

## 4. 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.

### 4.1 Spécifications

#### 4.1.1 Décodeur JPEG baseline

Les spécifications sont très simples : vous devez implémenter un programme qui convertit une image JPEG baseline séquentiel 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 sequential, DCT, Huffman, 8 bits} sont acceptées (application de type APP0, frame SOF0), avant d'éventuelles extensions ;
- 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.
- vérifiez bien que l'image de sortie est dans le même répertoire que l'image d'entrée. Par exemple une image dont le nom est `"../..images/zut.jpg"` sera décodée dans `"../..images/zut.ppm"`

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. Vous pouvez si le souhaitez ajouter des options, par exemple `-v` (verbose) pour afficher des informations supplémentaires.<sup>1</sup>

#### ❗ Respect des spécifications

Il est très important de respecter ces consignes simple, pour que tous les projets puissent être facilement évalués par des tests automatiques et par les enseignants lors des soutenances.

Si votre programme s'appelle plutôt `mon_decodeur`, qu'il attend des paramètres bizarres, et qu'il génère une image décodée qui s'appelle toujours `toto.ppm`, ce sera vite ingérable. Ce qui sera mauvais pour notre humeur, et donc *in fine* pour votre note...

### 4.1.2 Décodeur JPEG progressif

Dans un second temps (si vous avancez bien, hein) votre décodeur sera étendu pour accepter également des images JPEG en mode *{JFIF, progressive, DCT, Huffman, 8 bits}*, application de type `APP0`, frame `SOF2`. Vous pourrez sauvegarder une image PPM après chaque scan, de qualité croissante. Si l'envie (et le temps) vous en prend, vous pourriez aussi afficher les données décodées directement dans une fenêtre graphique.

**Attention cependant, la gestion du JPEG progressif est considérée comme une extension du projet, et ne sera évaluée à la soutenance que si votre décodeur JPEG baseline est terminé.** Plus d'infos là-dessus dans la section sur les extensions possibles sur cette page.

### 4.1.3 Format de fichier 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 peut-être déjà eu l'occasion d'utiliser dans la partie préparation au langage C et dans le projet de BPI.

Le principe est de lire d'abord un en-tête *textuel* comprenant:

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

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

```
$ 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|
```

En haut, trace de la commande `hexdump -C` sur une image  $3 \times 2$  représentant un drapeau français. Notez les caractères ASCII de l'en-tête 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.

## 4.2 Organisation, démarche conseillée

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

### 4.2.1 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 :
  - 2.1. Reconstruction de chaque bloc :
    - Extraction (lecture dans le flux de bits) et décompression ;
    - Quantification inverse (multiplication par les tables de quantification) ;
    - Réorganisation zig-zag ;
    - Calcul de la transformée en cosinus discrète inverse (iDCT) ;
  - 2.2. Mise à l'échelle des composantes Cb et Cr (*upsampling*), en cas de sous-échantillonnage ;
  - 2.3. Reconstruction des pixels : conversion YCbCr vers RGB ;
- 3. Ecriture du résultat dans le fichier PPM.

Vous trouverez ci-dessous une estimation de la difficulté de ces différentes étapes :

Etape	Difficulté pressentie
Lecture en-tête JPEG	★★★ à ★★★★★
Gestion des tables de Huffman	★★★★★
Extraction des blocs	★★★
Quantification inverse	★
Zig-zag	★★
iDCT	★★
<i>Upsampling</i>	★★★★
Conversion YCbCr vers RGB	★
Ecriture du fichier PPM	★★ à ★★★★★
Gestion complète du décodage	★★ à ★★★★★★

Pour faire tout ça vous allez aussi devoir lire dans le flux de données, parfois pour lire des octets mais souvent pour lire 3 bits, 11 bits, 1 bit, etc. Humm, ça vaut bien "quelques" ★!

Pour l'extension au mode progressif, à vous de nous mettre des ★ plein les yeux !

### 4.2.2 Progression incrémentale 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 implémenter 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 plus d'être efficace et rationnelle, cette approche permet d'avoir toujours un programme fonctionnel, même incomplet, à présenter à votre client (nous), à tout instant. Imaginez que le rendu soit subitement avancé de deux jours<sup>2</sup>, 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<sup>3</sup>!

Plusieurs images de tests sont fournies, de "complexité" croissante. Il est conseillé de travailler sur ces images dans l'ordre suggéré (dans l'ordre du tableau), permettant une progression graduelle d'un décodeur simple vers celui capable de traiter des images quelconques.













Image	Caractéristiques	Progression
 invader	<ul style="list-style-type: none"> <li>• 8x8 (un seul bloc)</li> </ul>	<ul style="list-style-type: none"> <li>• Niveaux de gris (une seule composante)</li> <li>• Encodage MCU très simple</li> <li>• PPM simplifié</li> </ul>
 poupoupidou_bw	<ul style="list-style-type: none"> <li>• 16x16</li> </ul>	<ul style="list-style-type: none"> <li>• Niveaux de gris</li> <li>• Plusieurs blocs</li> <li>• Pas de troncature</li> </ul>
 gris	<ul style="list-style-type: none"> <li>• 320x320</li> </ul>	<ul style="list-style-type: none"> <li>• Niveaux de gris</li> <li>• Plusieurs blocs</li> <li>• Pas de troncature</li> <li>• Je sais plus pourquoi, mais des fois cette im</li> </ul>
 bisou	<ul style="list-style-type: none"> <li>• 585x487</li> </ul>	<ul style="list-style-type: none"> <li>• Niveaux de gris</li> <li>• Plusieurs blocs</li> <li>• Troncature à droite et en bas</li> </ul>
 poupoupidou	<ul style="list-style-type: none"> <li>• 16x16</li> </ul>	<ul style="list-style-type: none"> <li>• Couleur</li> <li>• Pas de troncature</li> <li>• Glamour</li> </ul>
 zig-zag	<ul style="list-style-type: none"> <li>• 480x680</li> </ul>	<ul style="list-style-type: none"> <li>• Couleur</li> <li>• Pas de troncature</li> </ul>

Image	Caractéristiques	Progression
 thumbs	<ul style="list-style-type: none"> <li>• 439x324</li> </ul>	<ul style="list-style-type: none"> <li>• Couleur</li> <li>• Tronquée à droite et/ou en bas</li> </ul>
 horizontal	<ul style="list-style-type: none"> <li>• 367x367</li> </ul>	<ul style="list-style-type: none"> <li>• Echantillonnage horizontal</li> </ul>
 vertical	<ul style="list-style-type: none"> <li>• 704x1246</li> </ul>	<ul style="list-style-type: none"> <li>• Echantillonnage vertical</li> </ul>
 shaun_the_sheep	<ul style="list-style-type: none"> <li>• 300x225</li> </ul>	<ul style="list-style-type: none"> <li>• Echantillonnage horizontal ET vertical</li> </ul>
 complexite	<ul style="list-style-type: none"> <li>• 2995x2319</li> </ul>	<ul style="list-style-type: none"> <li>• Niveaux de gris</li> <li>• C'est looong...</li> </ul>
 biiiiiig	<ul style="list-style-type: none"> <li>• 7392x3240</li> </ul>	<ul style="list-style-type: none"> <li>• Couleur</li> <li>• C'est loooooooooong !</li> </ul>



Un développement et une validation incrémentale pour la niveaux de gris serait donc:

1. Décodeur d'images 8x8 en niveaux de gris (ex: `invader.pgm`) ;
2. Extension à des images grises comportant plusieurs blocs (ex: `gris.pgm`) ;
3. Extension à des images dont les dimensions ne sont pas multiples de la taille d'un bloc (ex: `bisou.pgm`) ;

Et bien sûr on procèdera de la même façon pour étendre le décodeur à la gestion des images couleur de complexité croissante.

Ne sous-estimez la longueur/difficulté de l'étape 1, à savoir implémenter un décodeur d' `invader.jpg` ! Certes, commencer par travailler sur cette image simplifie beaucoup le décodage : elle ne contient qu'un seul bloc 8x8, et est en noir et blanc. En revanche, le décodage de cette image nécessite d'avoir mis en place la majeure partie de l'architecture de votre décodeur. Vous devrez en particulier être capable de récupérer les informations nécessaires dans l'entête JPEG avant même de commencer à décoder les données de l'image.

Plus généralement, notez que ces images doivent vous aider à valider certaines parties du projet, **mais ne seront pas suffisantes tester tout votre décodeur**. Il sera en particulier utile/nécessaire d'ajouter des **tests unitaires** pour tester hors contexte des parties spécifiques du décodeur (par exemple est-il possible de valider l'iDCT en tant que telle, pas au milieu du décodage?). Il sera aussi nécessaire d'ajouter **vos propres images** à cette base de tests, par exemple conçues de toute pièce avec `gimp` présentant des caractéristiques spécifiques . En plus de vous aider lors de la mise au point de certaines fonctionnalités du décodeur, vous pourrez vous appuyer sur ces images "*maison*" lors de la soutenance pour démontrer la robustesse de votre implémentation.

### ! Le piratage c'est du vol

Toutes ces images ne sont pas franchement libres de droits. Mais on les aime bien, alors au moins faisons leur un copyright à notre sauce: relisez vos BDs préférées, parlez l'anglais et l'américain, revoyez Les Temps Modernes, Bêêhh, aimez la physique (à défaut de comprendre), jouez à des jeux d'hier et d'aujourd'hui, Some Like It Hot, et bécotez-vous devant l'hôtel de ville! (mais pas de trop près en ce moment bien sûr. Euh... attends)

### ! Blague Carambar

L'image "vertical" traduit l'émotion d'un étudiant ou d'une étudiante qui vient d'arriver à finir le *downsampling* à peu près correctement. L'image "horizontal" traduit la réaction de ses enseignants. La première personne qui nous dit de quel(s) album(s) proviennent ces deux images gagne un Carambar! (il y a un piège). Et ce qui est chouette cette année avec le projet en distanciel, c'est que vous gagnez bien le Carambar mais c'est un ou une prof qui le mange!

## 4.2.3 Découpage en modules & fonctions, spécifications

Par rapport à la plupart des TPs réalisés cette année, une des difficultés de ce projet est sa *taille*, c'est-à-dire 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 l'écriture de code, il est essentiel de bien définir un découpage en *modules* et en *fonctions* pour les différents éléments à réaliser. Ils doivent être clairement **spécifiés** : rôle, structures de données éventuelles, fonctions proposées et leur signature précise.

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

Prenons pour exemple l'étape DCT. Il est naturel<sup>4</sup> 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 8x8 ?<sup>5</sup> Adresse dans un tableau de plus grande taille ? ...
- Quel est le type des éléments d'un bloc à ce stade du décodage ?
- ...

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 (et c'est bien 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 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 `jpeg2b1ab1a` distribué (voir sa description juste après !).

### ❗ Point de passage obligatoire

Pour éviter les déconvenues du genre *"On est à une semaine du rendu, mon projet est cassé et je n'arrive plus à le corriger parce que mes structures de données que j'ai construites au départ sont mal foutues."* (toute ressemblance avec ce qu'on entend tous les ans est fortuite), vous **devrez** organiser un point d'étape avec un enseignant, où vous lui présenterez l'architecture logicielle de votre projet (vos structures de données, le découpage du projet en modules, le découpage des modules en fonctions). De notre côté, on s'engage à dire tout le mal qu'on pense de ce que vous nous montrerez, ce qui vous



permettra de rectifier le tir avant de vous lancer dans l'implémentation. Bien entendu, plus ce point d'étape a lieu tôt dans le projet, mieux c'est, mais **ce sera à vous de venir nous voir quand vous vous sentirez prêts.**

## 4.3 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 d'informations :

- **Paramètres de l'image** : l'option `-v` (pour verbose) affiche dans la console les caractéristiques de l'image JPEG à décoder (celles utiles en mode baseline séquentiel DCT). Un exemple est donné figure 4.2. L'option `-vh` (ou `-v -h`) permet en plus d'afficher la structure des arbres de Huffman, c'est-à-dire les chemins menant aux différents symboles. Un exemple est donné figure 4.3.
- **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. Un exemple est donné figure 4.4.

Ces traces pourront s'avérer trèèèèèè utiles (sur des images de taille réduite) pour valider votre décodeur étape après étape, si nécessaire.

```

[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 (hvx) 1x1
            quantization table index 0
[DHT]   length 31 bytes
        Huffman table type DC
        Huffman table index 0
        total nb of Huffman symbols 12
[DHT]   length 81 bytes
        Huffman table type AC
        Huffman table index 0
        total nb of Huffman symbols 62
[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)
        End of Scan Header (SOS)

... (image decompression) ...
... Done
[EOI]   marker found
bitstream empty

```

Figure 4.2 : Trace de `./jpeg2blabla -v bisou.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 AC et une DC) définies dans des sections DHT séparées.

```

...
[DHT] length 30 bytes
    Huffman table type DC
    Huffman table index 0
    total nb of Huffman symbols 11
    path: 000 symbol: 2
    path: 001 symbol: 3
    path: 010 symbol: 4
    path: 011 symbol: 5
    path: 100 symbol: 6
    path: 101 symbol: 7
    path: 110 symbol: 8
    path: 1110 symbol: 1
    path: 11110 symbol: 9
    path: 111110 symbol: 0
    path: 1111110 symbol: a
...

```

Figure 4.3 : Trace (partielle) de `./jpeg2blabla -vh bisou.jpeg`. L'option `-h` permet de voir la structure des arbres de Huffman, ici un de type DC.

Voyons maintenant un exemple de trace complète de décodage, dans un fichier `.blabla`. Pour chaque composante de MCU et pour chaque bloc, on trouvera d'abord une première ligne comme:

```

[ DC/AC] 07(4) / 03(6) 04(3) 51(7) 01(3) f0(8) 21(5) 11(4) 41(6) a1(8)
11(4) 81(8) 00(4) -- total 92 bits

```

Comment lire ceci ?

- la magnitude du coefficient DC est 0x07, elle a été lue en consommant 4 bits (== profondeur du symbole 0x07 dans l'arbre de Huffman)
- le premier coefficient AC est 0x03 (6 bits lus), le second 0x4 (3 bits lus), etc.
- au total 92 bits ont été consommés dans le flux pour récupérer tous les coefficients du bloc (nb de bits lus dans les arbres + les magnitudes trouvées)

Ensuite, et toujours pour chaque bloc on trouvera:

- les blocs décompressés, puis après quantification inverse, zig-zag inverse et iDCT
- la composante de la MCU reconstruite à partir du ou des blocs, après éventuel *upsampling*.

La figure ci-dessous fournit un exemple complet d'une MCU en couleur.

[illegible]

```

ffff 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 ffff 0 0 0 0 0 0 0 0 0
[ izz] 6 fff7 ffff ffff 0 0 0 0 4 1 ffff 1 1 0 0 1 fff9 0 0 ffff 0 0 0 0 2 1 ffff 0 0 0 0 ffff
ffff 0 0 0 0 0 0 0 1 0 ffff 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
[ idct] 7f 7f 80 81 82 81 81 81 7f 7f 80 80 81 81 82 82 80 80 81 81 83 82 83 83 80 80 81 81 83 83
83 83 80 81 81 82 82 83 83 83 7f 80 80 81 81 82 82 83 7e 7f 80 80 80 81 81 82 7c 7d 7e 7e 7e 7f
80 81
* component mcu
[ mcu] 7f 7f 7f 7f 80 80 81 81 82 82 81 81 81 81 81 81 7f 7f 7f 7f 80 80 80 80 81 81 81 81 82 82
82 82 80 80 80 80 81 81 81 81 83 83 82 82 83 83 83 83 80 80 80 80 81 81 81 81 83 83 83 83 83
83 83 80 80 81 81 81 81 82 82 82 82 83 83 83 83 83 83 7f 7f 80 80 80 80 81 81 81 81 82 82 82 82
83 83 7e 7e 7f 7f 80 80 80 80 80 80 81 81 81 81 82 82 7c 7c 7d 7d 7e 7e 7e 7e 7e 7f 7f 80 80
81 81
...

```

Figure 4.4 : Extrait d'un fichier `.bLabLa` généré par `jpeg2bLabLa` sur une image avec sous-échantillonnage horizontal. Pour chaque composante de chaque MCU (la 787e ici, de taille 2x1 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 !

❗ Monsiieeur, Madaaammme, j'ai pas pareil.

Il se peut que les valeurs numériques diffèrent légèrement entre votre décodeur et `jpeg2bLabLa`. Il peut s'agir d'une erreur bien sûr, mais aussi de différences purement numériques en particulier lors de calculs en valeurs flottantes : précision, pas les même arrondis, etc.

Pour le cas particulier des résultats après DCT, sachez que `jpeg2bLabLa` s'appuie sur l'algorithme de Loeffler pour calculer la DCT. La DCT Loeffler est plus rapide mais ne fournit pas exactement les mêmes valeurs que la DCT naïve.

`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 en [section 3.1.2](#). 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 implémenterez la lecture de l'entête JPEG). Vous pourrez également l'utiliser pour regarder le contenu bit à bit d'un fichier et vérifier que les bits lus par votre décodeur 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.

`cjpeg`

`cjpeg` est un encodeur JPEG. Développé initialement par le Independent JPEG Group 5, cet encodeur se positionne comme l'implémentation de référence du standard JPEG. Il vous permettra notamment de contrôler certains aspects "ceinture noire" de la norme lors de la conversion des images, comme les facteurs d'échantillonnage, les tables de quantification utilisées ou encore l'utilisation du mode progressif. Si vous souhaitez étendre la base d'images de tests fournie avec le projet, par exemple pour stresser votre décodeur (ou accessoirement, votre trinôme), ou simplement briller en société, c'est l'outil qu'il vous faut !

## 4.4 Des idées d'améliorations et extensions

Si vous lisez cette section, c'est que vous êtes venus à bout de votre décodeur pour toutes les configurations d'images JPEG.<sup>6</sup> Félicitations ! Mais pourquoi s'arrêter en si bon chemin, alors voici déjà deux améliorations possibles à regarder. Et si jamais vous vous ennuyez après ceci, venez nous voir<sup>7</sup> on a toujours d'autres trucs sous le coude!

### 4.4.1 A la recherche du temps perdu

Votre décodeur décode, cool. Pourtant, il est possible que le temps d'exécution de votre `jpeg2jpeg` soit (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?

On s'intéresse d'abord à l'*optimisation* de votre décodeur. Par optimisation, on entend ici *tout ce qui peut le rendre meilleur qu'un décodeur aux fonctionnalités équivalentes*. A niveau de fonctionnalité équivalent, on préférera évidemment un décodeur plus rapide ou qui consomme moins de mémoire<sup>8</sup>.

### Y'a les bons et les mauvais décodeurs

*Le mauvais décodeur, il voit une image, il la décode...*

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

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

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

L'optimisation d'un programme peut être vue comme un processus cyclique, illustré ici sur l'amélioration du temps d'exécution du décodeur :

- Analysez les traces du profiler pour identifier les parties à optimiser en priorité : quelles sont les étapes les plus coûteuses en temps ? Sont-elles améliorables ?
- Posez-vous des questions sur vos structures de données, efficacité des accès mémoires, nombres d'allocations, algorithmes, etc. Puis implémentez ces améliorations et évaluez les.
- **Vérifiez** que votre décodeur fonctionne toujours aussi bien! Ben oui, si ça va plus vite mais que ça marche plus, ça sert à rien. Une suite de tests (dits de non-régression) qui vérifie le bon fonctionnement de votre décodeur pourrait à ce stade être de bon goût.
- **GOTO 1.** Et oui, on peut faire ça à l'infini (ou en gros, jusqu'à qu'on ait soit atteint un niveau de performance souhaité, soit qu'on n'ait plus d'idée.).

Bien entendu, l'approche est transposable à d'autres types d'optimisation que l'amélioration du temps d'exécution.

## Exemple d'optimisation: une DCT plus rapide

Sauf surprise, il est très probable qu'au moins 80% du temps soit consommé par l'étape de DCT.

En regardant de plus près l'algorithme présenté en [section 2.4](#), il est clairement de complexité quadratique. Même en bossant depuis votre canapé ces dernières semaines vous aurez compris que ce n'est pas terrible. Donc la DCT est une étape prioritaire à optimiser.

Déjà, on se rend compte qu'on recalcule plusieurs fois les mêmes valeurs de cosinus. Une première optimisation de votre décodeur consiste donc à précalculer et stocker tous les cosinus nécessaires. Même si la complexité de l'algorithme est inchangée, vous allez déjà gagner énormément !

Pour aller plus loin, il faudra s'attaquer à l'algorithme en lui-même pour réduire le nombre d'opérations, les multiplications en particulier. Comme vous l'avez vu peut-être en cours d'Algorithmique, il existe des méthodes de type "Diviser pour régner" pour écrire une version efficace de la transformée de Fourier (*Fast Fourier Transform*, FFT) en  $\mathcal{O}(n \log n)$ . Des

versions optimales existent en plus dans le cas particulier de la DCT sur un bloc 8x8. Reste plus qu'à les trouver, les comprendre, les implémenter, les valider et se congratuler. Mais comme vous avez des profs extra, allez donc voir en [annexe D](#).

## 4.4.2 Mode progressif

Comme vous pouvez le constater, tout vous est donné pour réaliser sereinement un décodeur JPEG en mode séquentiel : des explications et exemples aux petits oignons, une proposition de progression incrémentale dans les spécifications et les tests, des belles images, des profs bienveillants, etc.

Pour le mode progressif, on change de paradigme : **débrouillez-vous**. Il vous faudra notamment comprendre la norme, spécifier les restructurations à apporter à votre code, les faire, générer des tests adéquats et valider votre décodeur. L'objectif final est bien d'avoir un programme qui décode les modes séquentiel ET progressif.

Mais attention le progressif c'est (très) dur. Donc :

1. Commencez par finir votre décodeur séquentiel et le valider ;
2. Ensuite seulement abordez la partie progressive. Inutile d'aborder ce mode à la soutenance si votre décodeur panique sur la 2ème image séquentielle testée...

## 4.5 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. Ce document restera votre livre de chevet préféré pour les 42 années à venir ;
2. <http://www.impulseadventure.com/photo> Ce site fournit une approche par l'exemple pour qui veut construire un décodeur JPEG baseline. On y retrouve des illustrations des différentes étapes présentées dans ce document. Les détails des tables de Huffman, la gestion du sous-échantillonnage, etc., sont expliqués avec des schémas et force détail, ce qui permet de ne pas galérer sur les aspects algorithmiques ;
3. Pour une compréhension plus poussée du sous-échantillonnage des chrominances, consulter <http://dougkerr.net/pumpkin/articles/Subsampling.pdf> ;
4. Pour les informations relatives au format JFIF, aller voir du côté de <http://www.ijg.org/>.



Parmi les informations que vous pourrez trouver sur le web, il y aura du code, mais il aura du mal à rentrer dans le moule que nous vous proposons. **L'examen du code lors de la soutenance sera sans pitié pour toute forme de plagiat.** Mais surtout assimiler ce type de code vous demandera au moins autant d'effort que de programmer vous-même votre propre décodeur !

---

1. Pour la gestion en C des paramètres optionnels d'un exécutable, renseignez-vous sur la famille des fonctions `getopt`. D'accord la `man page` est ce qu'elle est, mais entre comprendre cette doc et tout faire à la mimine, le choix devrait être simple... ↩
2. Vos enseignants sont parfois très joueurs! ↩
3. Au-delà de ce projet, ceci sera surtout valable dans votre vraie vie d'ingénieur, si si! Pensez à nous remercier le moment venu. ↩
4. Si ce n'est pas le cas encore, ça devrait ! ↩
5. Tip: moyennant une petite gymnastique sur les indices, cette représentation 1D simplifiera beaucoup l'expression et la manipulation des différentes fonctions à implémenter! ↩
6. Si ce n'est pas le cas, commencez par finir votre décodeur avant d'attaquer les extensions ! ↩
7. de pas trop près hein, on n'a toujours pas le droit... ↩
8. Vous verriez les performances du `jpeg2blabla` à côté de ce qu'on a en interne... Hein, Big Brother? ↩