Grenoble INP ENSIMAG

Ensimag — Printemps 2015

—

Projet Logiciel en C

—

Sujet : Conversion d'images JPEG

Auteurs : Des enseignants actuels et antérieurs du projet C

Table des matières

Chapitre 1

Objectif

L'objectif de ce projet est de réaliser en langage C un convertisseur d'image au format JPEG, 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 répandus 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. [1]

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 qualité de 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 iden-

1. Donc votre projet C est en fait un « décodeur ISO/IEC IS 10918-1 | ITU-T Recommendation T.81 ». Qu'on se le dise !

tique 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 général du codec JPEG

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 macroblocs ou MCU pour *Minimum Coded Unit*. La plupart du temps, et en tout cas dans ce projet avant d'éventuelles extensions, les MCU sont de taille 8×8, 16×8, 8×16 ou 16×16 pixels selon le facteur d'échantillonnage (voir section 1.8). Chaque MCU est ensuite réorganisée en un ou plusieurs blocs de taille 8×8 pixels.

La suite porte sur la compression/décompression d'un bloc 8×8. Tout d'abord, chaque bloc est traduit dans le domaine fréquentiel par transformation en cosinus discrète (DCT). Le résultat de ce traitement, appelé bloc fréquentiel, est encore un bloc 8×8 mais dont les coordonnées sont des fréquences et non plus des pixels. On y distingue une composante continue *DC* aux coordonnées $(0, 0)$ et 63 composantes fréquentielles *AC*.[2] 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 bloc par bloc à l'aide d'un filtre de quantification spécifique à chaque image. Le bloc fréquentiel filtré est ensuite parcouru en zigzag (ZZ) afin de transformer le bloc en un vecteur de 1×64 fréquences avec les hautes fréquences en fin. De la sorte, on obtient statistiquement plus de 0 en fin de vecteur.

Ce bloc vectorisé est alors compressé en utilisant successivement plusieurs codages sans perte : d'abord un codage RLE pour exploiter les répétitions de 0, un codage des différences plutôt des valeurs, puis un codage entropique[3] dit de *Huffman* qui utilise un dictionnaire spécifique à l'image en cours de traitement.

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

Les étapes ci-dessus sont appliquées à tous les blocs composant les MCU de l'image. La concaténation de ces vecteurs compressés forment un flux de bits (*bitstream*) qui est stocké dans le fichier JPEG. Ces données brutes sont séparées par des marqueurs qui précisent la longueur et le contenu des données associées. Le format et les marqueurs sont spécifiés dans l'annexe B.

Le décodeur qui est le but de ce projet effectue les opérations dans l'ordre inverse. Les opérations de codage/décoge sont résumées figure 1.1, puis détaillées dans les sections suivantes (dans le sens du décodage). L'annexe A fournit elle un exemple numérique du codage d'une MCU.

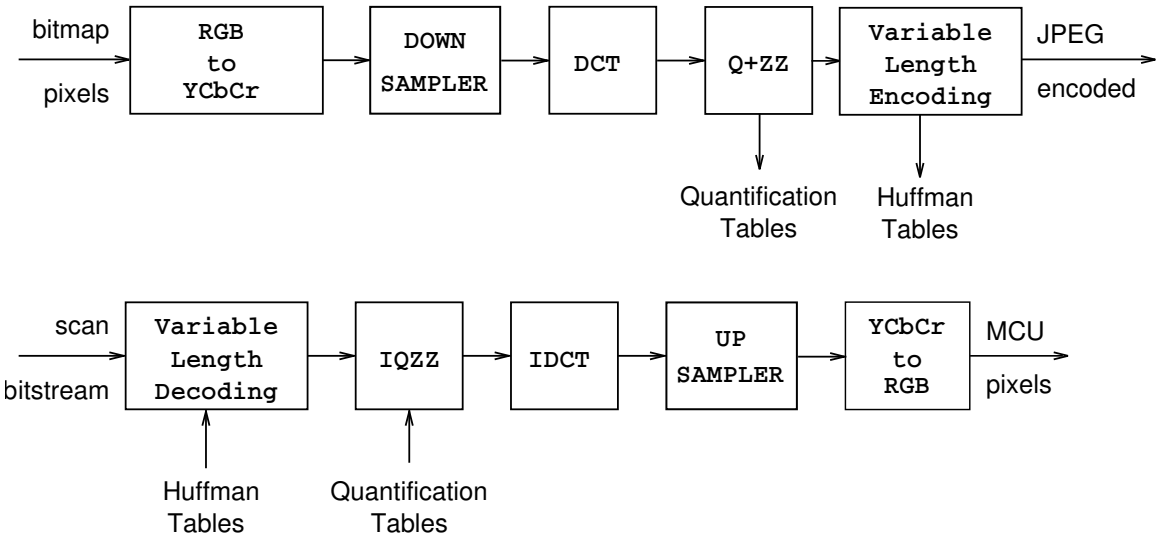


FIGURE 1.1 – Principe du codec JPEG : opérations de codage (en haut) et de décodage (en bas).

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 trois couleurs : le rouge (R), le vert (G), et le bleu (B). Une information de transparence *alpha* (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 trois 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 par la norme JPEG. Une confusion est souvent réalisée entre le format YCbCr et le format YUV. [4]

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.

4. 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.

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. Elles peuvent nécessiter un traitement particulier avant d'être utilisable ;

— représentation de l'image : ce sont les données brutes qui contiennent l'image encodée.

Une liste exhaustive des marqueurs est définie dans la norme JFIF. Les principaux vous sont donnés en annexe B de ce document, avec la représentation des données utilisées 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,

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-échantillonnage utilisés.

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

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

1.5 Décompression d'un bloc fréquentiel

Les blocs fréquentiels sont compressés sans perte dans le *bitstream* JPEG par l'utilisation de plusieurs techniques successives. Tout d'abord, les répétitions de 0 sont exploitées par un codage de type *RLE* (voir 1.5.3), puis les valeurs non nulles sont codées comme *différence* par rapport aux valeurs précédentes (voir 1.5.2), enfin, les symboles obtenus par l'application des deux codages précédents sont codés par un codage entropique de Huffman (1.5.1). Nous présentons dans cette sections les trois codages dans l'ordre qui nous intéresse pour ce projet : celui de la décompression.

1.5.1 Le codage de Huffman

Les codes de Huffman sont appelés codes *préfixés*. 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 *préfixés* 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 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.

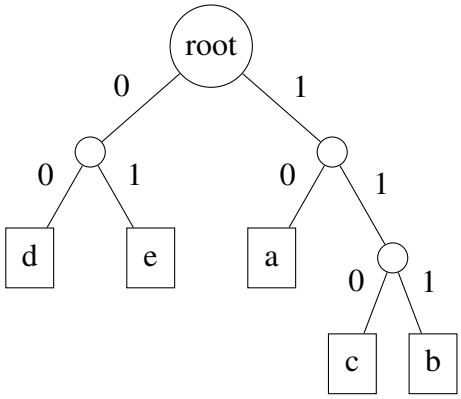
Ainsi, il n'est pas nécessaire d'avoir des « séparateurs » entre les symboles même s'ils n'ont pas tous la même taille, ce qui est ingénieux. [5] Par contre, le droit à l'erreur n'existe pas : si l'on perd un bit en route, tout le flux de données est perdu et l'on décodera n'importe quoi.

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 [6]). 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 (voir ci-dessous). 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 préfixés.

Dans le cas du JPEG, les tables de codage [7] 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 quatre tables :

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

6. ... ou vous souvenir de certains de vos pas si anciens TP ?

7. Chacune représentant un arbre de Huffman.

— deux tables pour la luminance Y, une pour les composantes DC et une pour les composantes AD ;
— deux tables communes aux deux chrominances Cb et Cr, une DC et une AC.

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. 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 :

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⟩

Dans le cas du JPEG, les table de Huffman permettent de coder (et décoder) des symboles pour reconstruire les composantes DC et les coefficients AC d'un bloc fréquentiel.

1.5.2 Composante continue : DPCM, magnitude et arbre DC

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

Représentation par magnitude 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.

Une valeur dans une classe de magnitude m donnée est retrouvée par son "indice", codé sur m bits. Ces indices sont définis par ordre croissant au sein d'une ligne du tableau. Par exemple, on codera -3 avec la séquence de bits `00` (car c'est le premier élément de la ligne), -2 avec `01` et 7 avec `111`.

De la sorte, on n'a besoin que de $4 + m$ bits pour coder une valeur de magnitude m : 4 bits pour la classe de magnitude et m pour l'indice dans cette classe. S'il y a en moyenne plus de magnitudes inférieures à 8 ($4 + m = 12$ bits), on gagne en place.

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

TABLE 1.1 – Classes de magnitude de la composante DC (pour AC, la classe max est la 10 et la magnitude 0 n'est jamais utilisée).

Encodage dans le flux de bits : Huffman Les classes de magnitude ne sont pas encodées directement dans le flux binaire. Au contraire un arbre de Huffman DC est utilisé afin de minimiser la longueur (en nombre de bits) des valeurs les plus courantes. C'est donc le chemin (suite de bits) menant à la feuille portant la classe considérée qui est encodé.

Ainsi, le *bitstream* au niveau d'un début de bloc contient un symbole de Huffman à décoder donnant une classe de magnitude m, puis une séquence de m bits qui est l'indice dans cette classe.

Décodage Pour ce qui est du décodage, il faut évidemment faire l'inverse : décoder la magnitude à partir des bits lus dans le flux et de l'arbre DC (qui a été préalablement lu dans le fichier JFIF). Ensuite, la valeur DC différentielle est retrouvée en lisant les bits d'indice dans la classe de magnitude.

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. Par exemple, la séquence `000b0eeeeed` pourrait être codée `30b05ed`. Dans le cas du JPEG, le symbole qui revient souvent dans une image est le 0. L'utilisation du zigzag (voir 1.6) permet de ranger les coefficients des fréquences en créant de longues séquences de 0 à la fin, qui se prêtent parfaitement à une compression de type RLE.

Codage des coefficients AC Chacun des 63 coefficients AC non nul est codé par un symbole sur un octet suivi d'un nombre variable de bits.

Le symbole est composé de 4 bits de poids fort qui indiquent le nombre de coefficients zéro qui précédent le coefficient actuel et 4 bits de poids faibles qui codent la classe de magnitude du coefficient, de la même manière que pour la composante DC (voir 1.5.2). Il est à noter que les 4 bits de la partie basse peuvent prendre des valeurs entre 1 et 10 puisque le zéro n'a pas besoin d'être codé et que la norme prévoie des valeurs entre -1023 et 1023 uniquement.

Ce codage permet de sauter au maximum 15 coefficients AC nuls. Pour aller plus loin, des symboles particuliers sont en plus utilisés :

— code ZRL : `0xF0` désigne un saut de 16 composantes nulles (et ne code pas de composante non nulle)

— code EOB : `0x00` (*End Of Block*) signale que toutes les composantes AC restantes du bloc sont nulles.

Ainsi, un saut de 21 composantes nulles serait codé par (`0xF0, 0x5?`) où le " ?" est la classe de magnitude de la prochaine composante non nulle. La table ci-après récapitule les symboles RLE possibles.

Symbole RLE	Signification
`0x00`	*End Of Block*
`0xF0`	16 composantes nulles
`0x?0`	symbole invalide (interdit !)
`0x`$\alpha\gamma$	α composantes nulles, puis composante non nulle de magnitude γ

Pour chaque coefficient non nul, le symbole RLE est ensuite suivi d'une séquence de bits correspondant à l'indice du coefficient dans sa classe de magnitude. Le nombre de bits est la magnitude du coefficient, comprise entre 1 et 10.

Encodage dans le flux Les symboles RLE sur un octet (162 possibles) ne sont pas directement encodés dans le flux, mais là encore un codage de Huffman est utilisé pour miniser la taille symboles les plus courants. On trouve donc finalement dans le flux (*bitstream*), après le codage de la composante DC, une alternance de symboles de Huffman (à décoder en symboles RLE) et d'indices de magnitude.

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

1.6 Zigzag inverse et Quantification inverse

Réordonnancement Zigzag Le vecteur 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.

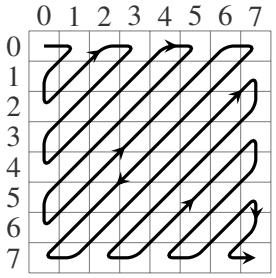


FIGURE 1.2 – Réordonnancement Zigzag.

Quantification inverse Au codage, la quantification consiste à diviser (terme à terme) chaque bloc 8×8 par une matrice de quantification, elle aussi de taille 8×8. Les résultats sont arrondis, de sorte que plusieurs coefficients initialement différents ont la même valeur après quantification. De plus de nombreux coefficients sont ramenés à 0, essentiellement dans les hautes fréquences auxquelles l'œil humain est peu sensible.

Au décodage, la quantification inverse consiste à multiplier élément par élément le bloc fréquentiel par la table de quantification. Moyennant la perte d'information, les blocs fréquentiels initiaux seront ainsi reconstruits. Remarque : la matrice de quantification est stockée dans le fichier au format zigzag. Attention au sens des opérations…

Deux tables de quantification sont généralement utilisées, une pour la luminance et une pour les deux chrominances. Le choix de ces tables, complexe mais fondamental quant à la qualité de la compression et au taux de perte d'information, n'est pas discuté ici. Les tables utilisées à l'encodage sont inclues dans le fichier JFIF.

La quantification est l'étape du codage qui introduit le plus de perte, mais aussi une de celles qui permet de gagner le plus de place (en réduisant l'amplitude des valeurs à encoder et en annulant de nombreux coefficients dans les blocs).

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 :

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

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 MCU

Dans le processus de compression, le JPEG peut exploiter la faible sensibilité de l'œil humain aux composantes de chrominance pour réaliser un sous-échantillonnage (*subsampling*) de l'image.

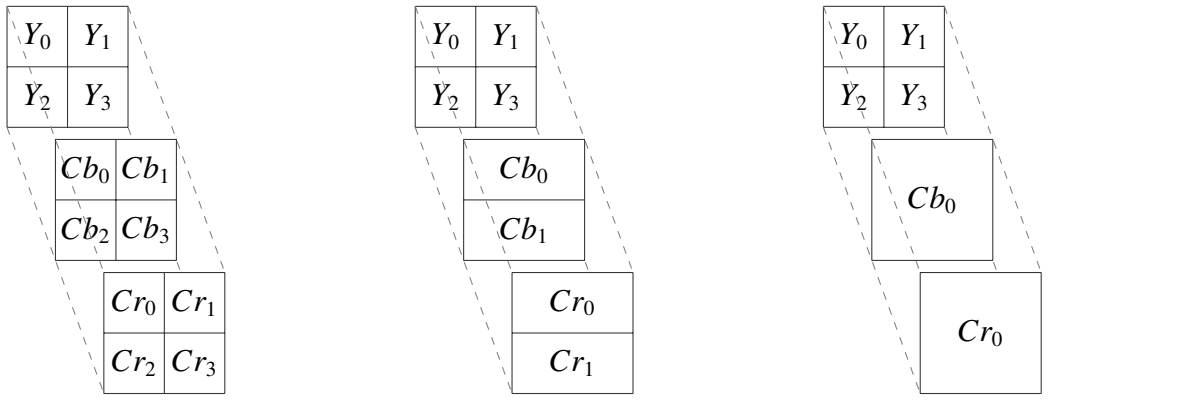
Le sous-échantillonnage est une technique de compression qui consiste en une diminution du nombre d'échantillons pour certaines composantes de l'image. Pour prendre un exemple, imaginons qu'on travaille sur une partition (MCU) de 2×2 pixels.

Ces 4 pixels ont chacun une valeur de chaque composante. Le stockage 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. Il suffirait alors de $4 + 2 \times 1 = 6$ emplacements pour représenter toutes les composantes, ce qui réduit notablement la place occupée !

On caractérise le sous-échantillonnage par une notation de type L:H:V. Ces trois valeurs ont une signification qui permet de connaître le facteur d'échantillonnage. [8]

Les modes d'échantillonnage supportés dans ce projets sont les plus courants :

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



(a) Sans sous-échantillonnage

(b) Sous-échantillonnage horizontal (4:2:2)

(c) Sous-échantillonnage horizontal et vertical (4:2:0)

FIGURE 1.3 – Composantes Y, Cb et Cr avec et sans sous-échantillonnage, pour une partition de 2×2 pixels

4:4:4 Pas de sous-échantillonnage. Le nombre de blocs est identique pour toutes les composantes Y, Cb et Cr. La qualité de l'image est optimale mais le taux de compression est le plus faible ;

4:2:2 La moitié de la résolution horizontale de la chrominance est éliminée pour Cb et Cr (figure 1.3(b)). À l'encodage, une seule valeur par chrominance Cb et Cr est utilisée pour deux pixels voisins d'une même ligne. Cette valeur est calculée sur la valeur RGB moyenne des deux pixels. 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 (figure 1.3(c)). Une seule valeur de chrominance Cb et Cr est donc utilisée pour quatre pixels. La qualité est visiblement moins bonne, mais sur un téléphone portable ou un timbre poste, c'est bien suffisant !

En fonction du sous-échantillonnage, les composantes Y, Cb, Cr d'une MCU sont codées sur un nombre de blocs 8×8 différents. Par exemple :

— pour le 4:4:4, une MCU est de taille 8×8, avec un bloc de 8×8 pixels pour chaque composante ;
— pour le 4:2:2, une MCU de 8×16 pixels est nécessaire. Elle est formée de deux blocs 8×8 pour Y arrangés horizontalement, et un par composante de chrominance ; par exemple Y_0, Y_1, Cb_0 et Cr_0 de la figure 1.3(b) forment une MCU ;
— pour le 4:2:0, une MCU de 16×16 pixels est nécessaire. Elle est formée de quatre blocs pour Y (deux blocs en horizontal, deux en vertical) et un par composante de chrominance ; par exemple Y_0, Y_1, Y_2, Y_3, Cb_0 et Cr_0 de la figure 1.3(c) forment une MCU ;

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. La taille des MCU et les coefficients d'échantillonnage, identiques partout dans une image, sont inclus dans les sections JPEG décrites en annexe B.

Il convient alors de se poser la question de l'ordonnancement des blocs. L'image est découpée en MCU, balayées de gauche à droite, de haut en bas, la taille des MCU dépendant du sous-échantillonnage L'ordonnancement des blocs à l'intérieur d'une MCU suit les séquences ci-dessous :

4:4:4 MCU 8×8 : Y Cb Cr

4:2:2 MCU 8×16 : Y_1 Y_2 Cb Cr

4:2:0 MCU 16×16 : Y_1 Y_2 Y_3 Y_4 Cb Cr

Au niveau du décodeur, il faut à l'inverse sur-échantillonner les blocs Cb et Cr pour qu'ils recouvrent la MCU en totalité. Ceci signifie par exemple qu'en 4:2:2, la moitié gauche de chaque chrominance couvre le premier bloc Y_1, alors que la moitié droite couvre le second bloc Y_2.

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 un tableau de taille 8×16 par composante.

1.9 Conversion vers des pixels RGB

La conversion YCbCr vers RGB s'effectue pixel par pixel à partir des trois composantes de la MCU reconstituée (éventuellement issues d'un sur-échantillonnage pour les composantes de chrominance). Ainsi, pour chaque pixel dans l'espace YCbCr, on effectue le calcul suivant (donné dans la norme de JPEG) pour obtenir le pixel RGB :

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

Les formules simplifiées suivantes sont encore acceptables pour respecter les contraintes de rapport signal à bruit du standard. [9]

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

9. 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

Base de code fournie : principe et API

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 découpée en plusieurs modules. Chacun réalise une étape du décodage, et est fourni sous la forme :

— d'un fichier d'entête `.h` fournissant le prototype des fonctions qu'il contient ;
— d'un fichier objet `.o` compilé prêt à être lié à l'exécutable.

Votre objectif, durant la première phase du projet, sera de remplacer chacun des objets fournis par vos propres modules, qui devront respecter l'interface que nous avons édicté. Vous pourrez ainsi vérifier que vos fonctions réalisent effectivement la partie demandée en testant un module seul au milieu de la chaîne fournie par les enseignants. A terme, la totalité des fichiers objet fournis seront remplacés par ceux compilés à partir de votre propre code : vous aurez écrit votre propre décodeur !

Les modules peuvent être implémentés dans l'ordre que vous voulez, une estimation de la difficulté étant donnée dans la partie 3.1.1.

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 brut (« raw »). Plusieurs formats d'image bruts existent. On trouve par exemple le format « BitMaP » (ou BMP) de Microsoft. Dans ce 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 l'image proprement dite (dont le JPEG, d'ailleurs), mais nous ne l'utiliserons que comme format non compressé brut.

Le format TIFF est décrit en annexe C. Seules les informations requises dans notre contexte sont présentes, le standard TIFF complet étant disponible sur le site web d'Adobe si besoin.

2.2 Interfaces des modules fournis

Cette section décrit l'interface (on parle d'API, *Application Programming Interface*) des fonctions des différents modules. Une dernière fonction, non présentée ici, est la fonction principale `main`.

Les types de données utilisés sont pour l'essentiel les types de base du langage C99, plus quelques structures non définies. Bien entendu vous pourrez utiliser des structures en interne dans vos modules.

Les données sont géréralement stockées dans des tableaux à une dimension. Par exemple un bloc 8×8 est en représenté en mémoire par un vecteur 1×64. Moyennant une petite gymnastique sur les indices, cette représentation simplifiera beaucoup l'expression et la manipulation des différentes fonctions à implémenter.

Il vous est demandé de respecter dans votre projet les conventions de codage du noyau Linux, déjà utilisées dans la phase de préparation au C.

2.2.1 Conversion YCbCr vers RGB (module `conv.h`)

```
extern void YCbCr_to_ARGB(uint8_t *mcu_YCbCr[3],
                          uint32_t *mcu_RGB,
                          uint8_t nb_blocks_h,
                          uint8_t nb_blocks_v);
```

La fonction de conversion des composantes YCbCr en RGB s'applique sur les 3 composantes déjà décodées d'une MCU. Le sur-échantillonnage de Cb et Cr a déjà réalisé, donc le nombre de pixels (typiquement 8×8, 8×16, ou 16×16) est identique pour toutes les composantes.

Elle requiert en paramètre :

- un tableau `mcu_YCbCr` des 3 composantes d'une MCU, dans l'ordre Y, Cb, puis Cr. Les valeurs des composantes sont des entiers non signés sur 8 bits. Leur nombre dépend de la taille de la MCU.
- un tableau `mcu_RGB`, déjà alloué avec une taille adéquate et ne contenant rien de significatif, qui contiendra en sortie les valeurs RGB des pixels. Une valeur ARGB est codée sur 4 octets, non signés. Le premier octet n'est pas utilisé et doit être initialisé à 0, il correspond au canal A (*alpha*) de transparence. Les 3 octets suivants sont dans l'ordre : R (*red*), G (*green*), et B(*blue*). L'ordre des octets, indépendamment de 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 de la MCU, en horizontal (`nb_blocks_h`) et en vertical (`nb_blocks_v`).

Attention : les calculs incluent des opérations arithmétiques variées sur des types flottant et 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` (on parle de saturation).

2.2.2 Quantification inverse et Zigzag inverse (module `iqzz.h`)

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

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

Ses paramètres représentent, dans l'ordre : le bloc 8×8 initial, le bloc 8×8 après réorganisation zigzag inverse et multiplication par la table de quantification, et la table de quantification. Attention, la table lue dans le fichier JPEG est ordonnancée au format zigzag.

Noter qu'à ce stade les valeurs des blocs sont sur des entiers 32 bits signés.

2.2.3 Calcul de la transformée en cosinus discrète inverse (module `idct.h`)

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

La fonction de calcul de la transformée en cosinus discrète inverse ne prend que deux paramètres, le bloc 8×8 en entrée et le bloc 8×8 décodé en sortie.

Outre la transformée, cette fonction passe d'une représentation signée à une représentation non signée sur 8 bits. Il conviendra donc d'ajouter 128 à chaque coefficient obtenu. Comme pour la convertion YCbCr vers RGB, gérer la saturation.

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.

2.2.4 Correction du sous échantillonnage (module `upsampler.h`)

La correction du sous échantillonnage permet d'étendre les blocs compressés, afin de les faire couvrir l'ensemble de la MCU qu'ils représentent. La fonction *upsampler* doit être appelée pour chaque composante Y, Cb et Cr d'une MCU, avec les attributs adéquats.

```
extern void upsampler(uint8_t *in,
                      uint8_t nb_blocks_in_h, uint8_t nb_blocks_in_v,
                      uint8_t *out,
                      uint8_t nb_blocks_out_h, uint8_t nb_blocks_out_v);
```

Les paramètres de la fonction sont les suivants :

— *in* est un tableau représentant le ou les blocs d'une composante, éventuellement sous-échantillonnée (*down sampled*). Dans le cas d'une MCU de taille 8×8 pixels, où il n'y a pas de sous échantillonnage, c'est un tableau de 64 valeurs (un bloc). Sinon, il contient plusieurs blocs 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. *in* sera alors un tableau de 256 valeurs, qu'on peut voir comme une succession de 4 tableaux de 64 valeurs ;

— après l'exécution de la fonction, *out* contient les blocs corrigés (*up sampled*) qui couvrent donc la totalité de la MCU. Dans l'exemple précédemment cité de la composante Y dans un sous-échantillonnage 4:2:0, *out* est toujours un tableau 1D de 256 valeurs. Par contre, il ne représente plus une succession de 4 tableaux de 8×8 pixels, mais un seul tableau unifié de 16×16 pixels. L'ordre des valeurs n'est plus le même.

— *nb_blocks_in_h* et *nb_blocks_in_v* sont le nombre de blocs 8×8 du tableau `in`, en horizontal et en vertical.

— *nb_blocks_out_h* et *nb_blocks_out_v* sont ceux du tableau *out*, c'est-à-dire la dimension de la MCU.

2.2.5 Lecture dans le flux (module `bitstream.h`)

Ce module permet de lire des bits dans le fichier d'entrée (et non des octets comme dans une lecture standard), pour extraire les informations du format JFIF (annexe B) ou directement les données brutes de l'image compressée.

Notez que ce module est très facile à tester, de manière totalement indépendante du reste du projet JPEG.

```
struct bitstream;

extern struct bitstream *create_bitstream(const char *filename);

extern bool end_of_bitstream(struct bitstream *stream);

extern uint8_t read_bitstream(struct bitstream *stream,
                              uint8_t nb_bits, uint32_t *dest,
                              bool byte_stuffing);

extern bool skip_bitstream_until(struct bitstream *stream,
                                 uint8_t byte);

extern void free_bitstream(struct bitstream *stream);
```

Type de données Un flux de bits est représenté par le type de données `struct bitstream`. C'est une structure C dite *opaque* qui n'est que *déclarée* dans le fichier `bitstream.h`, mais dont on ne connait pas le contenu. Par contre ceci est suffisant pour l'utiliser ! Il suffit de créer une variable de type *pointeur* sur la structure, qui sera utilisée au travers des différentes fonctions. Par exemple :

```
struct bitstream *stream;
stream = create_bitstream("broquedis.jpg");
...
uint32_t bits;
uint8_t nb_read = read_bitstream(stream, 5, &bits, false);
...
free_bitstream(stream);
```

Quand vous implémenterez ce module, il vous reviendra de *définir* la structure dans votre fichier `bitstream.c`, avec les champs que vous aurez choisi pour réaliser les tâches demandées. [1]

Création et destruction La fonction `create_bitstream` crée un flux, positionné au début du fichier `filename`. Le prochain bit à lire est le donc le premier du fichier. Cette fonction alloue dynamiquement une variable de type `struct bitstream`, la remplie de manière adéquate, puis retourne son adresse mémoire. En cas d'erreur, la fonction retourne `NULL`.

La fonction `free_bitstream` sert à désallouer proprement un flux de bit référencé par le pointeur `stream` (la structure elle-même et tous ses champs alloués dynamiquement, le cas échéant).

1. Ce principe de masquer le contenu d'un type et de n'en permettre l'utilisation qu'au travers de fonctions se nomme l'*encapsulation*. C'est bien entendu un des principes fondamental de la programmation orientée objet, mais on peut s'en servir dans d'autres contexte. La preuve.

Lecture La fonction *read_bitstream* permet de lire des bits et d'avancer dans le flux. Ses paramètres sont :

- le flux de bits *stream* ;
- le nombre de bits *nb_bits* devant être lus, au maximum 32 ;
- *dest* contient l'adresse d'un entier 32 bits non signé dans lequel sont stockés les bits lus dans le flux. Si 11 bits ont été lus, le dernier est le bit de poids faible de `*dest` alors que le premier lu est à la position 10 ;
- un booléen *byte_stuffing* qui indique s'il faut tenir compte ou non du *byte stuffing* décrit section 2.2.7. S'il vaut *true*, tout octet `0x00` suivant immédiatement un octet `0xff` sur une adresse alignée (multiple de 8) doit être ignoré lors de la lecture.
- la fonction retourne finalement le nombre de bits ayant été effectivement lus. Ceci permet de vérifier que la lecture s'est déroulée correctement (comme pour un *fread* classique).

Pour alléger votre code, vous pouvez rajouter des macros ou fonctions qui appellent *read_bitstream* avec différents paramètres (typiquement pour lire 8, 16 ou `n` bits) et vérifient que la lecture a bien été effectuée.

La fonction *end_of_bitstream* retourne *true* si le flux a été entièrement parcouru, *false* s'il reste des bits à lire.

Enfin, la fonction *skip_bitstream_until* permet d'avancer dans le flux jusqu'à trouver un octet *aligné* égal à *byte*. S'il est trouvé, le flux est positionné juste avant cet octet et la fonction retourne *true*. Les bits parcourus pour arriver jusqu'à *byte* sont perdus. Si la valeur n'est pas trouvée, la fonction retourne *false* (et normalement le flux a été entièrement parcouru).

Cette fonction sera en pratique utilisée pour dérouler le flux *stream* et se positionner sur le marqueur JPEG suivant. Ceci permet 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.

Soit par exemple un flux contenant les données : `a3 04 5b ff ...` :

- des appels successifs à *read_bitstream* pour lire 2, 1 puis 11 bits retourneront les séquences : `10`, 1 et `00011000001`.
- *skip_bitstream_until*(*stream*, `0xff`) avancera le flux jusqu'à l'octet `0xff` suivant, sur une adresse alignée. Au passage 10 bits du flux seront perdus.
- un nouvel appel à *read_bitstream* pour lire 8 bits retournera donc `0xff`.

2.2.6 Gestion des tables de Huffman (module `huffman.h`)

Ce module permet de représenter, créer, utiliser et détruire les tables de Huffman du fichier JPEG.

```
struct huff_table;

extern struct huff_table *load_huffman_table(struct bitstream *stream,
                                             uint16_t *nb_byte_read);

extern int8_t next_huffman_value(struct huff_table *table,
                                 struct bitstream *stream);

extern void free_huffman_table(struct huff_table *table);
```

Type de données Une table de Huffman est représentée par le type de données `struct huffman`. Comme pour le `bitstream`, c'est une structure opaque que vous devrez *définir* dans votre fichier `huffman.c` avec les champs que vous aurez choisi pour la représentation d'une table de Huffman.

Création La fonction `load_huffman_table` parcourt le flux de bits `stream` pour construire une table de Huffman propre à l'image en cours de décodage. Au départ, le flux doit être positionné 3 octets après le marqueur `DHT` concerné (voir l'annexe B.2.6). Ainsi, si dans un fichier on trouve `ff c4 00 1a 00 01`, le prochain octet à lire dans `stream` devra être `01`.

Cette fonction retourne l'adresse d'une structure de type `struct huffman`, qui a été crée puis remplie selon la méthode décrite en section 1.5.1. Après l'exécution, l'entier pointé par le paramètre `nb_byte_read` contient le nombre d'octets qui ont été lus dans le flux pendant la création de l'arbre.

En cas d'échec lors de la création, la fonction retourne `NULL` et `*nb_byte_read` vaut -1.

Lecture de la prochaine valeur La fonction `next_huffman_value` retourne la prochaine valeur atteinte en parcourant la table de Huffman `table` selon les bits extraits du flux *stream*. Le parcours se fait comme décrit section 1.5.1.

Destruction La fonction `free_huffman_table` sert à désallouer proprement une table de Huffman référencée par le pointeur `table` (la structure elle-même et tous ses champs alloués dynamiquement, le cas échéant).

2.2.7 Récupération d'un bloc dans le fichier (module `unpack.h`)

```
extern void unpack_block(struct bitstream *stream,
                         struct huff_table *table_DC,
                         int32_t *pred_DC,
                         struct huff_table *table_AC,
                         int32_t bloc[64]);
```

La fonction `unpack_block` sert à récupérer un bloc 8×8 dans le fichier JPEG, selon les instructions décrites section 1.5. Ses paramètres sont :

- le flux de bits `stream` ;
- les tables de Huffman `table_DC` et `table_AC` respectivement utilisées pour décoder les coefficients DC et AC ;
- l'adresse d'un entier contenant la valeur du prédicateur DC. Elle devra être mise à jour par la fonction ;
- un tableau `bloc` dans lequel les différents coefficients en fréquence sont écrits, après décodage DPCM, Huffman et RLE.

Remarque importante : Dans le flux des données brutes des blocs compressés, il peut arriver qu'une valeur `0xff` alignée apparaisse. Cependant, cette valeur est particulière puisqu'elle pourrait aussi marquer le début d'une section JPEG. 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 interpréter ce `0x00` pour reconstruire les blocs, il faut jeter cet octet nul !

2.2.8 Écriture du résultat (module `tiff.h`)

```
struct tiff_file_desc;
```

La structure *tiff_file_desc* permet de stocker toute information que vous jugez nécessaire à la construction du fichier TIFF, comme par exemple les dimensions du fichier de sortie ou encore l'endianness définissant l'ordre dans lequel les octets doivent être écrits dans le fichier.

• **Initialisation** :

```
extern struct tiff_file_desc *init_tiff_file(const char *file_name,
                                             uint32_t width,
                                             uint32_t height,
                                             uint32_t row_per_strip);
```

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. Elle prend en paramètre le nom du fichier de sortie, les dimensions de l'image et le nombre de lignes de pixels par bande du fichier TIFF. Elle renvoie une structure *tiff_file_desc* correctement initialisée.

• **Ecriture de l'image** :

```
extern void write_tiff_file(struct tiff_file_desc *tfd,
                            uint32_t *mcu_rgb,
                            uint8_t nb_blocks_h,
                            uint8_t nb_blocks_v);
```

write_tiff_file permet d'écrire la MCU *mcu_rgb* dans le fichier TIFF représenté par la structure *tiff_file_desc*. Plus précisément, *mcu_rgb* est un tableau représentant une MCU décodée en ARGB (4 octets), dont l'élément A est à l'offset 0. *nb_blocks_h* et *nb_blocks_v* représentent le nombre de blocs 8×8 de la MCU, en horizontal et en vertical.

• **Finalisation de l'image** :

```
extern void close_tiff_file(struct tiff_file_desc *tfd);
```

close_tiff_file permet de clore le fichier représenté par la structure *tiff_file_desc* passée en paramètre, en nettoyant proprement le module.

Notez que le module *tiff* vous laisse une certaine liberté de conception, la seule contrainte étant que l'intégralité des informations permettant la lecture du fichier TIFF ait été écrite après un appel à *close_tiff_file*. En particulier, on pourrait imaginer une implémentation dans laquelle un appel à *write_tiff_file* enregistrerait en mémoire les informations nécessaires à l'écriture de la MCU passée en paramètre, mais que l'écriture effective des MCUs soit réalisée par la fonction *close_tiff_file*.

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

Nous vous recommandons, afin de faciliter les périodes de débogages (dont vous ne manquerez pas), d'effectuer ces étapes dans l'ordre *inverse* du décodage, c'est-à-dire de commencer par le module de conversion vers RGB et remonter la chaîne de décodage jusqu'à la lecture des entêtes (de l'étape 7 à l'étape 1). Vous pourrez alors faire l'écriture du TIFF (étape 8, qui est un peu à part) et bien sûr l'écriture du programme principal `main` (qui est un gros morceau).

Vous êtes bien entendu libres de procéder dans un ordre différent, mais suivre cet ordre vous donnera plus de contrôle puisque à tout moment, vous maîtrisez ce qui est fait *après* la partie en cours de travail. Cela est particulièrement utile par exemple si votre module est buggé mais que cela fait planter un des modules suivants dans la chaîne de décodage : vous pourrez alors utiliser les outils de débogage habituels puisque vous aurez les sources du module qui plante alors que vous n'avez pas celles des modules fournis.

Modules	Difficulté présentie
conversion YCbCr vers RGB	☆
upsampling	☆☆☆☆
IDCT	☆
quantification et zigzag	☆☆
extraction des blocs	☆☆☆
gestion des tables de Huffman	☆☆☆☆
lecture du *bitstream*	☆☆☆☆
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 ceux compilés à partir de votre propre code. Ceci signifie que toutes les fonctions de décodage décrites précédemment doivent être implantées.

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

Une fois le décodeur terminé, 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

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.

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 :
— *hexdump*, en particulier avec l'option `-C`. Ce programme affiche de manière textuelle le contenu octet par octet d'un fichier. Dans ce projet, c'est très utile pour regarder les données du fichier JPEG, comprendre sa structure et vérifier que les données lues sont correctes.
— *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.

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.

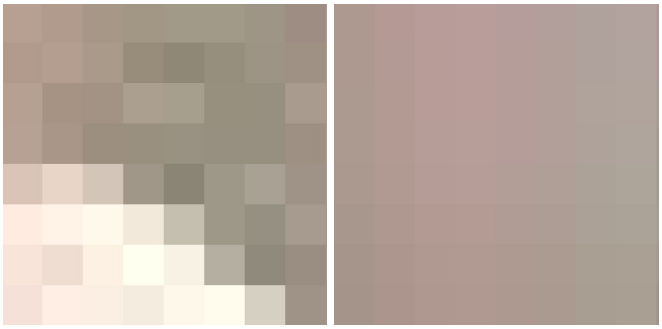
Annexe A

Exemple d'encodage d'une MCU

Cette annexe donne un exemple de codage d'une MCU, de la compression des pixels RBG initiaux jusqu'à l'encodage bit par bit dans le flux de données JPEG. A étudier en parallèle de la section 1.5 qui décrit les étapes du *dé*codage !

A.1 MCU en RGB

Pour cet exemple, on suppose une MCU de taille 8×16, qui sera sous-échantillonnée horizontalement 4:2:2. Les deux blocs en représentation RGB sont les suivants :



```
b8a092 b19b8d a79787 a29785 a19a88 a19a88 9e9586 9e8d83  ad9990 b59995 b89c99 b99d9a b59d9b b29f9b afa29c b2a39e
b29b8d b49e90 aa9a8a 988d7b 8f8876 968f7d 9d9485 9f9184  ad9990 b39a95 b89c98 b99d9a b59d99 b29f99 afa29c b1a49e
b6a092 a69384 a39384 ab9f8f a69f8d 97907e 979080 a99b8e  ac9a90 b39a93 b89c98 b99d99 b59d99 b29f99 b0a39b b1a49c
b7a194 a99688 9c8f7f 99907f 999282 979080 979080 9e9083  ac9a90 b39a93 b69d98 b79e99 b59d99 b29f99 afa29a afa59c
dac4b7 e8d5c7 d3c5b8 a09788 8b8575 9e9888 a9a194 9f9387  ab998f b19a92 b59c97 b69d98 b29e97 b09f97 aca298 ada59a
ffebe0 fff3e7 fff9ec f2e9da c5bfaf 9e9888 969082 a79b8f  a8978d af9890 b39a93 b49b94 b09c95 ae9d95 aba197 aba398
f8e4d9 efddd1 fdf1e3 ffffef f8f2e4 b5afa1 908a7c 9a8e82  a6958b ad968e b09991 b19a92 ae9a91 ac9b91 a99f93 aaa094
f5e1d8 ffeee4 fbefe3 f4ecdf fef8ea fffbed d6d0c2 9f9387  a5948a ac958d af9890 b09991 ad9990 ab9a90 a89e92 a99f93
```

A.2 Représentation YCbCr

La représentation de cette MCU en YCbCr est :

```
a6 a1 9b 9a 9b 9c 97 92  9e a1 a4 a5 a4 a4 a5 a7
9f a3 9d 8e 89 8f 95 94  9e a1 a4 a5 a4 a4 a5 a7
a5 97 96 a1 9e 90 90 9e  9e a1 a4 a5 a5 a4 a6 a7
```

```
a7 9b 91 91 92 91 91 94  9e a1 a4 a5 a4 a4 a6 a7
ca da c8 98 85 98 a2 96  9d a0 a3 a4 a3 a3 a4 a6
f0 f7 fb e8 bd 96 90 9d  9b 9e a1 a2 a1 a1 a3 a4
e9 e0 f1 ff ef ad 8a 90  99 9c 9f a0 9f 9f a1 a2
e7 f2 f1 eb f7 fb d0 97  98 9b 9e 9f 9e 9e a0 a1
```

```
75 75 75 76 76 76 76 77  79 79 79 7a 7a 7a 7b 7b
75 75 75 76 76 76 77 77  79 79 79 7a 7a 7a 7b 7b
75 75 76 76 76 77 77 77  79 79 79 79 7a 7a 7a 7a
76 76 76 76 77 77 77 77  78 79 79 79 79 7a 7a 7a
76 76 76 77 77 77 78 78  78 78 78 79 79 79 7a 7a
76 76 77 77 77 78 78 78  78 78 78 78 79 79 79 79
76 77 77 77 78 78 78 78  78 78 78 78 79 79 79 79
77 77 77 77 78 78 78 78  77 78 78 78 78 79 79 79
```

```
8d 8b 88 86 85 85 86 87  8c 8d 8e 8e 8d 8b 88 87
8d 8b 88 86 85 85 86 87  8c 8d 8e 8e 8d 8b 88 86
8c 8b 88 86 85 85 86 87  8b 8c 8e 8e 8d 8b 88 86
8c 8a 88 85 84 85 86 87  8b 8c 8d 8e 8d 8a 88 86
8c 8a 87 85 84 84 85 86  8b 8c 8d 8d 8c 8a 87 86
8b 8a 87 85 84 84 85 86  8a 8b 8d 8d 8c 8a 87 85
8b 89 87 84 83 84 85 86  8a 8b 8c 8d 8c 89 87 85
8b 89 87 84 83 84 85 86  8a 8b 8c 8d 8c 89 87 85
```

A.3 Sous échantillonnage

Cette MCU est sous-échantillonnée horizontalement, pour ne conserver qu'un seul bloc 8×8 par composante de chrominance. [1]

```
a6 a1 9b 9a 9b 9c 97 92   9e a1 a4 a5 a4 a4 a5 a7
9f a3 9d 8e 89 8f 95 94   9e a1 a4 a5 a4 a4 a5 a7
a5 97 96 a1 9e 90 90 9e   9e a1 a4 a5 a5 a4 a6 a7
a7 9b 91 91 92 91 91 94   9e a1 a4 a5 a4 a4 a6 a7
ca da c8 98 85 98 a2 96   9d a0 a3 a4 a3 a3 a4 a6
f0 f7 fb e8 bd 96 90 9d   9b 9e a1 a2 a1 a1 a3 a4
e9 e0 f1 ff ef ad 8a 90   99 9c 9f a0 9f 9f a1 a2
e7 f2 f1 eb f7 fb d0 97   98 9b 9e 9f 9e 9e a0 a1
```

```
75 75 76 77 79 7a 7b 7b
75 75 76 77 78 7a 7a 7b
75 76 76 77 78 79 7a 7a
76 76 77 77 78 79 7a 7a
76 77 77 78 78 79 79 79
77 77 77 78 78 78 79 79
77 77 78 78 78 78 78 78
78 78 78 78 78 78 78 78
```

1. Noter qu'il ne s'agit pas des moyennes des chrominances, mais d'un calcul YCbCr sur les moyennes RGB. Voir par exemple l'avant dernière valeur de la première ligne, deuxième bloc (page suivante).

```
8d 88 84 87 8d 8f 8b 86
8d 88 84 87 8c 8f 8b 86
8c 88 84 86 8c 8f 8b 86
8c 87 84 86 8c 8e 8b 86
8c 87 83 86 8c 8e 8a 85
8c 87 83 85 8b 8e 8a 85
8b 86 83 85 8b 8d 8a 85
8b 86 83 85 8b 8d 8a 85
```

A.4 DCT : passage au domaine fréquentiel

La DCT est ensuite appliquée à chacun des blocs de données. Les basses fréquences sont en haut à gauche et les hautes fréquences en bas à droite.

```
 389 134 -22  -4  -1 -2  1  1          270 -18 -6 -10 0  0 0 0
-207 -87  62  -1   2  2 -1 -3           17   0  0   0 0 -1 0 0
  79  -8 -54  29  -1 -1  0  1           -8   0  0   0 0  0 0 0
  16  60  15 -36   0  0  0  1            0   0  0   0 0  0 0 0
  -9 -35  20  33 -38  1 -1 -2            1   0  0   0 0  0 0 0
 -16   0 -32   1  46 -2  1  2            0   0  0   0 0  0 0 0
  32   1  -2  -1   2  2  0 -3            0   0  0   0 0  0 0 0
  -1   0   1   1  -1 -1  1  1            0   0  0   0 0  0 0 0
```

```
-65 -10  0 0 0 0  0 0
  0  -6  0 1 0 0  0 0
  0   0  0 0 0 0  0 0
  0   0  0 0 0 0  0 0
  1   0  0 0 0 0  0 0
  0   0  0 0 0 0  0 1
  0   0  0 0 0 0 -1 0
  0   0 -1 0 0 0  0 0
```

```
70 -5 0 28 0 0 0  0
   5  0 0  0 0 0 0 -1
   0  0 0  0 0 0 0  0
   0  1 0  1 0 0 0  0
   0  0 0  0 1 0 0  0
   0  0 0  0 0 0 0  1
   0  0 0  0 0 0 0  0
   0  0 0  0 0 0 0  0
```

A.5 Quantification

La quantification consiste à diviser les blocs par les tables de quantification, une pour la luminance et une pour les deux chrominances. Dans cet exemple, les deux tables sont issues du projet *The Gimp* :

```
05 03 03 05 07 0c 0f 12
04 04 04 06 08 11 12 11
04 04 05 07 0c 11 15 11
04 05 07 09 0f 1a 18 13
05 07 0b 11 14 21 1f 17
07 0b 11 13 18 1f 22 1c
0f 13 17 1a 1f 24 24 1e
16 1c 1d 1d 22 1e 1f 1e
```

```
05 05 07 0e 1e 1e 1e 1e
05 06 08 14 1e 1e 1e 1e
07 08 11 1e 1e 1e 1e 1e
0e 14 1e 1e 1e 1e 1e 1e
1e 1e 1e 1e 1e 1e 1e 1e
1e 1e 1e 1e 1e 1e 1e 1e
1e 1e 1e 1e 1e 1e 1e 1e
1e 1e 1e 1e 1e 1e 1e 1e
```

Après quantification, les blocs de la MCU sont finalement :

```
 78  45  -7 -1 0 0 0 0      54 -6 -2 -2 0 0 0 0
-52 -22  16  0 0 0 0 0       4  0  0  0 0 0 0 0
 20  -2 -11  4 0 0 0 0      -2  0  0  0 0 0 0 0
  4  12   2 -4 0 0 0 0       0  0  0  0 0 0 0 0
 -2  -5   2  2 2 0 0 0       0  0  0  0 0 0 0 0
 -2   0   2  0 2 0 0 0       0  0  0  0 0 0 0 0
  2   0   0  0 0 0 0 0       0  0  0  0 0 0 0 0
  0   0   0  0 0 0 0 0       0  0  0  0 0 0 0 0
```

```
-13 -2 0 0 0 0 0 0
  0 -1 0 0 0 0 0 0
  0  0 0 0 0 0 0 0
  0  0 0 0 0 0 0 0
  0  0 0 0 0 0 0 0
  0  0 0 0 0 0 0 0
  0  0 0 0 0 0 0 0
  0  0 0 0 0 0 0 0
```

```
14 -1 0 2 0 0 0 0
 1  0 0 0 0 0 0 0
 0  0 0 0 0 0 0 0
 0  0 0 0 0 0 0 0
 0  0 0 0 0 0 0 0
 0  0 0 0 0 0 0 0
 0  0 0 0 0 0 0 0
 0  0 0 0 0 0 0 0
```

A.6 Réordonnancement Zigzag

Suite à l'étape de quantification, beaucoup de coefficients des haute-fréquences en bas à droite du bloc sont nuls. La réorganisation en Zigzag a pour objectif de regouper les 0 à la fin pour pouvoir les supprimer.

Les blocs sont encore affichés sous forme 8×8 pour plus de lisibilité par rapport aux données précédentes, mais fondamentalement on les considère comme un vecteur 1×64.

```
78 45 -52 20 -22 -7 -1 16     54 -6 4 -2 0 -2 -2 0
-2  4  -2 12 -11  0  0  0      0  0 0  0 0  0  0 0
```

```
 0  4  2  -5  -2  2  0  2      0  0 0  0 0  0  0 0
-4  0  0   0   0  0  0  0      0  0 0  0 0  0  0 0
 2  2  0   0   0  0  0  2      0  0 0  0 0  0  0 0
 0  0  0   0   0  0  2  0      0  0 0  0 0  0  0 0
 0  0  0   0   0  0  0  0      0  0 0  0 0  0  0 0
 0  0  0   0   0  0  0  0      0  0 0  0 0  0  0 0
```

```
-13 -2 0 0 -1 0 0 0
  0  0 0 0  0 0 0 0
  0  0 0 0  0 0 0 0
  0  0 0 0  0 0 0 0
  0  0 0 0  0 0 0 0
  0  0 0 0  0 0 0 0
  0  0 0 0  0 0 0 0
  0  0 0 0  0 0 0 0
```

```
14 -1 1 0 0 0 2 0
 0  0 0 0 0 0 0 0
 0  0 0 0 0 0 0 0
 0  0 0 0 0 0 0 0
 0  0 0 0 0 0 0 0
 0  0 0 0 0 0 0 0
 0  0 0 0 0 0 0 0
 0  0 0 0 0 0 0 0
```

A.7 Codage différentiel DC

La composante continue DC est la première valeur d'un bloc.

Valeur DC En supposant que la MCU de cet exemple est la première du fichier (en haut à gauche de l'image), la valeur 78 doit être encodée en premier. Elle appartient à la classe de magnitude 7 : $-147, \ldots, -64, 64, \ldots, 127$. Dans cette classe, l'indice de 78 est `001110`

Dans cet exemple, on suppose que le code de Huffman de la magnitude 7 est `11110` (elle est portée par une feuille de profondeur 5). Finalement, la suite de bits à inclure dans flux de l'image compressé est le code de la magnitude puis l'indice dans la classe de magnitude, soit ici : `111101001110`

Bloc suivant La valeur DC du bloc suivant de la composante Y est 54. Par contre, la valeur à encoder est la différence par rapport à la valeur DC du bloc précédent (de la même composante de lumière), soit ici 54-78=-24. En supposant que le code de Huffman de la magnitude 5 soit `110`, l'encodage de -24 dans le flux de bits est `11000111`.

A.8 Codage AC avec RLE

On s'intéresse au codage des 63 coefficients AC du bloc de première chrominance. La composante continue -13 a déjà été encodée dans le flux. Il reste donc la séquence : -2 0 0 -1 0 0 ... 0.

Pour le premier coefficient -2 :

— Un premier symbole sur un octet est calculé. Les quatre bits de poids forts sont nuls, car aucun 0 ne précède ce coefficient. Les quatre bits de poids faible contiennent la magnitude de -2, qui est 2. Le symbole est donc ici 0x02.
— Dans la classe de magnitude 2, l'indice en binaire de -2 est `01`

Le prochain coefficient non nul est -1 :

— Le symbole sur un octet est ici 0x21 : 0x2 sur les poids forts car le coefficient est précédé d'une séquence de deux 0, et 0x1 en poids faible car -1 est de magnitude 1.
— Dans la classe de magnitude 1, l'indice de -1 est `0`

Tous les coefficients suivant étant nuls, il suffit de mettre une balise EOB 0x00 pour terminer le codage du bloc.

La séquence totale à encoder est donc : 0x02 suivi des bits `01`, puis 0x21 suivi d'un bit `0` puis 0x00.

Encodage dans le flux de bits En supposant que les codes de Huffman des symboles soient `1110` pour 0x02, `111110` pour 0x21 et `01` pour 0x0, l'ensemble des 63 coefficients AC du bloc est totalement encodé dans le flux par les 15 bits : `111001111110001`

Annexe B

Le format JPEG

B.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 repérer dans le flux JPEG, on utilise des marqueurs, ayant la forme `0xff??`, avec le `??` qui permet de distinguer les marqueurs entre eux (*cf.* B.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.

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 ($i_{AC/DC}, i_C$).

B.2 Sections JPEG

Le format classique d'une section JPEG est le suivant :

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)

B.2.1 Marqueurs de début et de fin d'image

Toute image JPEG débute par un marqueur SOI (Start of Image) et termine par un marqueur EOI (End of Image). Ces deux marqueurs font exception dans le JPEG puisqu'ils ne suivent pas le format classique décrit ci-dessus : ils sont utilisés sans aucune autre information et servent de repères.

Bien que dans la majorité des cas un fichier JPEG ne contient qu'une seule image, un fichier JPEG peut contenir plusieurs images JPEG successives, formant alors un fichier vidéo au format MJPEG. Chaque image de cette vidéo est délimitée par les balises SOI/EOI.

B.2.2 APPx - Section Application

Cette section permet d'enregistrer des informations propres à chaque application, application signifiant ici format d'encapsulation. Dans notre cas, on ne s'intéressera qu'au marqueur APP0, qui sert pour l'encapsulation 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`	5	'J' 'F' 'I' 'F' '\0'
`0x09`	x	Données spécifiques au JFIF, non traitées

B.2.3 COM - Commentaire

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

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

B.2.4 DQT - Define Quantization Table

Cette section permet de définir une ou plusieurs tables 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.

Un fichier JPEG peut contenir une seule section DQT avec plusieurs tables, ou plusieurs section DQT avec une table à chaque fois. C'est la longueur de la section qui permet de déterminer combien de tables sont contenues.

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

B.2.5 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 B.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 et la table de quantification associée à la composante (i_Q). Dans chaque direction `h` ou `v`, le « facteur d'échantillonnage » (*sampling factor*) indique en fait le nombre relatif de blocs d'une composante par rapport à la taille de la MCU. Soit par exemple une MCU de 2×2 blocs (16×16 pixels). En (4:2:2) les facteurs de la composante Y vaudront 2 : il faut 2 blocs Y pour les 2 blocs de la MCU, le taux d'échantillonnage est de 1 pour 1 (pas de sous-échantillonnage). Pour Cb, le facteur horizontal vaudra 1 : il faut 1 bloc Cb pour 2 blocs de la MCU, taux de 1 pour 2.

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'échantillonnage (sampling factor) horizontal - 4 bits : Facteur d'échantillonnage (sampling factor) vertical - 1 octet : Table de quantification i_Q associée

B.2.6 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`	 3 bits 1 bit 4 bits	Informations sur la table de Huffman : - non utilisés, doit valoir 0 (sinon erreur) - type (0=DC, 1=AC) - indice (0..3, ou 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)

B.2.7 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 échantillonnage). 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-échantillonnage horizontal de type 4:2:2, 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

B.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 étendue (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 C

Le format TIFF

C.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 un 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.

C.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.

C.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.

C.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

C.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**.

C.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.

C.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*.

C.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.

C.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

C.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)