Ensimag — Printemps 2018

—

Projet Logiciel en C

—

Sujet : Encodeur JPEG

Auteurs : Des enseignants actuels et antérieurs du projet C

Table des matières

Chapitre 1

Introduction

Le format JPEG est l'un des formats les plus répandus en matière d'image numérique. Il est en particulier utilisé comme format de compression par la plupart des appareils photo numériques, étant donné que le coût de calcul et la qualité sont acceptables, pour une taille d'image résultante petite.

Le *codec* [1] JPEG est tout d'abord présenté en détail au chapitre 2 et illustré sur un exemple en annexe A. Le chapitre 3 présente les spécifications, les modules et outils fournis, et quelques conseils importants d'organisation pour ce projet. Finalement, le chapitre 4 formalise le travail à rendre et les détails de l'évaluation.

L'objectif de ce projet est de **réaliser en langage C un encodeur qui convertit les images au format brut (PPM) en un format compressé (JPEG)**. Il est nécessaire d'avoir une bonne compréhension du codec, qui reprend notamment des notions vues dans les enseignements Théorie de l'information et Bases de la Programmation Impérative. Mais l'essentiel de votre travail sera bien évidemment de **concevoir** et d'**implémenter** votre encodeur en langage C.

Bon courage à tou(te)s, et bienvenue dans le monde merveilleux du JPEG ! *Enjoy* !

1. *code-decode*, format ou dispositif de compression/décompression d'informations

Chapitre 2

Présentation du codec 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 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.

L'encodeur JPEG demandé dans ce projet 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. Donc votre projet C est en fait un « encodeur `ISO/IEC IS 10918-1 | ITU-T Recommendation T.81` ». Qu'on se le dise !

2.1 Principe général du codec JPEG

Cette section détaille les étapes successives mises en œuvre lors de l'encodage, c'est-à-dire la conversion d'une image au format PPM vers une image au format JPEG.

Tout d'abord, l'image est partitionnée en macroblocs ou MCU pour *Minimum Coded Unit*. La plupart du temps, les MCUs sont de taille 8×8, 16×8, 8×16 ou 16×16 pixels selon le facteur d'échantillonnage (voir section 2.7). Chaque MCU est ensuite réorganisée en un ou plusieurs blocs de taille 8×8 pixels. La suite porte sur la compression d'un bloc 8×8.

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 positions du domaine spatial. 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 moins 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 zig-zag (ZZ) afin de transformer le bloc en un vecteur de 64×1 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 que des valeurs, puis un codage entropique[3] dit de *Huffman* qui utilise un dictionnaire spécifique à l'image en cours de traitement.

Les étapes ci-dessus sont appliquées à tous les blocs composant les MCUs de l'image. La concaténation de ces vecteurs compressés forme 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.

Les opérations de codage/décodage sont résumées figure 2.1, puis détaillées dans les sections suivantes (dans le sens du décodage). L'annexe A fournit un exemple numérique du codage d'une MCU.

2.2 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, l'encodeur utilise deux types de représentation de l'image.

Le format RGB, le plus courant, est le format utilisé manipulé en entrée. Il représente chaque couleur de pixel en donnant la proportion de trois couleurs primitives : le rouge (R), le vert (G), et le bleu (B). Une information de transparence *alpha* (A) peut également être fournie (on parle alors de ARGB), mais elle ne sera pas utilisée dans ce projet. Le format RGB est le format utilisé en amont et en aval de l'encodeur.

2. Il s'agit là de fréquences spatiales 2D avec une dimension verticale et une dimension horizontale.

3. C'est-à-dire qui cherche la quantité minimale d'information nécessaire pour représenter un message, aussi appelé entropie.

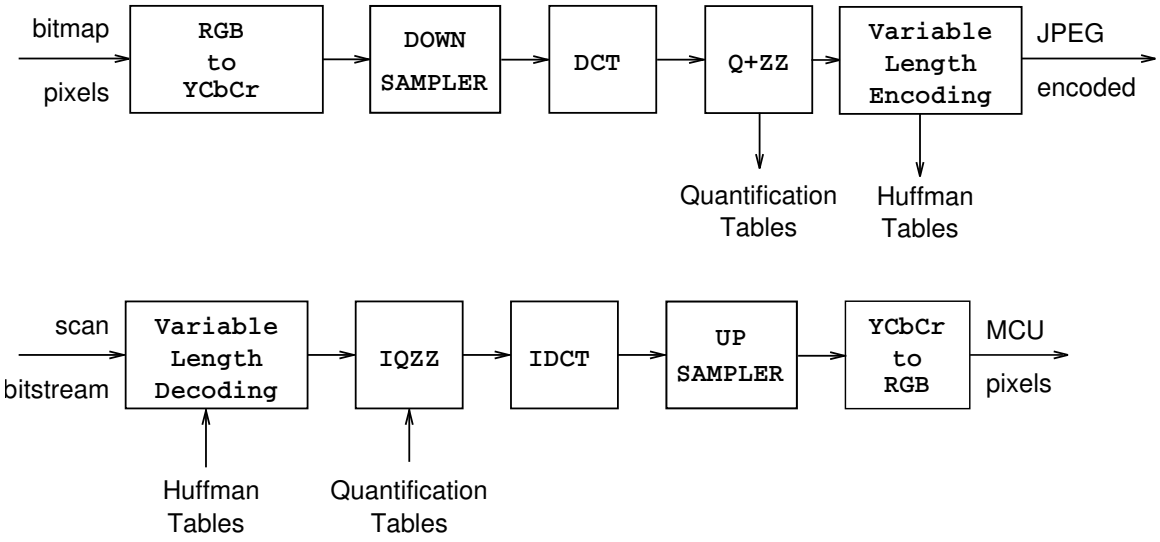


FIGURE 2.1 – Principe du codec JPEG : opérations de codage (en haut) et de décodage (en bas).

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 humain est moins sensible.

2.3 Structuration d'un fichier JPEG

Une image JPEG est stockée dans un fichier binaire considéré comme un flux d'octets, appelé dans la suite le *bitstream*. La norme JFIF dicte la façon dont ce flux d'octets est organisé. En pratique, le *bitstream* représente l'intégralité de l'image encodée et il est constitué d'une succession 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 lors du décodage). Un marqueur et ses données associées représentent une *section*.

On distingue deux grands types de sections :

— celles qui permettent de définir 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, les facteurs d'échantillonnage ou les tables de Huffman utilisées. Elles peuvent nécessiter un traitement particulier avant d'être utilisables. Il arrive souvent qu'on fasse référence à ces sections comme faisant partie de *l'entête* d'un fichier JPEG ;

— celles qui permettent de représenter l'image : ce sont les données brutes qui contiennent l'image encodée.

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.

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 qui leur sont associées lors du décodage de l'image. On notera ici 4 marqueurs importants :

SOI : le marqueur *Start Of Image* n'apparait qu'une fois par fichier JFIF, il représente le début du fichier ;

SOF : le marqueur *Start Of Frame* marque le début d'une *frame* JPEG, c'est-à-dire le début de l'image effectivement encodée avec son entête et ses données brutes. Le marqueur SOF est associé à un numéro, qui permet de repérer le type d'encodage utilisé. Dans notre cas, ce sera toujours *SOF0*. La section SOF contient notamment la taille de l'image et les facteurs d'échantillonnage utilisés. Un fichier JFIF peut éventuellement contenir plusieurs frames et donc plusieurs marqueurs SOF, par exemple s'il représente une vidéo au format MJPEG qui contient une succession d'images. Nous ne traiterons pas ce cas ;

SOS : le marqueur *Start Of Scan* indique le début des données brutes de l'image encodée. En mode *baseline*, on en trouve autant que de marqueurs SOF dans un fichier JFIF (1 dans notre cas). En mode *progressive* les données des différentes résolutions sont dans des *Scans* différents ;

EOI : le marqueur *End Of Image* marque la fin du fichier et n'apparait donc qu'une seule fois.

2.4 Format de fichier d'entrée PPM

Votre encodeur prendra en entrée des images au format PPM (*Portable PixMap*) dans le cas d'une image en couleur, ou PGM (*Portable GreyMap*) pour le cas en niveaux de gris. Il s'agit d'un format « brut » très simple, sans compression, que vous avez déjà eu l'occasion d'utiliser dans la partie préparation au langage C.

Le principe est de lire d'abord un entête *textuel* comprenant :

— un *magic number* précisant le format de l'image, `P6` pour des pixels à trois couleur RGB ;
— la largeur et la hauteur de l'image, en nombre de pixels ;
— le nombre de valeurs d'une composante de couleur (255 dans notre cas : chaque couleur prend une valeur entre 0 et 255) ;

Ensuite, les données de couleur sont directement écrites en binaire : trois octets pour les valeurs R, G et B du premier pixel, puis trois autres pour le second pixel, et ainsi de suite. Le format PGM est une variante pour les images en niveaux de gris : l'identifiant est cette fois `P5`, et la couleur de chaque pixel est codée sur un seul octet dont la valeur varie entre 0 (noir) et 255 (blanc). Un exemple est donné figure 2.2.

2.5 Découpage de l'image en MCUs

La premère étape consiste à découper l'image en MCUs.

La taille de l'image n'étant par forcément un multiple de la taille des MCUs, le découpage en MCUs peut « déborder » à droite et en bas. (figure 2.3). A l'encodage, la norme recommande de compléter les MCUs en dupliquant la dernière colonne (respectivement ligne) contenue dans l'image dans les colonnes (respectivement lignes) en trop.

```
$hexdump -C cocorico.ppm
```

```
00000000  50 36 0a 33 20 32 0a 32  35 35 0a 00 00 ff ff ff  |P6.3 2.255......|
00000010  ff ff 00 00 00 00 ff ff  ff ff ff 00 00           |.............|
```

```
$hexdump -C cocorico_bw.ppm
```

```
00000000  50 35 0a 33 20 32 0a 32  35 35 0a 12 ff 36 12 ff  |P5.3 2.255...6..|
00000010  36                                                |6|
```

FIGURE 2.2 – En haut, trace de la commande `hexdump -C` sur une image 3×2 représentant un drapeau français. Notez les caractères ASCII de l'entête textuel puis la suite des couleurs RGB, pixel par pixel. En bas, la version en niveaux de gris (français, italien, irlandais ? C'est moins clair...). Cette fois, il n'y a qu'un seul octet de couleur par pixel.

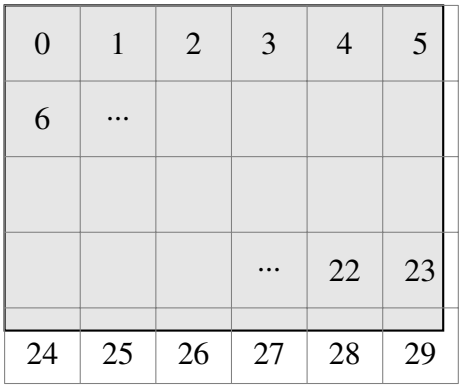


FIGURE 2.3 – Exemple d'une image 46 × 35 (fond gris) à compresser sans sous-échantillonnage, soit avec des MCUs composées d'un seul bloc 8 × 8. Pour couvrir l'image en entier, 6 × 5 MCUs sont nécessaires. Les MCUs les plus à droite et en bas (les 6e, 12e, 18e, puis 24e à 30e) devront être complétées lors de la compression.

2.6 Conversion RGB vers YCbCr

La conversion RGB vers YCbCr s'effectue pixel par pixel pour chaque bloc de chaque composante. Ainsi, pour chaque pixel dans l'espace RGB, on effectue le calcul suivant pour obtenir la valeur du pixel dans l'espace YCbCr :

$$
\begin{aligned}
Y &= 0.299 \times R + 0.587 \times G + 0.114 \times B \\
Cb &= -0.1687 \times R - 0.3313 \times G + 0.5 \times B + 128 \\
Cr &= 0.5 \times R - 0.4187 \times G - 0.0813 \times B + 128
\end{aligned}
$$

Ces calculs incluent des opérations arithmétiques sur des valeurs flottantes et signées. Sachant que les valeurs RGB sont forcément comprises entre 0 et 255, il conviendra de choisir le plus petit type de données entier permettant d'encoder toute la plage de valeurs possibles pour Y, Cb et Cr lors de l'écriture de votre encodeur.

2.7 Compression des MCUs

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 de valeurs, appelées échantillons, pour certaines composantes de l'image. Pour prendre un exemple, imaginons qu'on travaille sur une image couleur YCbCr partitionnée en MCUs de 2×2 blocs de 8×8 pixels chacun, pour un total de 256 pixels.

Ces 256 pixels ont chacun un échantillon pour chaque composante, le stockage nécessiterait donc $256 \times 3 = 768$ échantillons. **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 pour 2 pixels de l'image consécutifs horizontalement, un seul échantillon par composante de chrominance suffit. Il faudrait seulement alors $256 + 128 + 128 = 512$ échantillons pour représenter toutes les composantes, ce qui réduit notablement la place occupée ! Si on applique le même raisonnement sur les pixels de l'image consécutifs verticalement, on se retrouve à associer à 4 pixels un seul échantillon par chrominance, et on tombe à une occupation mémoire de $256 + 64 + 64 = 384$ échantillons.

2.7.1 Sous-échantillonnage de l'image

Dans ce document, nous utiliserons une notation directement en lien avec les valeurs présentes dans les sections JPEG de l'entête, décrites en annexe B, qui déterminent le facteur d'échantillonnage (*sampling factors*). Ces valeurs sont identiques partout dans une image. En pratique, on utilisera la notation $h \times v$ de la forme :

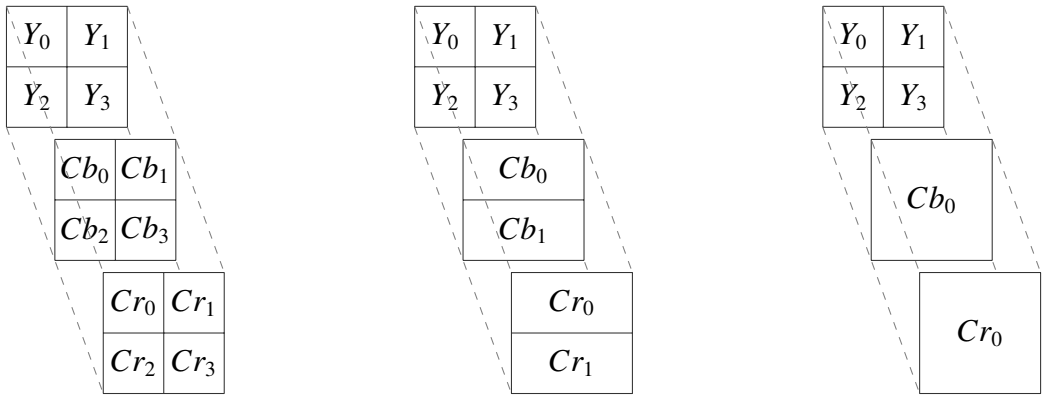
$$h_1 \times v_1, h_2 \times v_2, h_3 \times v_3$$

où h_i et v_i représentent le nombre de blocs horizontaux et verticaux pour la composante i. Comme Y n'est jamais compressé, **le facteur d'échantillonnage de Y donne les dimensions de la MCU en nombre de blocs**. Les sous-échantillonnages les plus courants sont décrits ci-dessous. Votre encodeur devra supporter au moins ces trois combinaisons, mais nous vous encourageons à gérer tous les cas !

Pas de sous-échantillonnage Le nombre de blocs est identique pour toutes les composantes. On se retrouve avec des facteurs de la forme $h_1 \times v_1$, $h_1 \times v_1$, $h_1 \times v_1$ (le même nombre de blocs répété pour toutes les composantes). La plupart du temps, on travaille dans ce cas sur des MCU de taille 1×1 ($h_1 = v_1 = 1$), mais on pourrait très bien trouver des images sans sous-échantillonnage découpées en MCUs de tailles différentes, comme par exemple 2×1 ou 1×2. [5] Sans sous-échantillonnage, la qualité de l'image est optimale mais le taux de compression est le plus faible.

Sous-échantillonnage horizontal La composante sous-échantillonnée comprend deux fois moins de blocs en horizontal que la composante Y. Le nombre de blocs en vertical reste le même pour les trois composantes. Par exemple, les facteurs d'échantillonnage « 2×2, 1×2, 1×2 » et « 2×1, 1×1, 1×1 » représentent tous deux une compression horizontale de Cb et Cr, mais avec des tailles de MCU différentes : 2×2 blocs dans le premier cas, 2×1 blocs dans le second. Comme la moitié de la résolution horizontale de la chrominance est éliminée pour Cb

5. La seule contrainte imposée par la norme JPEG étant que la somme sur i des $h_i \times v_i$ soit inférieure ou égale à 10.



(a) Sans sous-échantillonnage

(b) Sous-échantillonnage horizontal Cb et Cr

(c) Sous-échantillonnage horizontal et vertical Cb et Cr

FIGURE 2.4 – Composantes Y, Cb et Cr avec et sans sous-échantillonnage, pour une MCU de 2×2 blocs.

et Cr (figure 2.4(b)), un seul échantillon par chrominance Cb et Cr est utilisé pour deux pixels voisins d'une même ligne. Cet échantillon est calculé 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, qui peut aussi se décliner en vertical en suivant la même méthode.

Sous-échantillonnage horizontal et vertical La composante sous-échantillonnée comprend deux fois moins de blocs en horizontal et en vertical que la composante Y. La figure 2.4(c) illustre cette compression pour les composantes Cb et Cr, avec les facteurs d'échantillonnage 2×2, 1×1, 1×1. Comme la moitié de la résolution horizontale et verticale de la chrominance est éliminée pour Cb et Cr, un seul échantillon de chrominance Cb et Cr est utilisé pour quatre pixels. La qualité est visiblement moins bonne, mais sur un minitel ou un timbre poste, c'est bien suffisant !

Dans la littérature, on caractérise souvent 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.[6] Les notations suivantes font référence aux sous-échantillonnages les plus fréquemment utilisés :

4:4:4 Pas de sous-échantillonnage ;

4:2:2 Sous-échantillonnage horizontal (ou vertical) des composantes Cb et Cr ;

4:2:0 Sous-échantillonnage horizontal et vertical des composantes Cb et Cr.

2.7.2 Ordre d'écriture des blocs

L'image est découpée en MCUs, balayées de gauche à droite puis de haut en bas, la taille des MCUs étant donnée par les facteurs d'échantillonnage de la composante Y. L'ordonnancement des blocs dans le flux **suit toujours la même séquence**. La plupart du temps, l'ordre d'apparition des blocs est le suivant : ceux de la composante Y arrivent en premier, suivis de ceux de la composante Cb et enfin de la composante Cr. Mais on peut trouver des images dont l'ordre d'apparition des composantes est différente. Dans tous les cas, cet ordre est indiqué dans les entêtes de section décrits en B.2.5 et B.2.7.

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

L'ordre d'apparition des blocs d'une MCU pour une composante donnée dépend du nombre de composantes traitées.

Dans le cas d'une image couleur (3 composantes, Y, Cb, Cr), ils sont ordonnés de gauche à droite et de haut en bas. Prenons l'exemple d'une image de taille 16×16 pixels, composée de deux MCUs de 2×1 blocs et dont les composantes Cb et Cr sont sous-échantillonnées horizontalement. Dans le flux, on verra d'abord apparaitre les blocs des trois composantes de la première MCU ordonnés comme ci-dessus, à savoir $Y_0^0 Y_1^0 Cr^0 Cb^0$, puis ceux de la deuxième MCU ordonnés de la même façon, soit $Y_0^1 Y_1^1 Cr^1 Cb^1$. La séquence complète lue dans le flux sera donc $Y_0^0 Y_1^0 Cr^0 Cb^0 Y_0^1 Y_1^1 Cr^1 Cb^1$. La figure 2.5 tirée de la norme JPEG illustre cet ordonnancement.

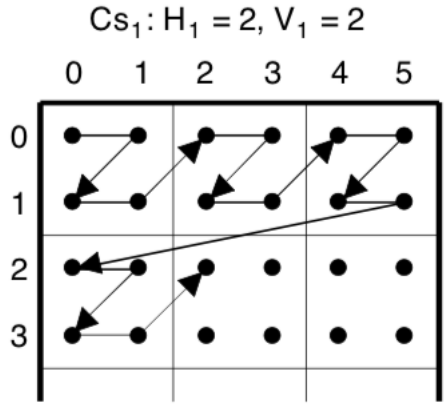


FIGURE 2.5 – Ordre d'apparition des blocs dans le bitstream, dans le cas d'une image couleur composée de MCUs de 2×2 blocs. Chaque point représente un bloc 8×8. Les barres verticales et horizontales délimitent les MCUs. La flèche indique l'ordre dans lequel les blocs sont écrits dans le bitstream.

Dans le cas d'une image noir-et-blanc (1 composante, Y) les blocs sont ordonnés dans le flux de la même façon que les pixels qu'ils représentent, c'est-à-dire séquentiellement de gauche à droite et de haut en bas pour toute l'image, et ce quelles que soient les dimensions de la MCU considérée, comme représenté en figure 2.6.

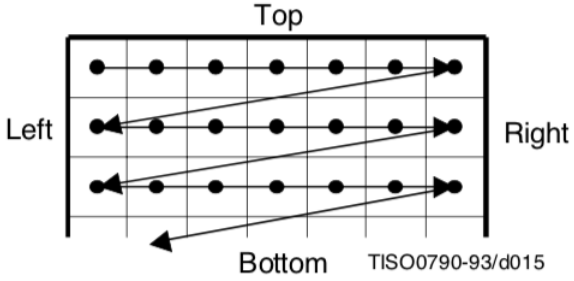


FIGURE 2.6 – Ordre d'apparition des blocs dans le bitstream, dans le cas d'une image noir-et-blanc. Chaque point représente un bloc 8×8. La flèche indique l'ordre dans lequel les blocs sont écrits dans le bitstream : on ne tient pas compte ici du découpage de l'image en MCUs pour ordonnancer les blocs.

2.8 Transformée en cosinus discrète (DCT)

Cette section traite du changement de domaine d'un bloc 8×8 à l'aide d'une transformée en cosinus discrète. Avant d'effectuer cette transformation, un offset de 128 est retranché à chaque valeur de pixel. On décale ainsi les valeurs des échantillons, initialement comprises entre 0 et 255, vers l'intervalle $[-128, 127]$. Ce décalage permet d'utiliser au mieux le codage par magnitude, avec des valeurs positives et négatives et de plus faible amplitude.

On applique ensuite une tranformée en cosinus discrète (DCT) qui convertit les informations spatiales en informations fréquentielles. C'est une formule mathématique « classique » de transformée. A chaque bloc de $n \times n$ pixels sont ainsi associées $n \times n$ fréquences. Dans sa généralité, la formule de la transformée en cosinus discrète (DCT, pour *Discrete Cosinus Transform*) pour les blocs de taille $n \times n$ pixels est :

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

Dans cette formule, S est le bloc spatial et Φ le bloc fréquentiel. Les variables x et y sont les coordonnées des pixels dans le domaine spatial et les variables i et j 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.

Attention à bien réfléchir aux types de données à utiliser quand il s'agira de porter cet algorithme en C.

2.9 Quantification et zig-zag

2.9.1 Zig-zag

L'opération de réorganisation en « zig-zag » permet de représenter un bloc 8×8 sous forme de vecteur 1D de 64 coefficients. Surtout, l'ordre de parcours présenté figure 2.7 place les coefficients des hautes fréquences en fin de vecteur. Comme ce sont ceux qui ont la plus forte probabilité d'être nul suite à la quantification (cf sous-section suivante), ceci permet d'optimiser la compression RLE décrite dans la section 2.10.

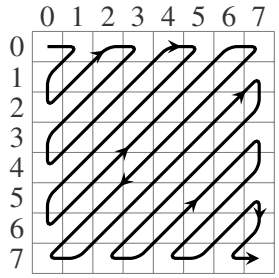


FIGURE 2.7 – Réordonnancement zig-zag.

2.9.2 Quantification

A l'encodage, 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.

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. On utilisera encore une fois des tables prédéfinies. Les tables de quantification fournies ici sont tirées du projet *The GIMP*. Elles sont utilisées par le mode de compression par défaut de l'application et vous sont distribuées dans le header `qtables.h`.

Les tables utilisées à l'encodage doivent être écrites dans le fichier JFIF. Avec une précision de 8 bits (c'est le cas du mode *baseline*), les coefficients sont des entiers non signés dont les valeurs sont comprises entre 0 et 255. Ces tables sont écrites dans le fichier au format zig-zag.

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.

Il est à noter que certains encodeurs disposent d'une option pour jouer sur le taux de compression de l'image générée. Par exemple, sur *The GIMP*, générer une image JPEG avec un taux de qualité de 100% aura pour effet d'utiliser *une seule table de quantification pour toutes les composantes, avec tous ses coefficients à 1*. Votre implémentation devra fonctionner avec les tables fournies, mais vous pouvez en ajouter d'autres.

2.9.3 Ordre des opérations

Les tables de quantification fournies dans le sujet sont stockées au format zig-zag. La norme JPEG impose aussi que les tables de quantification figurant dans l'entête JPEG soient aussi stockées au format zig-zag. On vous conseille donc d'effectuer l'opération de zig-zag sur un bloc avant de diviser ses coefficients par ceux de la table de quantification, pour éviter des opérations de réordonnancement inutiles.

2.10 Compression 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 2.10.3) ; puis les valeurs non nulles sont codées comme *différence* par rapport aux valeurs précédentes (voir 2.10.2) ; enfin, les symboles obtenus par l'application des deux codages précédents sont codés par un codage entropique de Huffman (2.10.1). Nous présentons dans cette section les trois codages, en commençant par détailler le codage de Huffman qui sera utilisé pour encoder les informations générées par les deux autres.

2.10.1 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.

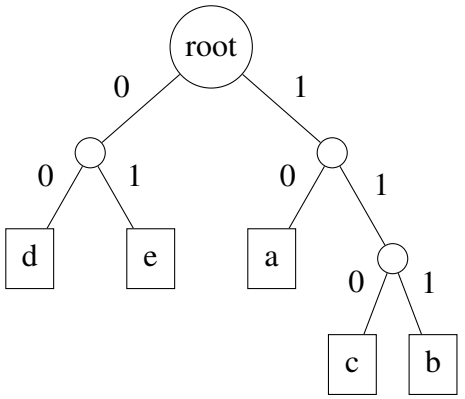
Ainsi, il n'est pas nécessaire d'avoir des « séparateurs » entre les symboles même s'ils n'ont pas tous la même taille, ce qui est ingénieux. [7] Par contre, le droit à l'erreur n'existe pas : si l'on perd un bit en route, tout le flux de données est perdu et l'on décodera n'importe quoi.

La construction des codes de Huffman n'entre pas dans le cadre initial de ce projet. Nous utiliserons en effet des tables de Huffman génériques, fournies en annexe par la norme, construites à partir de données statistiques récupérées sur une grande banque d'images. 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 nœud 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 avec un code de taille fixe (où tous les codes sont de même longueur), et donc la suite `decade` 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.

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

Dans le cas du JPEG, les tables de codage [8] 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. Autre particularité, la norme interdit les codes exclusivement composés de 1. L'arbre de Huffman présenté ci-dessus ne serait donc pas valide du point de vue du JPEG, et il faudrait stocker le symbole `b` sur une feuille de profondeur 4 pour construire un arbre conforme.

Ainsi, en mode baseline, l'encodeur supporte quatre tables :

— deux tables pour la luminance Y, une pour les composantes DC et une pour les composantes AC ;

— 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, expliqué dans les sections suivantes). 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.

Les tables de Huffman décrites dans la norme JPEG (section K.3 page 148) sont distribuées dans le header `htables.h`. Elles sont stockées sous la forme où elles apparaissent dans les entêtes JPEG, à savoir un tableau de 16 entrées contenant le nombre de symboles de chaque longueur et un tableau contenant les symboles encodés. Pour des raisons de compacité, on ne les reporte pas dans le sujet, rendez-vous dans la norme et/ou dans le fichier d'entête `htables.h` pour les récupérer.

2.10.2 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

8. Chacune représentant un arbre de Huffman.

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.

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

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.

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.

2.10.3 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 zig-zag (section 2.9) 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 2.10.2). 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 minimiser 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.

2.10.4 Byte stuffing

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 (voir annexe B). Afin de permettre aux décodeurs de faire un premier parcours du fichier en cherchant toutes les sections, ou de se rattraper lorsque 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 insérer ce `0x00` lors de l'encodage des données brutes, sans quoi le `0xff` correspondant sera faussement interprêté au décodage comme un marqueur de section.

Chapitre 3

Spécifications, modules fournis et organisation

Ce chapitre décrit ce qui est attendu, ce qui vous est fourni et comment l'utiliser, et quelques conseils sur la méthode à suivre pour mener à bien ce projet.

3.1 Spécifications

Les spécifications sont très simples : vous devez implémenter une application qui convertit une image PPM en une image au format JPEG :

- le nom de l'exécutable doit être `ppm2jpeg`
- il prend comme seul paramètre obligatoire le nom de l'image PPM à convertir, d'extension `.ppm` ou `.pgm`. Seules les images au format PPM P6 ou PGM P5 sont acceptées.
- en sortie une image au format JPEG sera générée, encodée en mode JFIF, baseline séquentiel, DCT, Huffman, de même nom que l'image d'entrée et d'extension `.jpg`.

`ppm2jpeg` prend aussi les paramètres optionnels suivants :

`--help` pour afficher la liste des options acceptées ;
`--outfile=sortie.jpg` pour redéfinir le nom du fichier de sortie ;
`--sample=h1xv1,h2xv2,h3xv3` pour définir les facteurs d'échantillonnage $h \times v$ des trois composantes de couleur.

Par exemple :

```
./ppm2jpeg shaun_the_sheep.ppm
```

génèrera le fichier `shaun_the_sheep.jpg`, alors que :

```
./ppm2jpeg --outfile=mouton.jpg --sample=2x2,1x1,1x1 shaun_the_sheep.ppm
```

génèrera le fichier `mouton.jpg` contenant l'image dont les composantes de chrominance ont été sous-échantillonnées.

C'est tout. Aucune trace particulière n'est attendue, hormis peut-être en cas d'erreur. Vous pouvez, si vous le souhaitez, ajouter des options, par exemple un mode verbose pour afficher des informations supplémentaires.

3.2 Organisation, démarche conseillée

Vous êtes bien entendu libres de votre organisation pour mener à bien votre projet, selon les spécifications présentées en section 3.1. Nous vous proposons tout de même une démarche générale, incrémentale, adaptée aux modules et aux images de test fournis. À vous de la suivre, de l'adapter ou de l'ignorer, selon votre convenance !

3.2.1 Résumé des étapes & difficultés

Pour résumer, les étapes de l'encodeur sont les suivantes :

1. Récupération des paramètres lus sur la ligne de commande et dans le fichier d'entrée PPM (dimensions de l'image, facteurs d'échantillonnage, ...) ;
2. Ecriture des différents marqueurs qui forment l'entête JPEG dans le fichier de sortie ;
3. Découpage de l'image en MCUs, récupération des échantillons RGB à partir du fichier d'entrée PPM ;
4. Encodage de chaque MCU :
   (a) Changement de représentation des couleurs : conversion RGB vers YCbCr ;
   (b) Compression des composantes Cb et Cr en cas de sous-échantillonnage (*downsampling*) ;
   (c) Compression de chaque bloc :
      i. Calcul de la transformée en cosinus discrète (DCT) ;
      ii. Réorganisation zig-zag ;
      iii. Quantification (division par les tables de quantification) ;
      iv. Compression AC/DC et écriture dans le flux ;

Une estimation de la difficulté de ces différentes étapes est la suivante :

Etape	Difficulté pressentie	Module
Ecriture bit à bit dans le flux	☆☆☆☆	`bitstream`
Ecriture entête JPEG	☆☆☆☆	`jpeg_writer`
Gestion des tables de Huffman	☆☆☆☆☆	`huffman`
Gestion des paramètres sur la ligne de commande	☆☆ à ☆☆☆	
Découpage de l'image en MCUs, des MCUs en blocs	☆☆☆	
Lecture fichier PPM	☆☆ à ☆☆☆☆	
Conversion RGB vers YCbCr	☆	
Downsampling	☆☆☆☆	
DCT	☆☆	
Zig-zag	☆☆	
Quantification	☆	
Gestion complète de l'encodage	☆☆ à ☆☆☆☆☆	

3.2.2 Modules fournis

Trois modules sont fournis dans ce projet, sous forme d'une spécification (fichier `.h`) et d'un fichier objet binaire compilé (extension `.o`). Un fichier d'entête contenant la définition des tables de quantification est aussi fourni. Le code source n'est pas distribué.

— le module `jpeg_writer` fournit des fonctions pour accéder aux paramètres définis dans les sections d'un fichier JPEG décrites en annexe B. Il répond aux besoins de l'étape 2.
— le module `huffman` permet de représenter, écrire, utiliser et détruire les tables de Huffman du fichier JPEG. Il sera utilisé à l'étape 4(c)iv pour encoder la magnitude des coefficients DC et les symboles RLE des coefficients AC, à partir des tables de Huffman correspondantes ;
— le module `bitstream` permet d'écrire des bits dans le flux de sortie, et non des octets comme avec une écriture standard. Il sera aussi utilisé à l'étape 4(c)iv pour écrire des séquences de bits, représentant les indices des coefficients DC et AC dans leur classe de magnitude. Ce module est également utilisé par `jpeg_writer` pour écrire dans l'entête JPEG.
— le fichier `qtables.h` contient la définition des tables de quantification génériques utilisées pour ce projet, tirées du logiciel `gimp`.

La spécification détaillée de ces modules est fournie en annexe C.

3.2.3 Encodage d'abord, réécriture des modules ensuite

À terme, vous devrez bien sûr implanter **toutes** les étapes de l'encodeur. Mais attaquer ce projet *from scratch*, sans aucune ressource, peut être difficile pour une majorité d'entre vous... et c'est bien normal ! C'est la raison pour laquelle trois des modules les plus difficiles et qui sont nécessaires dès le début de l'encodage vous sont fournis.

La démarche *fortement* conseillée est donc la suivante :

1. Commencez par travailler sur la **partie encodage** proprement dite, à savoir les étapes 3 et 4. Vous aurez simplement besoin d'**utiliser les modules fournis**, ce qui nécessite d'avoir bien compris le principe de l'encodage mais reste beaucoup plus simple que de les écrire. L'étape 1 peut être dans un premier temps **réduite à sa plus simple expression**, en implémentant seulement la récupération du nom du fichier d'entrée PPM. Les autres paramètres de l'encodeur pourront être codés en dur en attendant une implémentation complète de cette étape.
2. Dans un second temps seulement, vous pourrez (devrez) réécrire vous-même les différents modules pour remplacer notre implémentation par la vôtre.

3.2.4 Progression incrémentale sur les images à encoder

Le second point porte sur les images à encoder. Bien évidemment, votre encodeur devra *in fine* être capable de traiter toutes les images PPM qui répondent aux spécifications du sujet. Mais vouloir résoudre d'emblée le problème complet est risqué : il est possible que vous ayez pu coder la quasi-totalité des étapes mais sans avoir pu les valider ou finir de les intégrer. Vous aurez alors fourni un travail conséquent et écrit de magnifiques « bouts de programme », mais n'aurez pas écrit un encodeur qui fonctionne !

Nous vous conseillons donc d'adopter une approche dite ***incrémentale*** :

— L'objectif est d'obtenir le plus rapidement possible un encodeur **fonctionnel**, même s'il ne permet de traiter que des images très simples ;

— Chacune des étapes sera ensuite complétée (voire totalement reprise) pour couvrir des spécifications de plus en plus complètes.

En plus d'être efficace et rationnelle, cette approche permet d'avoir toujours un programme fonctionnel, même incomplet, à présenter à votre client (nous), à tout instant. Imaginez que le rendu soit subitement avancé de deux jours, [1] et bien vous rendrez votre projet dans l'état courant, sans (trop de) stress ; et hop. Et votre client sera satisfait, ce qui est bon pour lui et au final pour vous [2] !

Plusieurs images de tests sont fournies, de « complexité » croissante (voir table 3.1). Il est conseillé de travailler sur ces images dans l'ordre suggéré, permettant une progression graduelle d'un encodeur simple vers celui capable de traiter des images quelconques.

Image	Caractéristiques	Simplifications / progression
`invader`	Niveaux de gris (1 composante) 8×8 (un seul bloc)	Encodage MCU très simple, pas d'*upsampling*, PPM simplifié
`gris`	Niveaux de gris, 320×320	Plusieurs blocs, pas de troncature
`bisou`	Niveaux de gris, 585×487	Plusieurs blocs, troncature à droite et en bas
`zig-zag`	YCbCr, 480×680	Couleur, pas de troncature
`thumbs, horizontal, vertical, Shaun the sheep`	439×324, 367×367 704×1246, 300×225	Couleur, tronquée droite et/ou bas
`complexite`	Niveaux de gris, 2995×2319	C'est looonnnng...

TABLE 3.1 – Liste des images de test fournies, classées par ordre de « complexité ».

Ces images doivent vous aider à valider certaines parties du projet, **mais ne seront pas suffisantes pour permettre le test unitaire de certaines parties de l'encodeur**. Il sera donc nécessaire d'ajouter vos propres images à cette base de tests, par exemple conçues de toute pièce avec `gimp`. En plus de vous aider lors de la mise au point de certaines fonctionnalités de l'encodeur, vous pourrez vous appuyer sur ces images *"maison"* lors de la soutenance pour démontrer la robustesse de votre implémentation.

3.2.5 Découpage en modules & fonctions, spécifications

Par rapport à la plupart des TPs réalisés cette année, une des difficultés de ce projet est sa *taille*, *i.e.,* la quantité des tâches à réaliser. Une autre est que c'est principalement à vous de définir *comment* vous allez résoudre le problème posé. Avant de partir tête baissée dans du codage, il est essentiel de bien définir un découpage en *modules* et en *fonctions* pour les différents éléments à réaliser. Ils

1. Vos enseignants sont parfois très joueurs !
2. Au-delà de ce projet, ceci sera surtout valable dans votre vraie vie d'ingénieur, si si ! Pensez à nous remercier le moment venu.

doivent être clairement **spécifiés** : rôle, structures de données éventuelles, fonctions proposées et leur signature précise.

Dans le cadre de ce projet en équipe, vous serez amenés à *utiliser* les modules programmés par vos collègues. Il est donc nécessaire de bien comprendre leur usage, même si vous ne connaissez ou ne maîtrisez pas leur contenu. Une bonne spécification est donc fondamentale pour que la mise en commun des différentes parties du projet se fasse sans trop de heurts (vous verrez, ce n'est pas toujours simple...).

Prenons pour exemple l'étape DCT. Il est naturel[3] d'écrire une fonction spécifique qui réalise cette opération uniquement. Beaucoup de questions sont à soulever :

— quel nom donner à cette fonction ?
— dans quels fichiers sera-t-elle déclarée (`.h`) et définie (`.c`) ?
— quel est son rôle ? Réalise-t-elle le calcul sur un seul bloc, sur plusieurs ?
— quels sont ses paramètres : un bloc d'entrée et un de sortie ? Un seul bloc qui est modifié au sein de la fonction ? Faut-il allouer de la mémoire ? ...
— comment est représenté un bloc en mémoire : tableau 1D de 64 valeurs ou tableau 2D de taille 8×8 ?[4] Adresse dans un tableau de plus grande taille ? ...
— Quel est le type des éléments d'un bloc à ce stade de l'encodage ?
— ...

Attention, ce travail est difficile ! Mais il est vraiment fondamental, même si vous serez certainement amenés à modifier ces spécifications au fur et à mesure de l'avancement dans le projet (c'est normal).

Dans la mesure du possible (pas toujours facile), chaque module devra être testé de manière autonome c'est-à-dire hors contexte de son utilisation dans l'encodeur. Il s'agit de tester avec des entrées contrôlées et de vérifier que les sorties sont bien conformes à la spécification. Ceci sera facile à réaliser pour certains modules, comme `bitstream` ou `jpeg_writer` ou certaines étapes « simples » de l'encodage (zig-zag, ...). Sinon vous devrez tester les étapes séquentiellement (les sorties de l'une étant les entrées de la suivante), en comparant notamment les données à l'aide des outils `ppm2blabla` et `jpeg2blabla` distribués (voir section 3.3).

3.3 Outils et traces pour la mise au point

Cette section introduit quelques outils disponibles pour vous aider à mettre au point votre encodeur.

`ppm2blabla`

Ce programme est en fait une version « bavarde » de `ppm2jpeg`, réalisée par nos soins. En plus de décoder l'image, cet utilitaire crée également un fichier d'extension `.bla` qui fournit les valeurs numériques de tous les blocs de chaque MCU, étape après étape (après zig-zag, après quantication, etc.). Un exemple est donné en figures 3.1 et 3.2. Ceci pourra s'avérer très utile (sur des images de taille réduite) pour valider votre encodeur étape après étape, si nécessaire. Il se peut que les valeurs numériques diffèrent légèrement (calcul flottant, pas les mêmes arrondis...), mais vous pourrez vérifier que les images encodées sont bien les mêmes.

3. Si ce n'est pas le cas encore, ça devrait !
4. Moyennant une petite gymnastique sur les indices, cette une représentation 1D simplifiera beaucoup l'expression et la manipulation des différentes fonctions à implémenter !

`jpeg2blabla`

Ce programme est le symétrique du précédent : il prend en entrée une image jpeg, la décode et génère au passage un fichier `.blabla` qui affiche le même type d'informations que `ppm2blabla`, mais obtenues cette fois lors du décodage de l'image. `jpeg2blabla` vous sera en priorité utile pour valider le contenu de l'entête jpeg que vous génèrerez lors de l'implémentation de votre `jpeg_writer`. En effet, la commande `./jpeg2blabla -v [image.jpg]` affiche sur la sortie standard les informations lues dans l'entête du fichier `image.jpg`. Un exemple de sortie de cet outil est donné en figure 3.3.

`hexdump`

Cette application, en particulier avec l'option `-C`, affiche de manière textuelle le contenu octet par octet d'un fichier (un exemple de sortie est disponible figure 2.2). Dans ce projet, c'est très utile pour regarder les données d'un fichier JPEG, comprendre sa structure et vérifier que les données écrites sont correctes (notamment lorsque vous réécrirez le module `jpeg_writer`). Vous pourrez également l'utiliser pour tester votre module `bitstream`, toujours pour regarder le contenu bit à bit d'un fichier et vérifier que les bits écrits correspondent.

`identify`

C'est un utilitaire de la suite `ImageMagick`, qui décrit le format et les caractéristiques d'une image. À utiliser avec notamment l'option `-verbose`.

`gimp`

C'est un des outils de manipulation d'images les plus connus. Il permet entre autres de générer des fichiers au format PPM (P5 et P6) supportés par votre encodeur. Il est aussi bien entendu capable d'afficher des images JPEG ou PPM, et les messages d'erreur qu'il renvoie lors de l'ouverture d'un fichier JPEG mal formé sont parfois utiles. Pour l'affichage seul, préférez **`eog`** (Eye Of Gnome), moins gourmand en ressources et plus rapide à invoquer.

3.4 Extensions possibles

Si vous lisez cette section, c'est que vous êtes venus à bout de votre encodeur pour toutes les configurations d'images JPEG. [5] Félicitations !

3.4.1 Amélioration de la compression

L'un des intérêts du format JPEG est la réduction substantielle de la place occupée par l'image, la taille du fichier généré étant la plupart du temps bien inférieure à celle du fichier PPM d'entrée. Si l'application de sous-échantillonnage des composantes de couleur reste un facteur prépondérant dans la réduction de l'espace mémoire occupé par l'image, les étapes de compression des symboles fréquents par l'utilisation de codes de Huffman, et d'élimination de certains échantillons par leur mise à zéro lors de la phase de quantification, permettent aussi de gagner en taux de compression. Cette

5. Si ce n'est pas le cas, commencez par finir votre encodeur avant d'attaquer les extensions !

```
$ ./ppm2blabla ./images/samples/invader.pgm
$ cat ./images/samples/invader.bla
ppm_check_header:
presence de P5 ou P6 dans l'entete... PASS
presence des dimensions HxV dans l'entete... 8x8
presence de 255 dans l'entete... PASS

[MCU #0] Valeurs RGB initiales:
000000 000000 000000 0000ff 0000ff 000000 000000 000000
000000 000000 0000ff 0000ff 0000ff 0000ff 000000 000000
000000 0000ff 0000ff 0000ff 0000ff 0000ff 0000ff 000000
0000ff 0000ff 000000 0000ff 0000ff 000000 0000ff 0000ff
0000ff 0000ff 0000ff 0000ff 0000ff 0000ff 0000ff 0000ff
000000 000000 0000ff 000000 000000 0000ff 000000 000000
000000 0000ff 000000 0000ff 0000ff 000000 0000ff 000000
0000ff 000000 0000ff 000000 000000 0000ff 000000 0000ff

[MCU #0] Conversion RGB -> YCbCr:
[Y]:
  00 00 00 ff ff 00 00 00
  00 00 ff ff ff ff 00 00
  00 ff ff ff ff ff ff 00
  ff ff 00 ff ff 00 ff ff
  ff ff ff ff ff ff ff ff
  00 00 ff 00 00 ff 00 00
  00 ff 00 ff ff 00 ff 00
  ff 00 ff 00 00 ff 00 ff

[MCU #0] Downsampling:
[Y]:
  00 00 00 ff ff 00 00 00
  00 00 ff ff ff ff 00 00
  00 ff ff ff ff ff ff 00
  ff ff 00 ff ff 00 ff ff
  ff ff ff ff ff ff ff ff
  00 00 ff 00 00 ff 00 00
  00 ff 00 ff ff 00 ff 00
  ff 00 ff 00 00 ff 00 ff
```

FIGURE 3.1 – [1/2] Extrait d'un fichier `.bla` généré par `./ppm2blabla` sur l'image `invader.pgm`. On retrouve d'abord l'entête jpeg, puis pour chaque MCU de l'image, les valeurs RGB initiales, les valeurs après conversion en YCbCr, les valeurs après application du sous-échantillonnage, ...

```
[MCU #0] DCT:
[Y]:
  007b 0000 fee3 0000 0000 0000 ffeb 0000
  fffa 0000 feda 0000 006a 0000 ff7d 0000
  feb2 0000 ff8b 0000 0045 0000 0099 0000
  ff36 0000 ffa0 0000 0018 0000 0003 0000
  007f 0000 011c 0000 00ff 0000 0014 0000
  ffa6 0000 0013 0000 007d 0000 fe1e 0000
  ff76 0000 001a 0000 ff59 0000 00f4 0000
  00dc 0000 ffa8 0000 ffb9 0000 ff3c 0000

[MCU #0] ZZ:
[Y]:
  007b 0000 fffa feb2 0000 fee3 0000 feda
  0000 ff36 007f 0000 ff8b 0000 0000 0000
  006a 0000 ffa0 0000 ffa6 ff76 0000 011c
  0000 0045 0000 ffeb 0000 ff7d 0000 0018
  0000 0013 0000 00dc 0000 001a 0000 00ff
  0000 0099 0000 0000 0003 0000 007d 0000
  ffa8 0000 ff59 0000 0014 0000 0000 fe1e
  0000 ffb9 0000 00f4 0000 0000 ff3c 0000

[MCU #0] Quantification:
[Y]:
  0018 0000 fffe ffbe 0000 ffe9 0000 fff0
  0000 ffce 001f 0000 fff2 0000 0000 0000
  001a 0000 ffed 0000 fff9 fff8 0000 0010
  0000 000d 0000 fffe 0000 fffb 0000 0001
  0000 0002 0000 000c 0000 0000 0000 000b
  0000 000d 0000 0000 0000 0000 0003 0000
  fffb 0000 fff9 0000 0000 0000 0000 fff0
  0000 fffe 0000 0008 0000 0000 fffa 0000

[MCU #0] Compression AC/DC puis écriture dans le flux.
```

FIGURE 3.2 – [2/2] ... puis pour chaque composante de couleur, pour chaque bloc de cette composante, les valeurs de chaque échantillon après application de la DCT, du zig-zag et de la quantification. Ça pique les yeux au début, mais après quelques jours vous apprécierez !

section vous donne des pistes sur la manière d'améliorer votre projet vis-à-vis du taux de compression des images JPEG générées.

Arbres de Huffman spécifiques à une image

La norme JPEG prévoit l'utilisation d'arbres de Huffman spécifiques à une image, contrairement à la version de base implémentée dans ce projet qui utilise des arbres "génériques" construits à partir de statistiques récoltées sur des banques d'images couleur. L'objectif de cette extension consiste à générer les arbres DC et AC permettant d'encoder au mieux les symboles de l'image traitée, à savoir attribuer les codes les plus courts aux symboles qui apparaissent le plus souvent dans l'image en cours d'encodage.

Sélecteur de niveau de compression à l'exécution

La plupart des encodeurs JPEG définissent différents niveaux de compression, qui correspondent en fait à l'utilisation de tables de quantification plus ou moins "aggressives". Pour mémoire, l'étape de quantification consiste à diviser les valeurs des échantillons d'un bloc par les valeurs d'une table, la table de quantification, dans le but de faire apparaitre des 0 qui seront facilement compressés par la suite. Si cette table ne contient que des 1, la valeur des échantillons reste inchangée, l'image est de meilleure qualité visuelle mais le taux de compression est faible. A l'inverse, si la table comprend plusieurs coefficients très supérieurs à 1, beaucoup de valeurs d'échantillons du bloc prendront la valeur 0, la qualité de l'image sera dégradée mais la compression sera bien meilleure. Par exemple, Photoshop permet de spécifier un niveau de qualité représenté par un entier compris entre 1 et 12. A chaque valeur du taux de compression correspond des tables de quantification, dont une liste non-exhaustive peut être trouvée ici : https://www.impulseadventure.com/photo/jpeg-quantization.html.

3.4.2 Amélioration du temps d'exécution de l'encodeur

Afin d'améliorer le temps d'exécution de votre encodeur, commencez par étudier l'impact des optimisations à la compilation. Pour ce faire, jouez avec l'option `-O` du compilateur pour générer différentes versions utilisant différents niveaux d'optimisation (`-O0`, `-O1`, jusqu'à `-O3`). Pensez à toujours vérifier l'intégrité des images générées, les optimisations appliquées à partir du niveau 3 ne garantissant pas toujours l'exactitude des résultats numériques (!).

On utilisera ensuite un outil de *profiling* pour détecter dans quelle partie de l'encodeur on passe le plus de temps. Vous pouvez par exemple utiliser l'outil GNU `gprof` :

— recompilez votre programme en ajoutant l'option de compilation `-pg` (pour compiler les objets ET pour l'édition de liens) ;
— exécutez votre programme normalement. Un fichier `gmon.out` a normalement été crée ;
— étudiez le résultat à l'aide de la commande : `gprof ./ppm2jpeg gmon.out`. Vous trouverez en particulier la ventilation du temps d'exécution sur les différentes fonctions de votre programme. Sympa, non ?

Sauf surprise, il est très probable qu'au moins 80% du temps soit consommé par l'étape de DCT. Quand on regarde de plus près l'algorithme présenté en 2.8, on se rend compte qu'on recalcule plusieurs fois les mêmes valeurs de cosinus. Une première optimisation de votre encodeur consiste donc à précalculer et stocker tous les cosinus nécessaires. Pour aller plus loin, il faudra s'attaquer à l'algorithme en lui-même pour réduire le nombre d'opérations, les multiplications en particulier. Comme

vous l'avez vu en cours d'« Algorithmique et structures de données », il existe des méthodes de type « Diviser pour régner » pour écrire une version efficace de la transformée de Fourier (*Fast Fourier Transform*, FFT) en $O(n \log n)$ bien plus rapide que la version quadratique proposée en 2.8. Des versions optimales existent en plus dans le cas particulier de la DCT sur un bloc 8×8. Reste plus qu'à les trouver, les comprendre, les implémenter, les valider et se congratuler.

Si vous vous ennuyez après ceci, venez nous voir ! On peut encore trouver des tas de choses intéressantes à faire, comme implémenter d'autres pans de la norme JPEG, ou concevoir un décodeur.

3.5 Informations supplémentaires

Voici quelques documents ou pages Web qui vous permettront d'aller un peu plus loin en cas de manque d'information, ou simplement pour approfondir votre compréhension du JPEG si vous êtes intéressés.

1. En premier lieu, toute l'information sur la norme est évidemment disponible dans le document `ISO/IEC IS 10918-1 | ITU-T Recommendation T.81` disponible ici : `http://www.w3.org/Graphics/JPEG/itu-t81.pdf`. La norme ne se limite pas à notre spécification (mode baseline séquentiel uniquement), mais fondamentalement tout y est !
2. `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.
3. Pour une compréhension plus poussée du sous-échantillonnage des chrominances, consulter `http://dougkerr.net/pumpkin/articles/Subsampling.pdf`
4. Pour les informations relatives au format JFIF, aller voir du côté de `http://www.ijg.org/`

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 proposons. L'examen du code lors de la soutenance sera sans pitié pour toute forme de plagiat. Mais surtout assimiler ce type de code vous demandera au moins autant d'effort que de programmer vous-même votre propre encodeur !

```
[SOI]   marker found
[APP0]  length 16 bytes
        JFIF application
        other parameters ignored (9 bytes).
[DQT]   length 67 bytes
                quantization table index 0
                quantization precision 8 bits
                quantization table read (64 bytes)
[SOF0]  length 11 bytes
        sample precision 8
        image height 225
        image width 300
        nb of component  1
        component Y
                id 1
                sampling factors (hxv) 1x1
                quantization table index 0
[DHT]   length 31 bytes
                Huffman table type DC
                Huffman table index 0
                total nb of Huffman codes 12
[DHT]   length 181 bytes
                Huffman table type AC
                Huffman table index 0
                total nb of Huffman codes 162
[SOS]   length 8 bytes
        nb of components in scan 1
        scan component index 0
                associated to component of id 1 (frame index 0)
                associated to DC Huffman table of index 0
                associated to AC Huffman table of index 0
        other parameters ignored (3 bytes)

*** STOPPED SCAN, bitstream at the beginning of Scan raw compressed data ***
... (image decompression) ...
*** Scan finished
[EOI]   marker found
```

FIGURE 3.3 – Trace de `./jpeg2blabla -v grey_shaun_the_sheep.jpeg`. L'image est ici en niveaux de gris, donc avec une seule composante de couleur (la luminance Y) et pas de sous-échantillonnage, et utilise une seule table de quantification et deux tables de Huffman (une par composante AC/DC) définies dans des sections DHT séparées.

Chapitre 4

Travail demandé

4.1 Objectif

L'objectif de ce projet est de développer un encodeur `ppm2jpeg` répondant aux spécifications édictées en section 3.1.

Même si une approche incrémentale, progressive, est fortement conseillée, il est réellement attendu que vous fournissiez un encodeur complet ! Il devra être capable d'encoder les images PGM et PPM fournies, en supportant au minimum les sous-échantillonnages les plus répandus, cités en fin de section 2.7.1 (`4:4:4`, `4:2:2`, `4:2:0`).

Enfin, les modules fournis initialement devront avoir été réécrits complètement par votre équipe.

4.2 Rendu

Le rendu consistera en une unique archive `.tar.gz` qui devra être déposée sur `Teide`. La date limite de rendu est fixée au **mercredi 30 mai 2018 à 18h**.

Cette archive devra contenir :
- la totalité de votre code source, fichiers `.h` et `.c` ;
- un fichier `Makefile` permettant de compiler l'application en tapant simplement `make` ;
- toutes les données de test supplémentaires que vous aurez pu utiliser (scripts, tests unitaires,...). N'ajoutez pas les fichiers image à l'archive, ce serait bien trop volumineux. Votre banque d'image de tests sera de toute façon utilisée lors de la démonstration en soutenance. Vous pouvez si vous le souhaitez uploader vos images tests quelque part et indiquer où les télécharger dans un fichier `README` intégré à l'archive ;
- tout autre élément pertinent de votre projet ;

Quelques remarques supplémentaires :
- le code devra fonctionner dans les salles machines de l'Ensimag ;
- il vous est demandé de respecter les conventions de codage du noyau Linux, déjà utilisées dans la phase de préparation au C ;
- utilisez également les types entiers C99 définis dans `stdint.h` et `stdbool.h` ;
- ne considérez pas le `Makefile` comme une contrainte mais comme une aide ! Vous devez le mettre en place et l'utiliser dès le début et l'utiliser tout au long du projet. 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.

— il n'est PAS demandé de rapport écrit. Mais si vous avez rédigé des documents de travail ou schémas qui vous paraissent faciliter la compréhension de votre projet, n'hésitez pas à les joindre à l'archive rendue.

4.3 Soutenance

Les soutenances auront lieu les **jeudi 31 mai après-midi** et **vendredi 1er juin matin**. Un planning d'inscription sera mis en ligne sur `Teide`.

Voici quelques éléments d'information, qui pourront être précisés si besoin d'ici la fin du projet :

— tous les étudiants de l'équipe doivent être présents ;
— la durée de votre soutenance est de 40 minutes, en présence d'un enseignant ;
— vous aurez 10-15 minutes pour nous présenter votre projet : ce qui fonctionne, les limites, comment vous avez conçu et testé votre code, les points importants/spécifiques de votre implémentation, etc. À vous de « vendre » votre travail de la manière qui vous parait adéquate !
— la suite sera passée avec l'enseignant pour rentrer plus en détails dans votre projet, réaliser des tests supplémentaires, regarder le code, etc.
— si certains le souhaitent, vous pouvez préparer quelques transparents, schémas, etc. sur lesquels appuyer votre présentation. Mais ce n'est pas demandé et pas forcément utile, encore une fois c'est à vous de voir.
— les soutenances auront lieu dans les salles machines de l'Ensimag. Il n'est pas possible d'utiliser un ordinateur portable, votre projet devra fonctionner sur les postes Linux de l'école.
— l'évaluation sera faite à partir de l'archive rendue le mercredi soir. Toute modification ultérieure ne sera pas prise en compte. Vous pouvez par contre prévoir une banque d'images de test sur clé USB ou récupérable facilement depuis votre compte.

Tout est dit, il ne nous reste plus qu'à vous souhaiter bon courage !

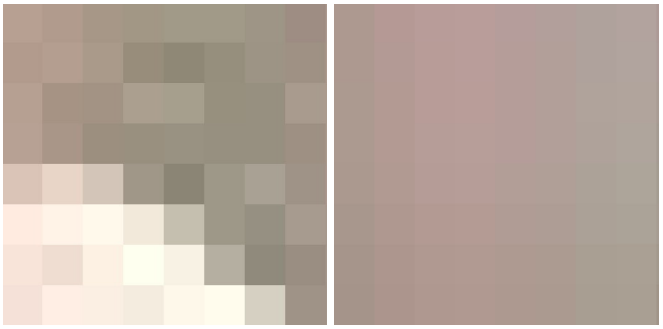
Annexe A

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. À étudier en parallèle du chapitre 2 !

A.1 MCU en RGB

Pour cet exemple, on suppose une MCU de taille 16×8, 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 :

Y

a6	a0	9a	98	9a	9a	96	91	9e	a1	a4	a5	a4	a4	a5	a7
a0	a3	9d	8e	88	8f	95	94	9e	a1	a4	a5	a4	a4	a5	a7
a5	97	96	a1	9f	90	90	9e	9e	a1	a4	a5	a4	a4	a6	a7
a6	9a	91	91	92	90	90	93	9e	a1	a4	a5	a4	a4	a5	a7
c9	d9	c8	98	85	98	a2	95	9d	a0	a3	a4	a3	a3	a4	a6
f0	f5	f9	ea	bf	98	90	9d	9b	9e	a1	a2	a1	a1	a3	a4
e9	e1	f3	fd	f2	af	8a	90	99	9c	9f	a0	9f	9f	a1	a2
e6	f2	f1	ed	f8	fb	d0	95	98	9b	9e	9f	9e	9e	a0	a1

Cb

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

Cr

8d	8c	89	87	85	85	86	89	8b	8e	8e	8e	8c	8a	87	88
8d	8c	89	87	85	85	86	88	8b	8d	8e	8e	8c	8a	87	87
8c	8b	89	87	85	85	85	88	8a	8d	8e	8e	8c	8a	87	87
8c	8b	88	86	85	85	85	88	8a	8d	8d	8d	8c	8a	87	86
8c	8b	88	86	84	84	85	87	8a	8c	8d	8d	8b	89	86	85
8b	87	84	86	84	84	84	87	89	8c	8d	8d	8b	89	86	85
8b	8a	87	81	84	84	84	87	89	8c	8c	8c	8b	89	86	86
8b	89	87	85	84	83	84	87	89	8c	8c	8c	8b	89	86	86

A.3 Sous échantillonnage

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

Y

a6	a0	9a	98	9a	9a	96	91
a0	a3	9d	8e	88	8f	95	94
a5	97	96	a1	9f	90	90	9e
a6	9a	91	91	92	90	90	93
c9	d9	c8	98	85	98	a2	95
f0	f5	f9	ea	bf	98	90	9d
e9	e1	f3	fd	f2	af	8a	90
e6	f2	f1	ed	f8	fb	d0	95

9e	a1	a4	a5	a4	a4	a5	a7
9e	a1	a4	a5	a4	a4	a5	a7
9e	a1	a4	a5	a4	a4	a6	a7
9e	a1	a4	a5	a4	a4	a5	a7
9d	a0	a3	a4	a3	a3	a4	a6
9b	9e	a1	a2	a1	a1	a3	a4
99	9c	9f	a0	9f	9f	a1	a2
98	9b	9e	9f	9e	9e	a0	a1

Cb

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

Cr

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

A.4 DCT : passage au domaine fréquentiel

La DCT est ensuite appliquée à chacun des blocs de données, après avoir soustrait 128 [1] aux valeurs. Les basses fréquences sont en haut à gauche et les hautes fréquences en bas à droite. On représente maintenant des *entiers signés 16 bits*. Ainsi, `0xffff` représente la valeur `-1` ici, et pas `65535` !

Y

```
0183 0086 ffe7 fffd ffff fffe 0000 0002
ff2e ffa9 0041 fffd 0002 0002 ffff fffd
004f fff8 ffca 001c fffe fffe 0000 0001
000e 003b 000f ffdd 0000 ffff 0000 0000
fff3 ffdd 0013 001f ffd9 0002 fffe fffd
fff0 0000 ffe0 0001 002d fffc 0001 0003
001f 0000 fffc ffff 0003 0003 fffe fffc
ffff 0000 0001 0000 fffd fffd 0000 0002
```

```
010d ffee fffa fff5 ffff 0000 ffff ffff
0010 0000 ffff ffff 0000 ffff 0000 ffff
fff8 ffff 0000 0000 ffff 0000 ffff 0000
0000 0000 0000 0000 0000 ffff 0000 ffff
0000 0000 ffff ffff 0000 ffff 0000 ffff
ffff ffff 0000 0000 ffff 0000 ffff 0000
0000 ffff 0000 0000 ffff 0000 ffff 0000
ffff ffff 0000 0000 ffff 0000 ffff 0000
```

Cb

```
ffbe fff6 0000 ffff ffff 0000 ffff ffff
ffff fff9 0000 0000 ffff 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 ffff ffff 0000 0000 0000 0000 ffff
0000 0000 0000 0000 ffff 0000 0000 ffff
0000 ffff 0000 0000 ffff 0000 0000 0000
0000 0000 0000 0000 ffff 0000 ffff ffff
ffff 0000 ffff 0000 ffff ffff ffff ffff
```

Cr

```
0044 fff9 0000 0016 ffff 0000 0000 0001
0005 0001 0000 0000 ffff ffff ffff ffff
0000 ffff 0000 ffff 0001 ffff 0000 ffff
ffff 0000 ffff 0000 0000 0000 0000 0000
0000 0001 0000 0000 0000 ffff ffff ffff
ffff ffff 0000 ffff 0000 0000 0000 0000
ffff ffff 0000 ffff ffff ffff ffff ffff
0000 0000 0000 0000 0000 ffff 0000 ffff
```

1. en fait 2^{P-1}, avec ici une précision $P = 8$

A.5 Zig-zag

On réordonne ensuite les blocs en zig-zag. Cette étape permettra de regrouper les coefficients nuls en "fin" de bloc après la phase de quantification. Pour des questions de praticité, on effectue cette réorganisation avant, puisque *les tables de quantification sont stockées au format zig-zag dans l'entête JPEG*. On pourrait bien entendu fusionner ces deux étapes (réordonnancement zig-zag + quantification) en une seule opération, comme indiqué sur la figure 2.1, mais `ppm2blabla` les considère comme deux étapes différentes pour vous aider à les débugger indépendamment.

Y

```
0183 0086 ff2e 004f ffa9 ffe7 fffd 0041
fff8 000e fff3 003b ffca fffd ffff fffe
0002 001c 000f ffdd fff0 001f 0000 0013
ffdd fffe 0002 0000 0002 ffff fffe 0000
001f ffe0 0000 ffff 0000 fffc 0001 ffd9
ffff 0000 fffd 0001 0000 0002 002d ffff
0001 0000 0003 fffc fffe 0000 fffd 0001
0003 fffd fffd fffe 0003 fffc 0000 0002
```

```
010d ffee 0010 fff8 0000 fffa fff5 ffff
ffff 0000 0000 0000 0000 ffff ffff 0000
0000 0000 0000 0000 ffff 0000 ffff ffff
0000 ffff ffff ffff ffff 0000 0000 0000
ffff 0000 ffff ffff ffff 0000 0000 0000
ffff ffff ffff 0000 0000 ffff ffff 0000
0000 0000 ffff 0000 0000 ffff ffff ffff
0000 ffff 0000 ffff 0000 0000 ffff 0000
```

Cb

```
ffbe fff6 ffff 0000 fff9 0000 ffff 0000
0000 0000 0000 ffff 0000 0000 ffff 0000
ffff 0000 ffff 0000 0000 0000 ffff 0000
0000 0000 0000 ffff ffff 0000 0000 0000
0000 0000 0000 ffff 0000 0000 0000 ffff
0000 0000 0000 0000 0000 0000 ffff 0000
ffff 0000 ffff 0000 0000 ffff ffff 0000
0000 ffff ffff ffff 0000 ffff ffff ffff
```

Cr

```
0044 fff9 0005 0000 0001 0000 0016 0000
ffff ffff 0000 0000 0000 0000 ffff 0000
ffff ffff ffff 0001 ffff ffff ffff 0000
0000 0001 ffff 0000 0001 ffff ffff 0000
0000 0000 ffff 0000 0000 0000 ffff 0000
0000 0000 ffff ffff 0000 ffff 0000 ffff
0000 0000 ffff 0000 ffff 0000 ffff 0000
ffff 0000 ffff ffff 0000 ffff 0000 ffff
```

A.6 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 :

Y

```
004d 002d ffba 0010 fff4 fffe 0000 0004
fffe 0004 fffd 000a fff9 0000 0000 0000
0001 0007 0003 fffb ffff 0002 0000 0001
fff7 0000 0000 0000 0000 0000 0000 0000
0006 fffb 0000 0000 0000 0000 0000 fffe
0000 0000 0000 0000 0000 0000 0001 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
```

```
0036 fffa 0005 fffe 0000 ffff ffff 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
```

Cb

```
fff3 fffe 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
```

Cr

```
000e ffff 0001 0000 0000 0000 0001 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
```

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 `0x004d (= 77)` 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 77 est `1001101`

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 le flux de l'image compressée est le code de la magnitude puis l'indice dans la classe de magnitude, soit ici : `111101001101`

Bloc suivant La valeur DC du bloc suivant de la composante Y est `0x0036 (= 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 $54 - 77 = -23$. En supposant que le code de Huffman de la magnitude 5 soit `110`, l'encodage de -23 dans le flux de bits est `11001000`.

A.8 Codage AC avec RLE

On s'intéresse au codage des 63 coefficients AC du bloc de Cr. La composante continue `0x000e` a déjà été encodée dans le flux. Il reste donc la séquence : `0xffff 0x0001 0x0000 0x0000 0x0000 0x0001 0x0000 ... 0x0000`.

Pour le premier coefficient `0xffff` (= -1) :

— Un premier symbole sur un octet est calculé. Les quatre bits de poids forts sont nuls, car aucun coefficient nul ne précède ce coefficient. Les quatre bits de poids faible contiennent la magnitude de -1, qui est 1. Le symbole est donc ici `0x01` ;

— Dans la classe de magnitude 1, l'indice en binaire de -1 est `0`.

Le prochain coefficient non nul est `0x0001` (= 1) :

— Un premier symbole sur un octet est calculé. Les quatre bits de poids forts sont nuls, car aucun coefficient nul ne précède ce coefficient. Les quatre bits de poids faible contiennent la magnitude de 1, qui est 1. Le symbole est donc ici `0x01` ;

— Dans la classe de magnitude 1, l'indice en binaire de 1 est `1`.

Le prochain coefficient non nul est `0x0001` (= 1) :

— Un premier symbole sur un octet est calculé. Cette fois-ci, on a 3 coefficients nuls qui précèdent le coefficient considéré. Les quatre bits de poids forts prennent donc la valeur 3. Les quatre bits de poids faible contiennent la magnitude de 1, qui est 1. Le symbole est donc ici `0x31` ;

— Dans la classe de magnitude 1, l'indice en binaire de 1 est `1`.

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

Encodage dans le flux de bits En supposant que les codes de Huffman des symboles soient `01` pour `0x01`, `11011` pour `0x31` et `1010` pour `0x00`, l'ensemble des 63 coefficients AC du bloc est totalement encodé dans le flux par les 16 bits : `01 0 01 1 11011 1 1010`.

Annexe B

Le format JPEG

B.1 Principe du format JFIF/JPEG

Le format d'un fichier JFIF/JPEG est 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). **La norme impose que les marqueurs soient toujours alignés dans le flux sur un multiple d'octets.**

Chaque section d'un flux JPEG a un rôle 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 et identifiants

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

— les *identifiants* des composantes de couleur, qu'on notera i_C ;
— les *indices* de table de Huffman, qui sont en fait la concaténation de deux indices $(i_{AC/DC}, i_H)$;
— et les *indices* de table de quantification, qu'on notera i_Q.

L'identifiant d'une composante est un entier entre 0 et 255. Les indices des tables sont eux des « vrais » indices : 0, 1, 2, etc. 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 général d'une section JPEG est le suivant :

Offset	Taille (octets)	Description
`0x00`	2	Marqueur pour identifier la section
`0x02`	2	Longueur de la section en nombre d'octets, y compris les 2 octets codant cette longueur
`0x04`	?	Données associées (dépendent 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*) `0xffd8` et termine par un marqueur EOI (*End of Image*) `0xffd9`. 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 qu'il soit possible qu'un fichier contienne plusieurs images (format de vidéo MJPEG, pour *Motion JPEG*), nous nous limiterons dans ce projet au cas d'une seule image par fichier.

B.2.2 APPx - *Application data*

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. Les seules choses qui nous intéressent sont la séquence des 4 premiers octets de la section, qui doit contenir la phrase « JFIF », et le numéro de version JFIF X.Y codé sur deux octets (un pour X, un pour Y).

Offset	Taille (octets)	Description
`0x00`	2	Marqueur APP0 (`0xffe0`)
`0x02`	2	Longueur de la section
`0x04`	5	'J' 'F' 'I' 'F' '\0'
`0x09`	1	Version JFIF (**1**.1) : **doit valoir 1**
`0x0A`	1	Version JFIF (1.**1**) : **doit valoir 1**
`0x0B`	7	Données spécifiques au JFIF, non traitées : **tout mettre à 0**

B.2.3 COM - Commentaire

Afin de rajouter des informations textuelles supplémentaires, il est possible d'ajouter des sections de commentaires dans le fichier (par exemple, on trouve parfois le nom de l'encodeur). Notez que cela nuit à l'objectif de compression, les commentaires étant finalement des informations inutiles.

Offset	Taille (octets)	Description
`0x00`	2	Marqueur COM (`0xfffe`)
`0x02`	2	Longueur de la section
`0x04`	?	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 (souvent 2, au maximum 4). 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 soit une seule section DQT avec plusieurs tables, soit plusieurs sections DQT avec une table à chaque fois. C'est la longueur d'une section qui permet de déterminer combien de tables elle contient.

Offset	Taille (octets)	Description
`0x00`	2	Marqueur DQT (`0xffdb`)
`0x02`	2	Longueur 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 zig-zag** (cf. section 2.9.3)

B.2.5 SOFx - *Start Of Frame*

Le marqueur SOF définit le début effectif d'une image, et donne les informations générales rattachées à cette image. Il existe plusieurs marqueurs SOF selon le type d'encodage JPEG utilisé. Dans le cadre de ce projet, nous ne nous intéressons qu'au JPEG *Baseline DCT (Huffman)*, soit SOF0 (`0xffc0`). Pour information, les autres types sont récapitulés en section B.3.

Les informations générales associées à une image sont la précision des données (le nombre de bits codant chaque coefficient, toujours 8 dans notre cas), les dimensions de l'image, et le nombre de composantes de couleur utilisées (1 en niveaux de gris, 3 en YCbCr).

Pour chacune de ces composantes sont définis :

— un identifiant i_C entre 0 et 255, qui sera référencé dans la section SOS ;
— les facteurs d'échantillonnage horizontal et vertical. Comme décrit section 2.7.1, ces facteurs $h \times v$ indiquent le *nombre de blocs par MCU* codant la composante (dans le cas de Y, il s'agit donc de la taille de la MCU) ;
— l'indice i_Q de la table de quantification associée à la composante.

Dans cette section SOF (et ce n'est garanti qu'ici), l'ordre des composantes est toujours le même : d'abord Y, puis Cb puis Cr. Les identifiants sont normalement fixés à 1, 2 et 3. Cependant, certains encodeurs ne suivent pas cette obligation.

Finalement, une section SOF suit le format :

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, toujours 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 : Identifiant de composante i_C, de 0 à 255 - 4 bits : Facteur d'échantillonnage (*sampling factor*) horizontal, de 1 à 4 - 4 bits : Facteur d'échantillonnage (*sampling factor*) vertical, de 1 à 4 - 1 octet : Table de quantification i_Q associée, de 0 à 3

B.2.6 DHT - *Define Huffman Table*

La section DHT permet de définir une (ou plusieurs) table(s) de Huffman, selon le format décrit en section 2.10.1. Pour chaque table sont aussi définis ses indices de repérage $i_{AC/DC}$ et i_H.

Comme pour DQT, une section DHT peut contenir une ou plusieurs tables. Dans ce dernier cas, il y a en fait répétition des 3 dernières cases du tableau suivant. La longueur en octets de la section représente la taille nécessaire pour stocker toutes les tables contenues. Au décodage, pour déterminer si une section contient plusieurs tables, il faut donc regarder combien d'octets ont été lus pour construire une table. S'il en reste, la section contient encore (au moins) une autre table. A l'encodage, libre à vous, lorsque plusieurs tables de Huffman sont utilisées, de les définir dans autant de sections DHT, ou de les rassembler dans une seule section.

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

Offset	Taille (octets)	Description
`0x00`	2	Marqueur DHT (`0xffc4`)
`0x02`	2	Longueur 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 ou égale à 256
`0x15`	?	Table contenant les symboles, triés par longueur (*cf* 2.10.1)

B.2.7 SOS - *Start Of Scan*

La section SOS marque le début de l'encodage effectif du flux JPEG, c'est-à-dire le début des données brutes encodant l'image. 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 lues les composantes. D'après la norme, cet ordre devrait être le même que dans la section SOF, soit Y, Cb puis Cr.

Les trois derniers octets de l'entête de la section SOS (on parle de *Scan Header*) ne sont pas utilisés dans le cadre de ce projet, et doivent prendre les valeurs respectives 0, 63 et 0.

Les données brutes sont ensuite stockées par blocs 8×8 encodés RLE, dans l'ordre des composantes indiqué dans la section SOS. Le nombre de blocs à lire par composante de MCU dépend des facteurs d'échantillonnage lus en section SOF. Leur ordre est spécifié en 2.7.2.

Pour plus d'informations sur la signification de ces champs, rendez-vous en section B.2.3 (page 37) de la norme JPEG (document `itu-t81.pdf` distribué avec le sujet).

Offset	Taille (octets)	Description
`0x00`	2	Marqueur SOS (`0xffda`)
`0x02`	2	Longueur 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 : identifiant i_C de la composante 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 : **doit valoir 0**
...	1	Se : Fin de la sélection : **doit valoir 63**
...	1	Approximation successive Ah : 4 bits, poids fort : **doit valoir 0** Al : 4 bits, poids faible : **doit valoir 0**

B.3 Récapitulatif des marqueurs

Code (0xff??)	Identifiant	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

Spécification des modules fournis

Cinq modules sont fournis dans ce projet, sous forme d'une spécification (fichier `.h`). Pour trois d'entres eux, nous fournissons aussi un fichier objet binaire compilé (extension `.o`), les deux autres n'étant composé que d'un fichier entête. Le code source n'est pas distribué.

Dans un premier temps, vous devrez utiliser ces modules, pour vous concentrer sur la partie encodage de l'image. Ensuite, il vous sera demandé de les réécrire vous-même.

C.1 Ecriture des sections JPEG : module `jpeg_writer`

Ce module, d'entête `jpeg_writer.h`, fournit des fonctions pour accéder aux paramètres définis dans les sections d'un fichier JPEG décrites en annexe B. Ce module ne permet d'écrire que des fichiers JPEG encodés en mode {JFIF, baseline séquentiel, DCT, Huffman}. Seuls les paramètres principaux nécessaires à votre encodeur sont accessibles. Ceux inutilisés ou constants (par exemple *precision*, qui vaut toujours 8 dans notre cas) ne peuvent pas être accédés.

Enumérations Quelques types énumérés sont d'abord définis pour les composantes de couleur (Y, Cb, Cr), les directions (horizontal ou vertical) et les composantes fréquentielles (DC ou AC).

```
/* Type énuméré représentant les composantes de couleur YCbCr. */
enum color_component
{
    Y,
    Cb,
    Cr,
    NB_COLOR_COMPONENTS
};

/*
    Type énuméré représentant les types de composantes fréquentielles (DC ou
    AC).
*/
enum sample_type
{
    DC,
    AC,
    NB_SAMPLE_TYPES
};

/*
    Type énuméré représentant la direction des facteurs d'échantillonnage (H
    pour horizontal, V pour vertical).
*/
enum direction
{
    H,
    V,
    NB_DIRECTIONS
};
```

Descripteur Le module déclare une structure `struct jpeg_desc`, un descripteur contenant toutes les informations à écrire dans les entêtes des sections JPEG.

```
struct jpeg_desc;
```

C'est une structure C dite *opaque* qui n'est que *déclarée* dans le fichier `jpeg_writer.h`, mais dont on ne connaît 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 passée en paramètre aux différentes fonctions `jpeg_desc_get_XXX`, aussi définies dans `jpeg_writer.h`, pour accéder aux informations, ou appeler les fonctions `jpeg_desc_set_XXX` pour surcharger le comportement par défaut (indiquer les dimensions de l'image, le nombre de composantes de couleur, la table de huffman à utiliser, ...).

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

Ouverture, fermeture et fonctions générales

```
/* Alloue et retourne une nouvelle structure jpeg_desc. */
extern struct jpeg_desc *jpeg_desc_create(void);

/*
    Détruit un jpeg_desc. Toute la mémoire qui lui est
    associée est libérée.
*/
extern void jpeg_desc_destroy(struct jpeg_desc *jdesc);

/*
    Ecrit toute l'entête JPEG dans le fichier de sortie à partir des
    informations contenues dans le jpeg_desc passé en paramètre. En sortie, le
    bitstream est positionné juste après l'écriture de l'entête SOS, à
    l'emplacement du premier octet de données brutes à écrire.
*/
extern void jpeg_write_header(struct jpeg_desc *jdesc);

/* Ecrit le footer JPEG (marqueur EOI) dans le fichier de sortie. */
extern void jpeg_write_footer(struct jpeg_desc *jdesc);
```

- `jpeg_desc_create` crée une nouvelle structure `jpeg_desc`, pleine de vide ;
- `jpeg_write_header` écrit dans le fichier JPEG de sortie l'entête JPEG, à savoir tout ce qui se trouve *avant* les données brutes d'une image. Les informations écrites dans le fichiers sont lues depuis le contenu de la structure `jpeg_desc` passée en paramètre ;
- `jpeg_desc_write_footer` écrit le footer JPEG dans le fichie de sortie ;
- `jpeg_desc_destroy` détruit le `jpeg_desc` passé en paramètre en libérant la mémoire qui lui est associée.

Le schéma général d'utilisation est toujours :

1. on crée un `jpeg_desc` vide ;
2. on y insère les informations pertinentes qui doivent figurer dans les sections de l'entête JPEG, à l'aide des fonctions `jpeg_desc_set_XXX` du fichier `jpeg_writer.h` ;

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 fondamentaux de la programmation orientée objet, mais on peut s'en servir dans d'autres contextes. La preuve.

3. on écrit l'entête JPEG construite à partir des informations contenues dans le `jpeg_desc` à l'aide de la fonction *jpeg_write_header*;
4. à partir de là, si on a besoin d'informations définies dans l'entête JPEG, on peut appeler les fonctions *jpeg_desc_get_XXX* sur le `jpeg_desc` pour les récupérer;
5. on fait ce qu'il faut pour générer et écrire dans le fichier de sortie les données brutes de l'image;
6. quand c'est terminé, on n'oublie pas d'écrire le footer JPEG en appelant *jpeg_write_footer* sur le `jpeg_desc`;
7. enfin, on libère la mémoire associée au `jpeg_desc` en appelant *jpeg_desc_destroy*.

Par exemple :

```
/* Hop, un nouveau jpeg_desc tout propre! */
struct jpeg_desc *jdesc = jpeg_desc_create();

/* On met tout ce qu'il faut dedans... */
jpeg_desc_set_ppm_filename(jdesc, "chabanas.ppm");
jpeg_desc_set_jpeg_filename(jdesc, "bachanas.jpg");
jpeg_desc_set_image_width(jdesc, 640);
jpeg_desc_set_image_height(jdesc, 480);
...

/* C'est prêt! Plus qu'à écrire l'entête jpeg
   dans le fichier de sortie. */
jpeg_write_header(jdesc);
...
```

Les modifieurs et accesseurs des différents paramètres du `jpeg_desc` sont définis dans la suite de ce document. Le fait qu'ils soient présents ne signifie par forcément que vous en aurez besoin, tout dépend de l'ordre dans lequel vous attaquerez l'implémentation des modules "prof" !

Caractéristiques de l'image à traiter *set/get* relatifs au fichier PPM à encoder :

```
/* Ecrit le nom de fichier PPM ppm_filename dans le jpeg_desc jdesc. */
extern void jpeg_desc_set_ppm_filename(struct jpeg_desc *jdesc,
                                       const char *ppm_filename);

/* Retourne le nom de fichier PPM lu dans le jpeg_desc jdesc. */
extern char *jpeg_desc_get_ppm_filename(struct jpeg_desc *jdesc);
```

```
/*
    Ecrit la hauteur en nombre de pixels de l'image traitée image_height dans
    le jpeg_desc jdesc.
*/
extern void jpeg_desc_set_image_height(struct jpeg_desc *jdesc,
                                       uint32_t image_height);

/*
    Retourne la hauteur en nombre de pixels de l'image traitée lue dans le
    jpeg_desc jdesc.
*/
extern uint32_t jpeg_desc_get_image_height(struct jpeg_desc *jdesc);

/*
    Ecrit la largeur en nombre de pixels de l'image traitée image_width dans le
    jpeg_desc jdesc.
*/
extern void jpeg_desc_set_image_width(struct jpeg_desc *jdesc,
                                      uint32_t image_width);

/*
    Retourne la largeur en nombre de pixels de l'image traitée lue dans le
    jpeg_desc jdesc.
*/
extern uint32_t jpeg_desc_get_image_width(struct jpeg_desc *jdesc);
```

```
/*
    Ecrit le nombre de composantes de couleur de l'image traitée nb_components
    dans le jpeg_desc jdesc.
*/
extern void jpeg_desc_set_nb_components(struct jpeg_desc *jdesc,
                                        uint8_t nb_components);

/*
    Retourne le nombre de composantes de couleur de l'image traitée lu à partir
    du jpeg_desc jdesc.
*/
extern uint8_t jpeg_desc_get_nb_components(struct jpeg_desc *jdesc);
```

Paramètres de l'encodage *set/get* relatifs au fichier JPEG de sortie et la façon de le générer :

```
/* Ecrit le nom du fichier de sortie jpeg_filename dans le jpeg_desc jdesc. */
extern void jpeg_desc_set_jpeg_filename(struct jpeg_desc *jdesc,
                                        const char *jpeg_filename);

/* Retourne le nom du fichier de sortie lu depuis le jpeg_desc jdesc. */
extern char *jpeg_desc_get_jpeg_filename(struct jpeg_desc *jdesc);
```

```
/*
    Ecrit dans le jpeg_desc jdesc le facteur d'échantillonnage sampling_factor
    à utiliser pour la composante de couleur cc et la direction dir.
*/
extern void jpeg_desc_set_sampling_factor(struct jpeg_desc *jdesc,
                                          enum color_component cc,
                                          enum direction dir,
                                          uint8_t sampling_factor);

/*
    Retourne le facteur d'échantillonnage utilisé pour la composante de couleur
    cc et la direction dir, lu à partir du jpeg_desc jdesc.
*/
extern uint8_t jpeg_desc_get_sampling_factor(struct jpeg_desc *jdesc,
                                             enum color_component cc,
                                             enum direction dir);
```

Tables de Huffman *set/get* relatifs aux tables de Huffman à utiliser :

```
/*
    Ecrit dans le jpeg_desc jdesc la table de Huffman huff_table à utiliser
    pour encoder les données de la composante fréquentielle acdc pour la
    composante de couleur cc.
*/
extern void jpeg_desc_set_huffman_table(struct jpeg_desc *jdesc,
                                        enum sample_type acdc,
                                        enum color_component cc,
                                        struct huff_table *htable);

/*
    Retourne un pointeur vers la table de Huffman utilisée pour encoder les
    données de la composante fréquentielle acdc pour la composante de couleur
    cc, lue à partir du jpeg_desc jdesc.
*/
extern struct huff_table *jpeg_desc_get_huffman_table(struct jpeg_desc *jdesc,
                                                      enum sample_type acdc,
                                                      enum color_component cc);

/*
```

Tables de quantification *set/get* relatifs aux tables de quantification à utiliser :

```
/*
    Ecrit dans le jpeg_desc jdesc la table de quantification qtable à utiliser
    pour compresser les coefficients de la composante de couleur cc.
*/
extern void jpeg_desc_set_quantization_table(struct jpeg_desc *jdesc,
                                             enum color_component cc,
                                             uint8_t *qtable);

/*
    Retourne la table de quantification associée à la composante de couleur
    cc dans le jpeg_desc jdesc.
*/
extern uint8_t *jpeg_desc_get_quantization_table(struct jpeg_desc *jdesc,
                                                 enum color_component cc);
```

Bitstream La fonction suivante permet de récupérer le bitstream associé au `jpeg_desc`. Ce bitstream est créé lors de l'appel à la fonction *jpeg_write_header*, et est détruit par un appel à la fonction *jpeg_desc_destroy*.

```
/*
    Retourne le bitstream associé au fichier de sortie enregistré dans le
    jpeg_desc.
*/
extern struct bitstream *jpeg_desc_get_bitstream(struct jpeg_desc *jdesc);
```

C.2 Tables de quantification : module `qtables`

Vous utiliserez dans un premier temps les tables de quantification définies "en dur" dans le fichier d'entête `qtables.h`. L'utilisation d'autres tables de quantification pourra être étudiée en extension du projet.

```
/* Tables de "The GIMP" (cf sujet). */
static uint8_t compressed_Y_table[] = {
    0x05, 0x03, 0x03, 0x05, 0x07, 0x0c, 0x0f, 0x12,
    0x04, 0x04, 0x04, 0x06, 0x08, 0x11, 0x12, 0x11,
    0x04, 0x04, 0x05, 0x07, 0x0c, 0x11, 0x15, 0x11,
    0x04, 0x05, 0x07, 0x09, 0x0f, 0x1a, 0x18, 0x13,
    0x05, 0x07, 0x0b, 0x11, 0x14, 0x21, 0x1f, 0x17,
    0x07, 0x0b, 0x11, 0x13, 0x18, 0x1f, 0x22, 0x1c,
    0x0f, 0x13, 0x17, 0x1a, 0x1f, 0x24, 0x24, 0x1e,
    0x16, 0x1c, 0x1d, 0x1d, 0x22, 0x1e, 0x1f, 0x1e
};

static uint8_t compressed_CbCr_table[] = {
    0x05, 0x05, 0x07, 0x0e, 0x1e, 0x1e, 0x1e, 0x1e,
    0x05, 0x06, 0x08, 0x14, 0x1e, 0x1e, 0x1e, 0x1e,
    0x07, 0x08, 0x11, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e,
    0x0e, 0x14, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e,
    0x1e, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e,
    0x1e, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e,
    0x1e, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e,
    0x1e, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e, 0x1e
};
```

C.3 Ecriture bit à bit dans le flux : module `bitstream`

Ce module permet d'écrire des bits dans le fichier de sortie (et non des octets comme dans une écriture standard), pour écrire les informations portées par l'entête du fichier JPEG 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.

```
/*
    Type opaque représentant le flux d'octets à écrire dans le fichier JPEG de
    sortie (appelé bitstream dans le sujet).
*/
struct bitstream;

/* Retourne un nouveau bitstream prêt à écrire dans le fichier filename. */
extern struct bitstream *bitstream_create(const char *filename);

/*
    Ecrit nbits bits dans le bitstream. La valeur portée par cet ensemble de
    bits est value. Le paramètre is_marker permet d'indiquer qu'on est en train
    d'écrire un marqueur de section dans l'entête JPEG ou non (voir section
    2.10.4 du sujet).
*/
extern void bitstream_write_nbits(struct bitstream *stream,
                                  uint32_t value,
                                  uint8_t nbits,
                                  bool is_marker);

/*
    Force l'exécution des écritures en attente sur le bitstream, s'il en
    existe.
*/
extern void bitstream_flush(struct bitstream *stream);

/*
    Détruit le bitstream passé en paramètre, en libérant la mémoire qui lui est
    associée.
*/
extern void bitstream_destroy(struct bitstream *stream);
```

Type de données Un flux de bits est représenté par le type de données `struct bitstream`. Là encore c'est une structure C *opaque* qui n'est que déclarée dans `bitstream.h`. Quand vous implémenterez ce module, il vous reviendra de *définir* la structure dans votre fichier `bitstream.c`, avec les champs que vous aurez choisis pour réaliser les tâches demandées.

Création et fermeture La fonction `bitstream_create` crée un flux associé à un fichier initialement vide de nom `filename`. Le prochain bit à écrire est le donc le premier du fichier. Cette fonction alloue dynamiquement une variable de type `struct bitstream`, la remplit de manière adéquate, puis retourne son adresse mémoire. En cas d'erreur, la fonction retourne `NULL`.

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

Le schéma d'utilisation est :

```
struct bitstream *stream = bitstream_create("chabanas.jpg");

...

// écrit la valeur 42 stockée sur 5 bits dans le flux
bitstream_write_nbits(stream, 5, 42, false);

...

bitstream_destroy(stream);
```

Dans le premier temps où vous utiliserez tous nos modules, notez que la création et la fermeture du `bitstream` est faite dans les fonctions *jpeg_write_header* et *jpeg_desc_destroy* du module `jpeg_writer`.

Ecriture La fonction *bitstream_write_nbits* permet d'écrire des bits et d'avancer dans le flux.
Ses paramètres sont :

— le flux de bits *stream* ;
— la valeur à écrire dans le flux *value* ;
— le nombre de bits *nbits* à écrire dans le flux, dont l'ensemble porte la valeur *value*. Ce nombre ne peut pas excéder 32.
— un booléen *is_marker* qui indique si on est en train d'écrire un marqueur de section JPEG ou non (voir section 2.10.4 pour une explication détaillée de pourquoi c'est important !).

De manière similaire aux fonctions de la libc manipulant des entrées/sorties, il est courant d'implémenter l'écriture bit-à-bit dans un fichier à l'aide d'un buffer. Ce buffer permet de stocker les bits en attente d'écriture dans le fichier, leur écriture effective étant provoquée quand le buffer est plein.

La fonction *bitstream_flush* permet de purger le contenu du buffer interne au module bitstream, forçant ainsi l'écriture dans le fichier de sortie des bits en attente de traitement. Cette fonction est utile dans le projet **pour garantir qu'un marqueur de section JPEG sera toujours aligné dans le flux sur un multiple d'un octet, comme le spécifie la norme JPEG.**

C.4 Gestion des tables de Huffman : modules huffman et htables

La gestion des tables de Huffman est découpée en deux modules. Le premier, huffman, permet de construire et d'utiliser n'importe quelle table de Huffman décrite sous la forme proposée en section 2.10.1. Le deuxième, htables, n'est composé que d'un fichier entête qui définit les informations nécessaires à la construction des tables de Huffman génériques décrites dans la norme JPEG.

C.4.1 huffman

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

```
/* Type opaque représentant un arbre de Huffman. */
struct huff_table;

/*
    Construit un arbre de Huffman à partir d'une table de symboles comme
    présenté en section 2.10.1 du sujet. nb_symb_per_lengths est un tableau
    contenant le nombre de symboles pour chaque longueur, symbols est le
    tableau des symboles ordonnés, et nb_symbols représente la taille du
    tableau symbols.
*/
extern struct huff_table *huffman_table_build(uint8_t *nb_symb_per_lengths,
                                              uint8_t *symbols,
                                              uint8_t nb_symbols);

/*
    Retourne le chemin dans l'arbre ht permettant d'atteindre la feuille de
    valeur value. nbits est un paramètre de sortie permettant de stocker la
    longueur du chemin.
*/
extern uint32_t huffman_table_get_path(struct huff_table *ht,
                                       uint8_t value,
                                       uint8_t *nbits);
/*
    Retourne le tableau des symboles associé à l'arbre de Huffman passé en
    paramètre.
*/
extern uint8_t *huffman_table_get_symbols(struct huff_table *ht);

/*
    Retourne le tableau du nombre de symboles de chaque longueur associé à
    l'arbre de Huffman passé en paramètre.
*/
extern uint8_t *huffman_table_get_length_vector(struct huff_table *ht);

/*
    Détruit l'arbre de Huffman passé en paramètre et libère toute la mémoire
    qui lui est associée.
*/
extern void huffman_table_destroy(struct huff_table *ht);
```

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

Création La fonction `huffman_table_build` construit un arbre de Huffman à partir de sa représentation simplifiée, comme présentée en 2.10.1. Pour ce faire, on passe à cette fonction :

- la table des nombres de symboles `nb_symb_per_lengths` : cette table, indicée par la longueur d'un symbole, contient le nombre de symboles pour chaque longueur possible (comprise entre 0 et 15) ;
- la table des symboles `symbols`, ordonnée comme présenté en 2.10.1 ;
- le nombre de symboles dans la table `nb_symbols` : autrement dit, la longueur du tableau `symbols`.

L'arbre de Huffman construit est retourné par la fonction sous la forme d'un `struct huff_table *`. En plus de construire l'arbre, cette fonction enregistre le tableau des longueurs et le tableau des symboles qui lui sont associés.

Calcul d'un chemin dans l'arbre Dans un arbre de Huffman, les symboles encodés sont portés par les feuilles, et leur code correspond au chemin parcouru depuis la racine pour attendre cette feuille. La fonction `huffman_table_get_path` permet de récupérer ce chemin sous la forme d'une valeur codée sur au plus 32 bits. Le nombre de bits nécessaires pour représenter ce chemin est passé dans le paramètre de sortie `nbits`. Autrement dit, `nbits` contient l'adresse mémoire d'une variable de type `uint8_t` dont la valeur sera, à la sortie de la fonction, le nombre de bits nécessaires pour représenter le chemin parcouru entre la racine de l'arbre et la feuille qui porte la valeur `value`.

Accès aux tableaux

Les fonctions `huffman_table_get_symbols` et `huffman_table_get_length_vector` permettent d'accéder au tableau des symboles et au tableau des nombres de symboles par longueur associés à la table de Huffman `ht`. Ces tableaux ont été associés à la table lors de sa création par la fonction `huffman_table_build`.

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

C.4.2 `htables`

Ce module consigne les informations nécessaires à la construction des arbres de Huffman décrits dans la norme JPEG sous forme de tables indexées par le type d'échantillon (DC, AC) et la composante de couleur (Y, Cb, Cr). On retrouve ainsi dans le fichier `htables.h` les tables de symboles et leur taille, ainsi que les tables des nombres de symboles par longueur. Un extrait de ce fichier est donné ci-après :

```
/*
    Table des nombres de symboles par longueur, indexée par le type
    d'échantillon (DC ou AC), la composante de couleur (Y, Cb ou Cr) et la
    longueur du symbole (comprise entre 0 et 15).
*/
extern uint8_t
   htables_nb_symb_per_lengths[NB_SAMPLE_TYPES][NB_COLOR_COMPONENTS][16];

/*
    Table des symboles, indexée par le type d'échantillon (DC ou AC) et la
    composante de couleur (Y, Cb ou Cr).
*/
extern uint8_t *htables_symbols[NB_SAMPLE_TYPES][NB_COLOR_COMPONENTS];

/*
    Table des longueurs des tableaux de symboles, indexée par le type
    d'échantillon (DC ou AC) et la composante de couleur (Y, Cb ou Cr).
*/
extern uint8_t htables_nb_symbols[NB_SAMPLE_TYPES][NB_COLOR_COMPONENTS];

#endif /* _HTABLES_H_ */
```