Décodage Progressif :

Pour le décodage d'images encodées progressivement, nous allons essayé d'écrire un pseudo_code qui résume toutes les étapes/changements à faire par rapport au cas simple (baseline) faute de temps :(.
Néanmoins, nous avons écrit quelques bouts de codes qui accompagnent ce fichier texte (les codes sont eux aussi présents dans ce même dossier).

Alors, commençons déjà par les bases, le JPEG Progressive affiche l'image en basse résolution puis améliore celle-ci au fur et à mesure, à l'opposé du Baseline qui lui déroule l'image de haut en bas.

Ceci suggère évidemment un travail sur tous les MCU et blocs en même temps.

En se documentant un peu plus sur l'encodage progressif, on découvre que la plus grande différence réside dans le fait que les composantes sont encodées dans plusieurs scans au lieu d'une seule.

Il existe deux méthodes de diviser les composantes sur différents scans, l'une d'entre elles est nommée SPECTRAL SELECTION.
Cette dernière réfère à la division des composantes en bandes de coefficients DCT, chaque bande étant une gamme continue utilisant l'ordre zig-zag.

Ceci laisse songer qu'à l'inverse du Baseline, le processus Progressif agira sur non pas 64 valeurs d'un blocs en une seule passe, mais petit à petit sur des fragments de blocs, les traitant à chaque fois indépendamment et totalement avant de passer au fragment qui suit.

Aussi et pour capter les différences entre les deux modes (Sequentiel et Progressif) nous avons créé une image invader en progressif à l'aide de gimp et deux fichiers hexdump avec l'option -C, un pour l'invader classique et l'autre pour le progressif. Et après comparaison, nous avons pu voir la répétition de scans SOS (de marqueur 0xFFDA) ce qui suggère la division susmentionnée.

Pour y voir plus clair, voici par nos soins, en pseudo-code, un traitement grossier des images dans les deux cas Baseline et Progressif.

Baseline Sequential:

```
{
While (EXISTE_UN_SCAN){
   data = Read_scan();
        }
/* Fin de la boucle While */
data_apres_idct = IDCT(data);
data_apres_rgb = traitement_couleurs(data_apres_idct);
afficher_image(data_apres_rgb);
}
```

Progressive:

```
{
While (EXISTE_UN_SCAN){
   data = Read_scan();
   data_apres_idct = IDCT(data);
   data_apres_rgb = traitement_couleurs(data_apres_idct);
   afficher_image(data_apres_rgb);
        }
/* Fin de la boucle While */
}
```

On voit bien dans le pseudo_code que le traitement en entier doit se faire fragment par fragment...

Cela pose la question sur les changements à apporter sur le module jpeg_reader.c .

Les changements vont porter en majeur partie sur le traitement des sections SOS et SOF.
Commençons par SOF...
Pour cette section, qui avait pour marqueur 0xFFC0 qui concernait un traitement Baseline DCT (Huffman) va devoir aussi accepter le marqueur 0xFFC2 qui lui réfère au DCT Progressive (Huffman).

En ce qui concerne la section SOS, le changement adviendra à la fin de la fonction qui traite d'ores et déjà cette section.
Au lieu de sauter les trois derniers octets qui étaient insignifiants jusqu'à maintenant, il faut les lire et les stocker pour les utiliser plus tard.
Ainsi on rajoute dans la structure jpeg 4 champs, indice_debut, indice_fin, Approximation_high, et Approximation_low qui seront mis à jour chaque fois qu'on rencontre une section SOS pour qu'ils soient utilisés dans les fonctions décodant les MCUS.

Aussi un dernier changement mineur est dans la fonction read_jpeg, plus précisemment dans la boucle WHILE qui désormais ne s'arrête pas directement après la première section SOS lue, mais jusqu'à la lecture du marqueur de fin EOI 0xFFD9 vu qu'en il existe maintenant plusieurs (sections SOS).

(Pour voir le module jpeg_reader.c modifié, rdv le fichier existant dans ce dossier même).

Discutons maintenant un peu de l'utilité des 4 champs ajoutés ci dessus dans la structure JPEG.
Pour les indice_debut et indice_fin, ils déterminent (obviously) les indices de début et de fin des coefficients de la bande à lire pour pouvoir les traiter indépendamment juste après. En gros, ce sont les délimiteurs des fragments dont on a parlé un peu plus haut, ils appartiennent donc à l'intervalle [0,63] le premier indice_debut du tout premier fragment étant 0, et le dernier indice_fin du dernier fragment étant 63.
Aussi, si c'est deux valeurs sont 0, alors la section SOS concerne le corfficient DC, sinon, si les deux valeurs sont toutes deux différentes non nulles, le scan concerne les AC.

Reste à savoir maintenant l'utilité des A_high et A_low...
Mais avant cela, il faut savoir qu'il existe quatres types de scans:

1) First DC scan.
2) Refining DC scan.
3) First AC scan.
4) Refining AC scan.

Si A_high == 0, le scan est un first scan pour la bande, sinon c'est un Refining (Raffinage).
Et si les deux champs A_high et A_low sont égaux à 0, alors l'approximation successive est abandonée pour la bande.

Ceci donne beacoup d'asserts à vérifier lors de l'écriture du code principal, parmi ceux là:

{
 - if (indice_debut == 0) alors (indice_fin == 0).
 - if (indice_fin == 0) alors (indice_debut == 0).
 - assert(indice_fin >= indice_debut).
 - if (indice_fin != 0) alors le scan ne doit comporter qu'une seule composante.
 - assert(indice_fin <= 63).
}

Faisons maintenant un petit point sur l'usage des tables de Huffman dans ce mode.

La section SOS spécifie l'identificateur numérique des tables de Huffman utilisées parchaque composant. La norme JPEG exige que toutes les tables Huffman requises par une analyse doivent être définies avant la section SOS. En outre de ce qu'on a comprs, quand un décodeur gère des analyses progressives, il doit valider l'existence des tables de Huffman utilisées par le scan différemment de la façon dont il le fait avec un balayage séquentiel.
Normalement, chaque composant d'une analyse séquentielle nécessite deux tables de Huffman (DC et AC). En mode JPEG progressif, un scan soit utilise des tables Dc ou AC, mais pas les deux.
Et qu'en fait, selon notre un balayage continu progressif de raffinage (Refining - deuxième passage après first scan) n'utilise pas du tout les tables de Huffman.
Et on pense aussi qu'il est tout à fait possible qu'un balayage continu progressif DC se produise dans un fichier JPEG avant que toutes les tables AC Huffman ne soient définies, ce qui est illégal dans le mode séquentiel de JPEG.

Parlons maintenant un petit peu des fonctions des fonctions de traitement "quantification inverse", "zig-zag inverse" et "iDCT":
En ce qui concerne les deux premières, de changements doivent être effectués de sorte qu'on ne traite qu'un fragment à la fois. Les paramètres "indice_debut" et "indice_fin" sont ainsi ajoutés.
(Voir la nouvelle version des codes dans le dossier courant).

Quant à l'iDCT, plus particulièrement celui de loeffler, aucun changement ne peut être effectué car celui commence à chaque dès le début, mais cela ne cause aucun problème pour le décodage.

Passons maintenant plus en détails sur comment on envisageait de faire le décodages sur les différentes composantes suivant les types de scans.

PREMIERS SCANS DC:
Le processus est ici le même, ou presque.
La seule différence étant le fait qu'on doit désormais décaler à gauche la valeur DC de "Approximation successive" bits.

SCANS DC DE RAFFINAGE:
Là aussi, c'est un peu simple, le scan de raffinage est entièrement constitué de bits appartenant aux données brutes. Voilà un petit pseudo_code résumant cette étape:

```
{
RAFFINAGE_DC(){
  bit = read_bitstream(1); // lire un bit
  coefficients_DC[0] = coefficients_DC[0] || (bit << (APPROXIMATION_SUCCESSIVE);
        }
}
```

Pour les coefficients AC, les choses se corsent...
Pour l'instant, on n'a pas encore une idée très claire de comment cela se fait.
Mais on essaiera d'en savoir plus avant la soutenance le 11 juin.

Pour finir voilà ce qu'on pense être un pseudo_code complet pour décoder des images encodées progressivement :

```
{
DEBUT
READ_JPEG_PREMIERES_SECTIONS(); //Decode les premières sections jpeg (jusqu'à SOF de marqueur 0xFFC2)

/* Le stream est maintenant au début de la première section DHT (marqueur 0xFFC4) */

WHILE ( JDESC->FIN != FALSE){
        marqueur = LIRE_MARQUEUR();
        if (marqueur == 0xFFD9){  //0xFFD9 est le marqueur de fin d'image (EOI).
                JDESC->FIN = TRUE;
        }
        table_huffman = READ_SECTION_DHT();
        READ_SECTION_SOS();
        /*Maintenant qu'on a toutes les informations, on traite les blocs, MCU par MCU*/
        for (i = 0; i < nombre_mcu; i++){
         for (j = 0; j < nombre_de_blocs; j++){
         TRAITEMENT(blocs[j]);

         /* - Le traitement ci dessus se fait le type de scans (4 types déjà mentionnés).
            - Effectue la nouvelle version de la quantification inverse et zig zag inverse, et
            l'iDCT.*/

         CONVERSION_RGB(blocs);
         GENERER_IMAGE();

           /*On genere petit à petit l'image qui va s'éclaircir tout en avançant dans le
        décodage*/
           }
        }
}
```

Voilà Voilà pour le progressif...