixels sous la forme de 3 composantes R, G, et B. La taille de ces composantes en bits est donnée par le tag *BitsPerSample*. *SamplesPerPixels* permet de donner le nombre de composantes associé à un pixel, dans notre cas il vaudra donc 3.

Les données permettant de coder les bandes sont stockées aux offsets indiqués par le tag *StripOffsets*, et la taille de ces données est donnée par *StripByteCounts*.

B.4.3 Divers

Afin de pouvoir créer un fichier "vide", il est possible d'utiliser le tag Software, qui permet de préciser l'application ayant servi à créer l'application.

B.4.4 Récapitulatif

Le tableau ci-après récapitule les différents tags. Dans le cas d'une image RGB non compressée, tous les champs sont requis sauf le champ Software.

Nom	Code	Type	Valeur	Description
ImageWidth	0x100	SHORT/LONG		Nombre de colonnes de l'image
ImageLength	0x101	SHORT/LONG		Nombre de lignes de l'image
BitsPerSample	0x102	SHORT	8,8,8	Taille d'une composante couleur
Compression	0x103	SHORT	1	Pas de compression
			2	Huffman (inutilisé)
			32773	PackBit (inutilisé)
PhotometricInterpretation	0x106	SHORT	2	Indique une image RGB
StripOffsets	0x111	SHORT/LONG		Offsets des différentes lignes de l'image
SamplesPerPixel	0x115	SHORT	3 +	Nombre de composantes par pixels
RowsPerStrip	0x116	SHORT/LONG		Hauteur (en pixels) des lignes TIFF
StripByteCounts	0x117	SHORT/LONG		Taille en octets des différentes lignes
XResolution	0x11a	RATIONAL		Nombre de pixels par unité de mesure (horizontal)
YResolution	0x11b	RATIONAL		Nombre de pixels par unité de mesure (vertical)
ResolutionUnit	0x128	SHORT	1	Pas d'unité de mesure
			2	Centimètres
			3	Pouces
Software	0x131	ASCII		Chaine de caractères pour indiquer l'application

B.5 Exemple

Afin de mieux illustrer la forme d'un fichier TIFF voici le contenu d'une image au format TIFF non compressé représentant un rectangle rouge, de 8 pixels de large et 16 pixels de haut, enregistrée sous forme de lignes de 8 pixels de haut, en big endian (pour plus de lisibilité).

Addresse	Info	Valeur
0x000	header	0x4D4D 0x002a 0x00000008
0x008	IFD	0x000c
0x00a		0x0100 0x0004 0x00000001 0x00000008
0x016		0x0101 0x0004 0x00000001 0x00000010
0x022		0x0102 0x0003 0x00000003 0x0000009e
0x02e		0x0103 0x0003 0x00000001 0x00010000
0x03a		0x0106 0x0003 0x00000001 0x00020000
0x046		0x0111 0x0003 0x00000002 0x00b40174
0x052		0x0115 0x0004 0x00000001 0x00000003
0x05e		0x0116 0x0004 0x00000001 0x00000008
0x06a		0x0117 0x0003 0x00000002 0x00c000c0
0x076		0x011a 0x0005 0x00000001 0x000000a4
0x082		0x011b 0x0005 0x00000001 0x000000ac
0x08e		0x0128 0x0003 0x00000001 0x00020000
0x09a	Offset suivant	0x00000000
0x09e	BitsPerSamples	0x0008 0x0008 0x0008
0x0a4	XResolution	0x00000064 0x00000001
0x0ac	YResolution	0x00000064 0x00000001
0x0b4	Ligne 1	0xff0000 0xff0000 0xff0000 ... (64 valeurs)
0x174	Ligne 2	0xff0000 0xff0000 0xff0000 ... (64 valeurs)