Ensimag — Printemps 2017

—

Projet Logiciel en C

—

Sujet : Décodeur 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 décodeur d'images compressées au format JPEG**, pour les écrire dans un format brut PPM. Il est nécessaire d'avoir une bonne compréhension du codec, qui reprend notamment des notions vues dans les enseignements de théorie de l'information et d'algorithmique et structures de données. Mais l'essentiel de votre travail sera bien évidemment de **concevoir** et d'**implémenter** votre décodeur 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.

Le décodeur 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 « décodeur `ISO/IEC IS 10918-1 | ITU-T Recommendation T.81` ». Qu'on se le dise !

2.1 Principe général du codec JPEG

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

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/décompression d'un bloc 8×8. Tout d'abord, chaque bloc est traduit dans le domaine fréquentiel par transformation en cosinus discrète (DCT). Le résultat de ce traitement, appelé bloc fréquentiel, est encore un bloc 8×8 mais dont les coordonnées sont des fréquences et non plus des pixels. On y distingue une composante continue *DC* aux coordonnées $(0, 0)$ et 63 composantes fréquentielles *AC*. [2] Les plus hautes fréquences se situent autour de la case $(7, 7)$.

L'œil étant 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.

Le décodeur qui est le but de ce projet effectue les opérations dans l'ordre inverse. Les opérations de codage/décoge sont résumées figure 2.1, puis détaillées dans les sections suivantes (dans le sens du décodage). L'annexe A fournit elle un exemple numérique du codage d'une MCU.

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, le décodeur utilise deux types de représentation de l'image.

Le format RGB, le plus courant, est le format utilisé en sortie. 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 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 du décodeur.

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.

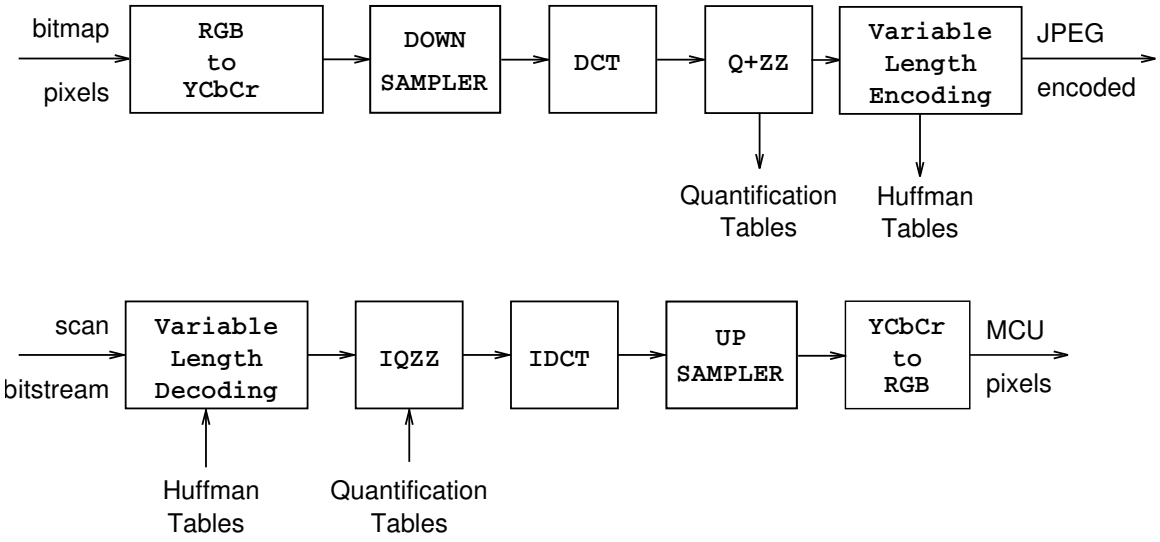


FIGURE 2.1 – Principe du codec JPEG : opérations de codage (en haut) et de décodage (en bas).

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 est moins sensible.

2.3 Lecture et analyse du flux

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

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

Deux grands types de sections peuvent être distingués :

— définition de l'environnement : ces sections contiennent des données permettant d'initialiser le décodage du flux. La plupart des informations du JPEG étant dépendantes de l'image, c'est une étape nécessaire. Les informations à récupérer concernent, par exemple, la taille de l'image, ou les tables de Huffman utilisées. Elles peuvent nécessiter un traitement particulier avant d'être utilisables ;

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

Une liste exhaustive des marqueurs est définie dans la norme JFIF. Les principaux vous sont

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.

donnés en annexe B de ce document, avec la représentation des données utilisées et la liste des actions à effectuer. On notera cependant ici 4 marqueurs importants :

SOI : le marqueur *Start Of Image* 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 un SOF0. La section SOF contient notamment la taille de l'image et les facteurs de sous-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 Décompression d'un bloc fréquentiel

Les blocs fréquentiels sont compressés sans perte dans le *bitstream* JPEG par l'utilisation de plusieurs techniques successives. Tout d'abord, les répétitions de 0 sont exploitées par un codage de type *RLE* (voir 2.4.3), puis les valeurs non nulles sont codées comme *différence* par rapport aux valeurs précédentes (voir 2.4.2), enfin, les symboles obtenus par l'application des deux codages précédents sont codés par un codage entropique de Huffman (2.4.1). Nous présentons dans cette sections les trois codages dans l'ordre qui nous intéresse pour ce projet : celui de la décompression.

2.4.1 Le codage de Huffman

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

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

Ces codes sont dits *préfixés* car par construction aucun code de symbole, considéré ici comme une suite de bits, n'est le préfixe d'un autre symbole. Autrement dit, si on trouve une certaine séquence de bits dans un message et que cette séquence correspond à un symbole qui lui est associé, cette séquence correspond forcément à ce symbole et ne peut pas être le début d'un autre code.

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

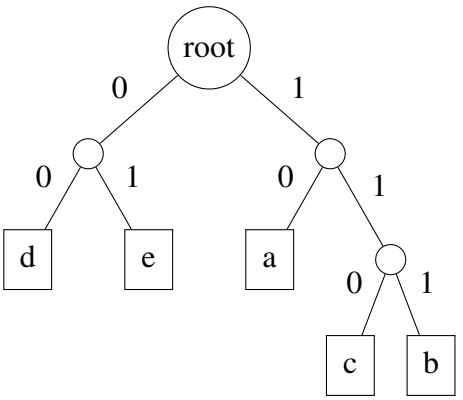
5. Pour une fréquence d'apparition des symboles connue et un codage de chaque symbole indépendamment des autres, ces codes sont optimaux.

La construction des codes de Huffman n'entre pas dans le cadre initial de ce projet. 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 ont 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.

Dans le cas du JPEG, les tables de codage [6] sont fournies avec l'image. On notera que la norme requiert l'utilisation de plusieurs arbres pour compresser plus efficacement les différentes composantes de l'image. Ainsi, en mode baseline, le décodeur supporte quatre tables :

— 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). 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

6. Chacune représentant un arbre de Huffman.

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.

2.4.2 Composante continue : DPCM, magnitude et arbre DC

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

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

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.

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

2.4.3 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 2.5) 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 2.4.2). Il est à noter que les 4 bits de la partie basse peuvent prendre des valeurs entre 1 et 10 puisque le zéro n'a pas besoin d'être codé et que la norme prévoie des valeurs entre -1023 et 1023 uniquement.

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

— code ZRL : `0xF0` désigne un saut de 16 composantes nulles (et ne code pas de composante non nulle)
— code EOB : `0x00` (*End Of Block*) signale que toutes les composantes AC restantes du bloc sont nulles.

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

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

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

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

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

2.4.4 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 ne pas interpréter ce `0x00` pour reconstruire les blocs, il faut jeter cet octet nul !

2.5 Quantification inverse et Zig-zag inverse

2.5.1 Quantification inverse

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

Deux tables de quantification sont généralement utilisées, une pour la luminance et une pour les deux chrominances. Le choix de ces tables, complexe mais fondamental quant à la qualité de la compression et au taux de perte d'information, n'est pas discuté ici. Les tables utilisées à l'encodage sont inclues dans le fichier JFIF. Avec une précision de 8 bits (le cas du mode *baseline*), les coefficients sont des entiers non signés entre 0 et 255.

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).

Au décodage, la quantification inverse consiste à multiplier élément par élément le bloc fréquentiel par la table de quantification. Moyennant la perte d'information, les blocs fréquentiels initiaux seront ainsi reconstruits.

2.5.2 Zig-zag inverse

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 2.2 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, ceci permet d'optimiser la compression RLE décrite précédemment.

Au décodage, le vecteur obtenu après décompression doit être réorganisé par l'opération zig-zag inverse qui recopie les 64 coefficients en entrée aux coordonnées fournie par le zig-zag de la figure 2.2. Ceci permet d'obtenir à nouveau une matrice 8×8.

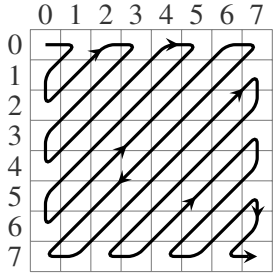


FIGURE 2.2 – Réordonnancement Zig-zag.

2.5.3 Ordre des opérations

Au codage, la quantification a lieu avant la réorganisation zig-zag. Par contre la matrice de quantification stockée dans le fichier JPEG est elle aussi réorganisée au format zig-zag ! Au décodage, l'ordre des opérations est donc inversé : il faut d'abord multiplier les coefficients du bloc par ceux de la matrice de quantification lue, puis effectuer la réorganisation zig-zag inverse.

2.6 Transformée en cosinus discrète inverse (iDCT)

Lors du codage, on applique une tranformée en cosinus discrète (DCT) qui convertit les informations spatiales en informations fréquentielles. Au décodage, l'étape inverte consiste à retransformer les informations fréquentielles en informations spatiales. C'est une formule mathématique « classique » de transformée. Dans sa généralité, la formule de la transformée en cosinus discrète inverse (iDCT, pour *Inverse Discrete Cosinus Transform*) pour les blocs de taille $n \times n$ pixels est :

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

Dans cette formule, S est le bloc spatial et Φ le bloc fréquentiel. Les variables x et y sont les coordonnées des pixels dans le domaine spatial et les variables λ et μ sont les coordonnées des fréquences dans le domaine fréquentiel. Finalement, le coefficient C est tel que

$$C(\xi) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } \xi = 0, \\ 1 & \text{sinon.} \end{cases}$$

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

A l'issue de cette opération :

— un offset de 128 est ajouté à chaque $S(x, y)$; [7]
— chaque valeur est « clampée », c'est-à-dire fixée à 0 si elle est négative, et à 255 si elle est supérieure à 255 ;
— finalement les valeurs sont converties en entier non signé sur 8 bits.

Attention, le calcul ci-dessus se fait bien en flottants. Ce n'est qu'à la fin qu'on effectue une conversion des valeurs en entier 8 bits non signés.

2.7 Reconstitution 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

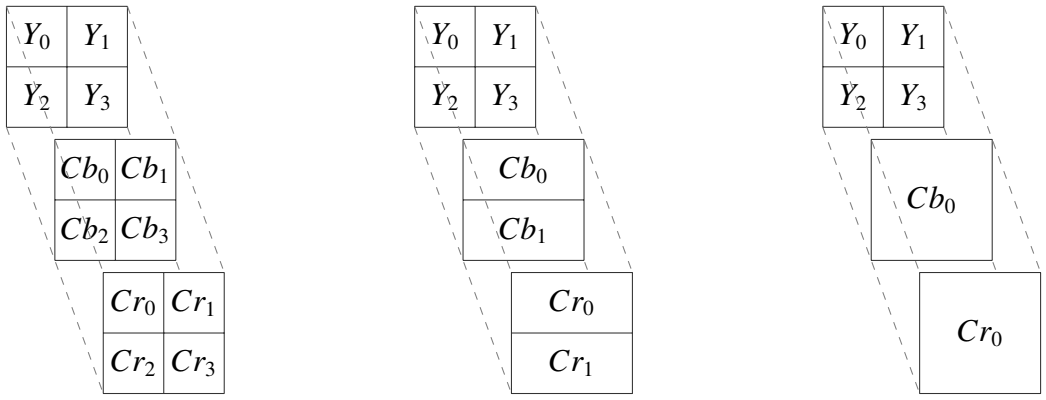
Dans ce document, nous utiliserons une notation directement en lien avec les valeurs présentes dans les sections JPEG de l'entête, décrites en annexe B, qui déterminent le facteur d'échantillonnage (*sampling factors*). Ces valeurs sont identiques partout dans une image. En pratique, on utilisera la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

7. A l'encodage, ce décalage des valeurs de $[0, 255]$ vers $[-128, 127]$ permet d'utiliser au mieux le codage par magnitude, avec des valeurs positives et négatives et de plus faible amplitude.



(a) Sans sous-échantillonnage

(b) Sous-échantillonnage horizontal Cb et Cr

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

FIGURE 2.3 – Composantes Y, Cb et Cr avec et sans sous-échantillonnage, pour une MCU de 2×2 blocs.

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. [8] 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 et Cr (figure 2.3(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 2.3(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. [9] 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.

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

2.7.2 Ordre de lecture 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.

Lorsqu'une composante de MCU comprend plusieurs blocs, 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 $Y_0^0 Y_1^0 Cr^0 Cb^0 Y_0^1 Y_1^1 Cr^1 Cb^1$.

Au niveau du décodeur, il faut à l'inverse sur-échantillonner (on parle d'*upsampling*) les blocs Cb et Cr pour qu'ils recouvrent la MCU en totalité. Ceci signifie par exemple que dans le cas d'un sous-échantillonnage horizontal, la moitié gauche de chaque bloc de chrominance couvre le premier bloc Y_0, alors que la moitié droite couvre le second bloc Y_1.

2.8 Conversion vers des pixels RGB

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

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

Les formules simplifiées suivantes sont encore acceptables pour respecter les contraintes de rapport signal sur bruit du standard. [10]

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

Ces calculs incluent des opérations arithmétiques sur des valeurs flottantes et signées, et dont les résultats sont potentiellement hors de l'intervalle $[0 \ldots 255]$. En sortie, les valeurs de R, G et B devront être entières et clampées entre 0 et 255 (comme suite à la DCT inverse).

2.9 Des MCUs à l'image

La dernière étape est de reconstituer l'image à partir de la suite des MCUs décodées.

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

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 à gauche (figure 2.4). Au codage, 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. Au décodage, les MCUs sont reconstruites normalement, il suffit de tronquer celles à droite et en bas de l'image selon le nombre exact de pixels.

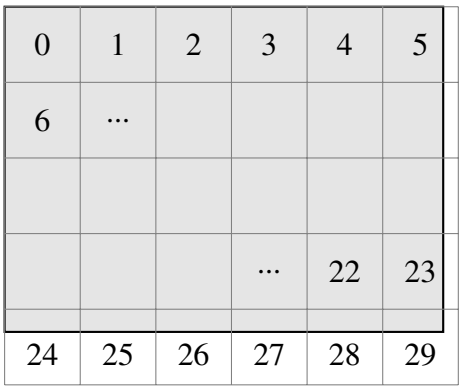


FIGURE 2.4 – Exemple d'une image 46 × 35 (fond gris) à compresser sans sous-échantillonnage, soit avec des MCUs composées d'un seul bloc 8 × 8. Pour couvrir l'image en entier, 6 × 5 MCUs sont nécessaires. Les MCUs les plus à droite et en bas (les 6^{e}, 12^{e}, 18^{e}, puis 24^{e} à 30^{e}) devront être tronquées lors de la décompression.

2.10 Format de sortie PPM

Une fois décodées, les images seront enregistrées au format 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.

La principe est d'écrire 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 valeur R, G et B du premier pixels, 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 de valeur entre 0 (noir) et 255 (blanc). Un exemple est donné figure 2.5.

```
$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 2.5 – En haut, trace de la commance `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.

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 JPEG en une image au format brut PPM :

— le nom de l'exécutable doit être `jpeg2ppm`
— il prend en unique paramètre le nom de l'image JPEG à convertir, d'extension `.jpeg` ou `.jpg`. Seules les images encodées en mode {JFIF, baseline séquentiel, DCT, Huffman} sont acceptées (application de type `APP0`).
— en sortie une image au format PPM sera générée, de même nom que l'image d'entrée et d'extension `.ppm` si elle est en couleur ou `.pgm` si elle est en niveaux de gris.

Par exemple :

```
./jpeg2ppm shaun_the_sheep.jpeg
```

génèrera le fichier `shaun_the_sheep.ppm`. C'est tout. Aucune trace particulière n'est attendue, hormis peut-être en cas d'erreur. Pour pouvez si le souhaitez ajouter des options, par exemple `-v` (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 du décodeur sont les suivantes :

1. Extraction de l'entête JPEG, récupération de la taille de l'image, des facteurs d'échantillonnage et des tables de quantification et de Huffman ;
2. Décodage de chaque MCU :
   (a) Reconstruction de chaque bloc :
      i. Extraction et décompression ;
      ii. Quantification inverse (multiplication par les tables de quantification) ;
      iii. Réorganisation zig-zag ;
      iv. Calcul de la transformée en cosinus discrète inverse (iDCT) ;
   (b) Mise à l'échelle des composantes Cb et Cr (*upsampling*), en cas de sous-échantillonnage ;
   (c) Reconstruction des pixels : conversion YCbCr vers RGB ;
3. Écriture du résultat dans le fichier PPM.

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

Etape	Difficulté pressentie	Module
Lecture bit à bit dans le flux	☆☆☆☆	`bitstream`
Lecture entête JPEG	☆☆☆☆	`jpeg_reader`
Gestion des tables de Huffman	☆☆☆☆☆	`huffman`
Extraction des blocs	☆☆☆	
Quantification inverse	☆	
Zig-zag	☆☆	
iDCT	☆☆	
Upsampling	☆☆☆☆	
conversion YCbCr vers RGB	☆	
Ecriture fichier PPM	☆☆ à ☆☆☆☆	
Gestion complète du décodage	☆☆ 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`). Le code source n'est pas distribué.

— le module `jpeg_reader` 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 1.

— le module `huffman` permet de représenter, lire, utiliser et détruire les tables de Huffman du fichier JPEG. Il sera utilisé à l'étape 2(a)i pour décoder la magnitude des coefficients DC et les symboles RLE des coefficients AC, à partir des tables de Huffman correspondantes ;

— le module `bitstream` permet de lire des bits dans le flux d'entrée, et non des octets comme avec une lecture standard. Il sera aussi utilisé à l'étape 2(a)i pour extraire des séquences de bits, représentant les indices des coefficients DC et AC dans leur classe de magnitude. Ce module est également utilisé par `jpeg_reader` pour lire dans l'entête JPEG.

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

3.2.3 Décodage d'abord, réécriture des modules ensuite

À terme, vous devrez bien sûr implanter **toutes** les étapes du décodeur. 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 du décodage vous sont fournis.

La démarche *fortement* conseillée est donc la suivante :

1. Commencez par travailler sur la **partie décodage** proprement dite, reconstruction des MCUs puis de l'image.
   Vous aurez simplement besoin d'**utiliser les modules fournis**, ce qui nécessite d'avoir bien compris le principe du décodage mais reste beaucoup plus simple que de les écrire.
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 décoder

Le second point porte sur les images à décoder. Bien évidemment, votre décodeur devra *in fine* être capable de traiter toutes les images JPEG 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 décodeur qui fonctionne !

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

— L'objectif est d'obtenir le plus rapidement possible un décodeur **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 outre d'être efficace et rationnel, 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 décodeur simple vers celui capable de traiter des images quelconques.

Bien entendu, vous pouvez utiliser vos propres images pour réaliser des tests supplémentaires !

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 vrai vie d'ingénieur, si si ! Pensez à nous remercier le moment venu.

Image	Caractéristiques	Simplifications / progression
`invader`	Niveaux de gris (1 composante) Taille 8×8 (un seul bloc)	Décodage 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 $1 \times 1, 1 \times 1, 1 \times 1$	Couleur, pas de troncature, pas d'échantillonnage
`thumbs`	439×324 $1 \times 1, 1 \times 1, 1 \times 1$	Couleur, tronquée droite et bas, pas d'échantillonnage
`horizontal`	367×367 $2 \times 1, 1 \times 1, 1 \times 1$	Échantillonnage horizontal
`vertical`	704×1246 $1 \times 2, 1 \times 1, 1 \times 1$	Échantillonnage vertical
`Shaun the sheep`	300×225 $2 \times 2, 1 \times 1, 1 \times 1$	Échantillonnage horizontal ET vertical
`complexite`	Niveaux de gris, 2995×2319	C'est looonnnng...

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

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

Dans 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 iDCT. 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 du décodage ?

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 !

— ...

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 le décodeur. 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_reader` ou certaines étapes « simples » du décodage (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 de l'outil `jpeg2blabla` distribué (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 décodeur.

`jpeg2blabla`

Ce programme est en fait une version « bavarde » de `jpeg2ppm`, réalisée par nos soins, qui fournit deux types informations :

Paramètres de l'image l'option `-v` (pour *verbose*) affiche dans la console les caractéristiques de l'image JPEG à décoder (celles utiles à ce projet). Un exemple est donné figure 3.1.

Traces de toutes les étapes du décodage en plus de décoder l'image, cet utilitaire crée également un fichier d'extension `.blabla` qui fournit les valeurs numériques de tous les blocs de chaque MCU, étape après étape (après décodage, après quantication inverse, etc.). Un exemple est donné figure 3.2. Ceci pourra s'avérer très utile (sur des images de taille réduite) pour valider votre décodeur 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 décodées sont bien les mêmes.

`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 2.5). Dans ce projet, c'est très utile pour regarder les données d'un fichier JPEG, comprendre sa structure et vérifier que les données lues sont correctes (notamment lorsque vous réécrirez le module `jpeg_reader`). 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 lus 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 JFIF baseline (supporté par votre décodeur) en jouant sur différents paramètres, de faire des modifications dans les fichiers, ou encore d'afficher des images JPEG ou PPM. Pour l'affichage seul, préférez **eog** (Eye Of Gnome), moins gourmand en ressources et plus rapide à invoquer.

```
[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 3.1 – 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.

```
***************************************************************
*** mcu 186
** component Y
* bloc 0
[  bloc] ffb5 0 2 0 0 1 0 0 ffff 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[iquant] fd12 0 e 0 0 6 0 0 fff8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[   izz] fd12 0 6 0 0 0 0 0 e 0 0 0 0 0 0 0 0 fff8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[  idct] 24 24 23 23 24 25 27 27 25 24 23 23 23 24 25 26 25 25 24 23 22 23 23 24 26 25 23 22 21 21
    22 22 25 24 22 21 20 20 21 21 23 22 21 20 20 20 21 21 20 20 1f 1f 1f 20 21 22 1f 1f 1e 1e 1f 20
    22 23
* bloc 1
[  bloc] ffdf ffd3 ffec ffff 3 2 ffff a 8 fffe 0 0 fffe 0 0 0 fffe fffe ffff 0 0 0 0 1 0 0 0 0 0 0
    1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[iquant] feb6 fec5 ff74 fff7 15 c fff6 5a 40 ffee 0 0 ffec 0 0 0 ffe0 ffe2 fff2 0 0 0 0 17 0 0 0 0
    0 0 23 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[   izz] feb6 fec5 c fff6 0 0 0 0 ff74 15 5a 0 ffe0 0 0 0 fff7 40 ffec ffe2 0 23 0 0 ffee 0 fff2 0
    20 0 0 0 0 0 17 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[  idct] 27 1f 2a 2b 22 3d 67 72 1c 26 2e 2b 31 56 76 79 18 2e 2d 26 44 72 89 8d 1c 2d 26 2a 58 80
    93 a5 1d 24 27 44 73 84 8d aa 1c 20 3a 69 89 85 86 9c 24 2c 54 80 8b 86 8b 91 31 3c 67 84 7f 86
    95 91
* component mcu
[   mcu] 24 24 23 23 24 25 27 27 27 1f 2a 2b 22 3d 67 72 25 24 23 23 23 24 25 26 1c 26 2e 2b 31 56
    76 79 25 25 24 23 22 23 23 24 18 2e 2d 26 44 72 89 8d 26 25 23 22 21 21 22 22 1c 2d 26 2a 58 80
    93 a5 25 24 22 21 20 20 21 21 1d 24 27 44 73 84 8d aa 23 22 21 20 20 20 21 21 1c 20 3a 69 89 85
    86 9c 20 20 1f 1f 1f 20 21 22 24 2c 54 80 8b 86 8b 91 1f 1f 1e 1e 1f 20 22 23 31 3c 67 84 7f 86
    95 91
** component Cb
* bloc 0
[  bloc] 2 3 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[iquant] 16 21 b 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[   izz] 16 21 0 0 0 0 0 0 b 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[  idct] 8a 8a 88 86 84 81 80 7f 8a 89 88 86 83 81 80 7f 8a 89 87 85 83 81 7f 7e 89 88 86 84 82 80
    7e 7d 88 87 86 84 81 7f 7e 7d 87 87 85 83 81 7e 7d 7c 87 86 84 82 80 7e 7c 7b 87 86 84 82 80 7e
    7c 7b
* component mcu
[   mcu] 8a 8a 8a 8a 88 88 86 86 84 84 81 81 80 80 7f 7f 8a 8a 89 89 88 88 86 86 83 83 81 81 80 80
    7f 7f 8a 8a 89 89 87 87 85 85 83 83 81 81 7f 7f 7e 7e 89 89 88 88 86 86 84 84 82 82 80 80 7e 7e
    7d 7d 88 88 87 87 86 86 84 84 81 81 7f 7f 7e 7e 7d 7d 87 87 87 87 85 85 83 83 81 81 7e 7e 7d 7d
    7c 7c 87 87 86 86 84 84 82 82 80 80 7e 7e 7c 7c 7b 7b 87 87 86 86 84 84 82 82 80 80 7e 7e 7c 7c
    7b 7b
** component Cr
* bloc 0
[  bloc] 1 fffe 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[iquant] b ffea 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[   izz] b ffea 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[  idct] 7e 7e 7f 81 82 84 85 85 7e 7e 7f 81 82 84 85 85 7e 7e 7f 81 82 84 85 85 7e 7e 7f 81 82 84
    85 85 7e 7e 7f 81 82 84 85 85 7e 7e 7f 81 82 84 85 85 7e 7e 7f 81 82 84 85 85 7e 7e 7f 81 82 84
    85 85
* component mcu
[   mcu] 7e 7e 7e 7e 7f 7f 81 81 82 82 84 84 85 85 85 85 7e 7e 7e 7e 7f 7f 81 81 82 82 84 84 85 85
    85 85 7e 7e 7e 7e 7f 7f 81 81 82 82 84 84 85 85 85 85 7e 7e 7e 7e 7f 7f 81 81 82 82 84 84 85 85
    85 85 7e 7e 7e 7e 7f 7f 81 81 82 82 84 84 85 85 85 85 7e 7e 7e 7e 7f 7f 81 81 82 82 84 84 85 85
    85 85 7e 7e 7e 7e 7f 7f 81 81 82 82 84 84 85 85 85 85 7e 7e 7e 7e 7f 7f 81 81 82 82 84 84 85 85
    85 85

...
```

FIGURE 3.2 – Extrait d'un fichier `.blabla` généré par `./jpeg2blabla` sur une image avec sous-échantillonnage horizontal. Pour chaque composante de chaque MCU (la 187[e] ici, de taille 2×1 blocs) on trouve le(s) blocs décompressé(s), puis après quantification inverse, zig-zag inverse et iDCT, et enfin la composante de la MCU reconstruite après *upsampling*. Ça pique les yeux au début, mais après quelques jours vous apprécierez !

3.4 Extension : iDCT rapide

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

Pourtant, vous aurez peut-être remarqué que le temps d'exécution de votre `jpeg2ppm` semble (nettement) plus important que celui du `jpeg2blabla` distribué, qui écrit pourtant de nombreuses informations dans un fichier texte (ce qui est trèèèès gourmand en temps). Alors ?

Vous pouvez commencer par utiliser un *profiler*, pour analyser le coût des étapes de votre décodeur. 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 ./jpeg2ppm 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 d'iDCT, transformée en double cosinus inverse. En effet une rapide analyse de l'algorithme présenté section 2.6 vous montre qu'il est de complexité quadratique $O(n^2)$. S'il est une étape à optimiser en priorité, c'est celle-ci.

Une première optimisation est de précalculer et stocker tous les cosinus nécessaires. La seconde est de 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 $O(n \log n)$. Des versions optimales existent en plus dans le cas particulier de l'iDCT sur un bloc 8×8. Dans ce projet, nous vous proposons d'utiliser l'algorithme de Loeffler, complexe à comprendre mais assez simple à implanter (☆☆☆ ou ☆☆☆☆). Vous travaillerez directement à partir de la publication originelle de cette méthode.

Si vous vous ennuyez après ceci, venez nous voir on peut encore trouver encore des tas de choses intéressantes (dans la norme JPEG, encodeur ?, etc.).

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.

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

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 décodeur !

Chapitre 4

Travail demandé

4.1 Objectif

L'objectif de ce projet est de développer un décodeur `jpeg2ppm` 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 décodeur complet ! Il devra être capable de décoder toutes les images JPEG, et les modules fournis initialement auront été réécrits 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 7 juin 2017 à 19h**.

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 (inutile de joindre les images fournies). Ne joignez que les images JPEG, pas les PPM (bien trop volumineux) ;
- tout autre élément pertinent de votre projet ;
- si nécessaire, joignez un fichier `README` décrivant le contenu de l'archive et toute information nécessaire à son utilisation.

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 8 juin après-midi** et **vendredi 9 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.

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

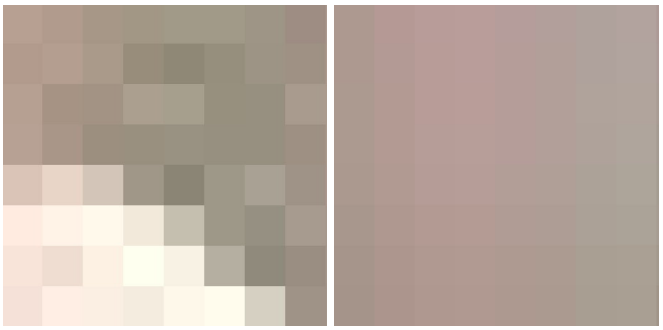
Annexe A

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 qui décrit les étapes du *dé*codage !

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 :

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

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

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

A.3 Sous échantillonnage

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

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

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

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

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

A.4 DCT : passage au domaine fréquentiel

La DCT est ensuite appliquée à chacun des blocs de données, après avoir soustrait 128 [2] aux valeurs. Les basses fréquences sont en haut à gauche et les hautes fréquences en bas à droite.

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

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

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

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

A.5 Quantification

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

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

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

Après quantification, les blocs de la MCU sont finalement :

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

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

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

A.6 Réordonnancement Zig-zag

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

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

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

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

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

A.7 Codage différentiel DC

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

Valeur DC En supposant que la MCU de cet exemple est la première du fichier (en haut à gauche de l'image), la valeur 78 doit être encodée en premier. Elle appartient à la classe de magnitude 7 : $-147, \ldots, -64, 64, \ldots, 127$. Dans cette classe, l'indice de 78 est `1001110`

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 : `111101001110`

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

A.8 Codage AC avec RLE

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

Pour le premier coefficient -2 :

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

— Dans la classe de magnitude 2, l'indice en binaire de -2 est `01`.

Le prochain coefficient non nul est -1 :

— Le symbole sur un octet est ici `0x21` : 2 sur les poids forts car le coefficient est précédé d'une séquence de deux 0, et 1 en poids faible car -1 est de magnitude 1 ;

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

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 `1110` pour `0x02`, `111110` pour `0x21` et `01` pour `0x00`, l'ensemble des 63 coefficients AC du bloc est totalement encodé dans le flux par les 15 bits : `111001111110001`. [3]

3. Sans codage de Huffman, la séquence aurait été `00000010 01 00100001 0 00000000`, soit 27 bits au lieu de 15 ! Et brutalement, sans RLE ni magnitude, il aurait fallu 64 octets...

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).

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. La seule chose qui nous intéresse est la séquence des 4 premiers octets de la section, qui doit contenir la phrase « JFIF ».

Offset	Taille (octets)	Description
`0x00`	2	Marqueur APP0 (`0xffe0`)
`0x02`	2	Longueur de la section
`0x04`	5	'J' 'F' 'I' 'F' '\0'
`0x09`	?	Données spécifiques au JFIF, non traitées

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 une seule section DQT avec plusieurs tables, ou 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 2.5.3)

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. **Vous devrez donc traiter les cas où les identifiant sont quelconques, entre 0 et 255**.

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 2.4.1. 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. 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. Dans le module `jpeg_reader` fourni, le nombre d'octets lus pour construire une table est disponible dans le paramètre de sortie `nb_byte_read`.

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* 2.4.1)

B.2.7 SOS - *Start Of Scan*

La section SOS marque le début du décodage effectif du flux JPEG, c'est-à-dire le début des données brutes 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. Mais puisque certains encodeurs ne respectent pas cette convention, seul l'identifiant i_C de la composante lue permet de l'associer correctement à la bonne composante de couleur.

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.

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.

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 (non utilisé)
...	1	Se : Fin de la sélection (non utilisé)
...	1	Ah : 4 bits, Approximation successive, poids fort Al : 4 bits, Approximation successive, poids faible

B.3 Récapitulatif 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

Trois modules sont fournis dans ce projet, sous forme d'une spécification (fichier `.h`) et d'un fichier objet binaire compilé (extension `.o`). Le code source n'est pas distribué.

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

C.1 Lecture des sections JPEG : module `jpeg_reader`

Ce module, d'entête `jpeg_reader.h`, fournit des fonctions pour accéder aux paramètres définis dans les sections d'un fichier JPEG décrites en annexe B. Ce module ne permet de lire que des fichiers JPEG encodés en mode {JFIF, baseline séquentiel, DCT, Huffman}. Seuls les paramètres principaux nécessaires à votre décodeur 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).

```
enum component {
    COMP_Y,
    COMP_Cb,
    COMP_Cr,
    /* sentinelle */
    COMP_NB
};

enum direction {
    DIR_H,
    DIR_V,
    /* sentinelle */
    DIR_NB
};

enum acdc {
    DC = 0,
    AC = 1,
    /* sentinelle */
```

```
    ACDC_NB
};
```

Type de données : descripteur Le module déclare une structure `struct jpeg_desc`, un descripteur contenant toutes les informations lues 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_reader.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 `get_XXX` pour accéder aux informations.

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

Ouverture, fermeture et fonctions générales

```
extern struct jpeg_desc *read_jpeg(const char *filename);

extern void close_jpeg(struct jpeg_desc *jpeg);

extern struct bitstream *get_bitstream(const struct jpeg_desc *jpeg);

extern char *get_filename(const struct jpeg_desc *jpeg);
```

- `read_jpeg` ouvre le fichier JPEG, lit tous les entêtes de sections rencontrés et stocke les informations requises dans le descripteur retourné. La lecture est stoppée au début des données brutes codant l'image (après l'entête de la section `SOS`) ;
- La fonction `get_bitstream` retourne l'adresse d'une variable de type `struct bitstream` (voir section C.2) permettant de lire les données brutes d'image encodées dans le flux ;
- La fonction `get_filename` retourne le nom de fichier de l'image ouverte ;
- `close_jpeg` ferme un descripteur préalablement ouvert, en libérant toute la mémoire nécessaire.

Le schéma général d'utilisation est toujours :

```
// ouvrir le fichier
struct jpeg_desc *jpeg = read_jpeg(filename);
// tester que jpeg n'est pas NULL (erreur)
...
// acceder aux differents parametres via les fonctions get_XXX
uint8_t nb_huffman_tables = get_nb_huffman_tables(jpeg, AC);
...
// ciao
close_jpeg(jpeg);
```

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.

Tables de quantifications Les données issues des sections DQT sont accessibles via :

```
extern uint8_t get_nb_quantization_tables(const struct jpeg_desc *jpeg);

extern uint8_t *get_quantization_table(const struct jpeg_desc *jpeg,
                                       uint8_t index);
```

— *get_nb_quantization_tables*(*jpeg*) retourne le nombre de tables de quantifications ;
— *get_quantization_table*(*jpeg*, *i*) retourne l'adresse mémoire de la i^e^ table de quantification, un tableau de 64 octets non-signés. Ce tableau est alloué par la fonction *read_jpeg* et sera libéré par *close_jpeg* ; ne le libérez pas vous-même !

Tables de Huffman Les données issues des sections DHT sont accessibles via :

```
extern uint8_t get_nb_huffman_tables(const struct jpeg_desc *jpeg,
                                     enum acdc acdc);

extern struct huff_table *get_huffman_table(const struct jpeg_desc *jpeg,
                                            enum acdc acdc,
                                            uint8_t index);
```

— *get_nb_huffman_tables*(*jpeg*, *acdc*) retourne le nombre de tables de Huffman de la composante *acdc* (*DC* ou *AC*) ;
— *get_huffman_table*(*jpeg*, *acdc*, *i*) retourne un pointeur vers la i^e^ table de Huffman de la composante DC ou AC. Le type de cette table est `struct huff_table`, fourni par le module huffman décrit en section 2.4.1. Ces tables sont allouées par la fonction *read_jpeg* et libérées par *close_jpeg*, ne les libérez pas vous-même !

Données du *Frame Header* Les données issues de la section SOF0 sont accessibles via :

```
extern uint16_t get_image_size(struct jpeg_desc *jpeg,
                               enum direction dir);

extern uint8_t get_nb_components(const struct jpeg_desc *jpeg);

extern uint8_t get_frame_component_id(const struct jpeg_desc *jpeg,
                                      uint8_t frame_comp_index);

extern uint8_t get_frame_component_sampling_factor(
                                      const struct jpeg_desc *jpeg,
                                      enum direction dir,
                                      uint8_t frame_comp_index);

extern uint8_t get_frame_component_quant_index(
                                      const struct jpeg_desc *jpeg,
                                      uint8_t frame_comp_index);
```

— *get_image_size*(*jpeg*, *dir*) retourne la taille de l'image (nombre de pixels) dans la direction *dir* (horizontale ou verticale) ;

— `get_nb_components` retourne le nombre de composantes de couleur (1 pour une image en niveau de gris, 3 pour une image en YCbCr, ...) ;
— `get_component_id(jpeg, i)` retourne l'identifiant de la i^e composante définie dans le *Frame Header* ;
— `get_component_sampling_factor(jpeg, dir, i)` retourne le facteur de sous-échantillonnage dans la direction `dir` de la i^e composante de couleur ;
— `get_component_quant_index(jpeg, i)` retourne l'index de la table de quantification de la i^e composante de couleur.

Données du *Scan Header* Les données issues de la section SOS sont accessibles via :

```
extern uint8_t get_scan_component_id(const struct jpeg_desc *jpeg,
                                     uint8_t scan_comp_index);

extern uint8_t get_scan_component_huffman_index(
                                     const struct jpeg_desc *jpeg,
                                     enum acdc acdc,
                                     uint8_t scan_comp_index);
```

— `get_scan_component_id(jpeg, i)` retourne l'identifiant de la i^e composante lue dans le scan. Cet identifiant doit être un des identifiants de composantes définis dans le *Frame Header*.
— `get_scan_component_huffman_index(jpeg, acdc, i)` retourne l'index de la table de Huffman (DC ou AC) associée à la i^e composante lue dans le scan.

C.2 Lecture bit à bit dans le flux : module `bitstream`

Ce module permet de lire des bits dans le fichier d'entrée (et non des octets comme dans une lecture standard), pour extraire les informations du format JFIF (annexe B) ou directement les données brutes de l'image compressée. Notez que ce module est très facile à tester, de manière totalement indépendante du reste du projet JPEG.

```
struct bitstream;

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

extern void close_bitstream(struct bitstream *stream);

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

extern bool end_of_bitstream(struct bitstream *stream);

/* Optionnel! */
extern void skip_bitstream_until(struct bitstream *stream,
                                 uint8_t byte);
```

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

La fonction `close_bitstream` 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;
stream = create_bitstream("broquedis.jpg");
...
uint32_t bits;
uint8_t nb_read = read_bitstream(stream, 5, &bits, false);
...
close_bitstream(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 `read_jpeg` et `close_jpeg` du module `jpeg_reader`.

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

- le flux de bits `stream` ;
- le nombre de bits `nb_bits` devant être lus, au maximum 32 ;
- `dest` contient l'adresse d'un entier 32 bits non signé dans lequel sont stockés les bits lus dans le flux. Si 11 bits ont été lus, le dernier est le bit de poids faible de `*dest` alors que le premier lu est à la position 10 ;
- un booléen `discard_byte_stuffing` qui indique s'il faut tenir compte ou non du *byte stuffing* décrit section 2.4.4. S'il vaut `true`, tout octet `0x00` suivant immédiatement un octet `0xff` sur une adresse alignée (multiple de 8) doit être ignoré lors de la lecture. Sinon, les bits de cet octet `0x00` seront lus normalement.
- la fonction retourne finalement le nombre de bits ayant été effectivement lus. Ceci permet de vérifier que la lecture s'est déroulée correctement (comme pour un `fread` classique).

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

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

Enfin, la procédure `skip_bitstream_until` permet d'avancer dans le flux jusqu'à trouver un octet égal à `byte` sur une adresse *alignée* (multiple de 8). S'il est trouvé, le flux est positionné juste *avant* cet octet. Les bits parcourus pour arriver jusqu'à `byte` sont perdus. Si la valeur n'est pas trouvée, le flux a normalement été entièrement parcouru. Cette fonction est *optionnelle* : vous n'en aurez pas

obligatoirement besoin dans votre décodeur. Elle est surtout utile pour dérouler le flux et se positionner sur le marqueur JPEG suivant. Ceci permet de s'abstraire de l'interprétation d'un marqueur non implanté dans le décodeur ou inutile au décodage, par exemple un commentaire associé à l'image.

Exemple Soit par exemple un flux contenant les données : `a3 04 5b ff ab...` :
- des appels successifs à *read_bitstream* pour lire 2, 1 puis 11 bits retourneront les séquences : `10`, 1 et `00011000001`.
- *skip_bitstream_until*(*stream*, `0xff`) avancera le flux jusqu'à l'octet `0xff` suivant, sur une adresse alignée. Au passage 10 bits du flux seront perdus.
- un nouvel appel à *read_bitstream* pour lire 13 bits retournera donc `1111111110101`.

C.3 Gestion des tables de Huffman : module `huffman`

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

```
struct huff_table;

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

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

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

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

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

Cette fonction retourne l'adresse d'une structure de type `struct` *huff_table*, qui a été crée puis remplie selon la méthode décrite en section 2.4.1. Après l'exécution, l'entier pointé par le paramètre *nb_byte_read* contient le nombre d'octets qui ont été lus dans le flux pendant la création de l'arbre.

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

Lecture de la prochaine valeur La fonction *next_huffman_value* retourne la prochaine valeur atteinte en parcourant la table de Huffman *table* selon les bits extraits du flux *stream*. Le parcours se fait comme décrit section 2.4.1 : on descend dans la feuille de gauche à chaque bit `0` et dans celle de droite à chaque `1`, jusqu'à atteindre une feuille contenant une valeur. Dans notre cas cette valeur, codée sur un octet non signé, représente la magnitude du prochain coefficient DC ou un symbole RLE des coefficients AC. Ensuite, le parcours redémarre depuis la racine de l'arbre de Huffman.

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

Dans le premier temps où vous utiliserez tous nos modules, notez que la création et la destruction des tables de Huffman est faite dans les fonctions *`read_jpeg`* et *`close_jpeg`* du module `jpeg_reader`.