Ensimag — Juin 2014

Projet Logiciel en C

Sujet Décodeur Vorbis

Auteurs : des enseignants actuels et antérieurs du projet C.

Table des matières

Chapitre 1

Généralités et organisation du projet

1.1 Objectifs du projet

Tout informaticien doit connaître le langage C. C'est une espèce d'espéranto de l'informatique. Les autres langages fournissent en effet souvent une interface avec C (ce qui leur permet en particulier de s'interfacer plus facilement avec le système d'exploitation) ou sont eux-mêmes écrits en C. D'autre part c'est le langage de base pour programmer les couches basses des systèmes informatiques. Par exemple, on écrit rarement un pilote de périphérique en Ada ou Java. Le langage C est un excellent langage pour les programmes dont les performances sont critiques, en permettant des optimisations fines, à la main, des structures de données ou des algorithmes (typiquement, les systèmes de gestions de base de données et d'une manière générale les logiciels serveurs sont majoritairement écrits en C). Finalement, en compilation, C est souvent choisi comme cible de langages de plus haut niveau.

Cependant, beaucoup d'entre vous ne seront que rarement, voire jamais, confrontés à de gros développements logiciels entièrement en C dans leur vie professionnelle. L'objectif pédagogique du projet logiciel en C est donc surtout de montrer comment C peut servir d'interface entre les langages de haut niveau et les couches basses de la machine. Plus précisément, les objectifs du projet logiciel en C sont :

— Apprentissage de C (en soi, et pour la démarche qui consiste à apprendre un nouveau langage).
— Lien du logiciel avec les couches basses de l'informatique, ici logiciel de base et architecture.
— Le premier projet logiciel un peu conséquent, à développer dans les règles de l'art (mise en œuvre de tests, documentation, démonstration du logiciel, partage du travail, etc.).

1.2 Déroulement du projet

1.2.1 Organisation du libre-service encadré

Pendant tout le libre-service encadré, il faut consulter régulièrement la page d'EnsiWiki du projet. [1] En effet cette page contient les informations de dernière minute sur le déroulement et l'organisation du projet. En particulier, il est impératif de consulter régulièrement (i.e., au moins une fois par jour) la page *Actualité du projet C - VORBIS*.

Pour réaliser le projet, les étudiants bénéficient d'un encadrement du travail en libre-service (voir la page EnsiWiki pour les détails des horaires). Pendant ces heures de libre-service encadré, des salles machines de l'Ensimag ont été réservées pour les étudiants participant au projet. De plus, les enseignants assurent une permanence pour aider les étudiants sur :

— la programmation en langage C.
— l'environnement de développement (make, autres outils gnu, etc.) et les programmes fournis.
— la conception du programme.
— l'organisation du projet.
— la compréhension générale des sujets donnés aux étudiants.

Les enseignants ne sont pas là pour corriger les bugs, pour programmer ou concevoir le programme à la place des étudiants. Si les enseignants l'estiment nécessaire, ils peuvent débloquer les groupes en difficulté. Les questions posées par les étudiants doivent être précises et réfléchies.

1. http://ensiwiki.ensimag.fr/index.php/Projet_C

1.2.2 Cas de fraudes

Il est interdit de copier ou de s'inspirer, même partiellement, de fichiers concernant le projet C, en dehors des fichiers donnés explicitement par les enseignants et des fichiers écrits par des membres de son trinôme. Il est aussi interdit de communiquer des fichiers du projet C à d'autres étudiants que des membres de son trinôme. Les sanctions encourues par les étudiants pris en flagrant délit de fraude sont le zéro au projet (sans possibilité de rattrapage en deuxième session), plus les sanctions prévues dans le règlement de la scolarité en cas de fraude aux examens. Dans ce cadre, il est en particulier interdit :

— d'échanger (par mail, internet, etc.) des fichiers avec d'autres étudiants que les membre de son trinôme.
— de lire ou copier des fichiers du projet C dans des répertoires n'appartenant pas à un membre de son trinôme.
— de posséder sur son répertoire des fichiers de projets des années précédentes ou appartenant à d'autres trinômes.
— de laisser ses fichiers du projet C accessibles à d'autres étudiants que les membres du trinôme. Cela signifie en particulier que les répertoires contenant des fichiers du projet C doivent être des répertoires privés, avec autorisation en lecture, écriture ou exécution uniquement pour le propriétaire, et que les autres membres du trinôme ne peuvent y accéder que par ssh (échange de clef publique) ou un contrôle de droit avec les ACLs (dans les deux cas, des documentations se trouvent sur la page d'EnsiWiki pour vous aider). Pendant la durée du projet, seul les membres du trinôme doivent pouvoir accéder au compte.
— De récupérer du code sur Internet ou toute autre source (sur ce dernier point, contactez les responsables du projet si vous avez de bonnes raisons de vouloir une exception).

Les fichiers concernés par ces interdictions sont tous les fichiers écrits dans le cadre du projet : fichiers C, fichiers assembleurs, scripts de tests, etc.

Dans le cadre du projet C, la fraude est donc un gros risque pour une faible espérance de gain, car étant donné le mode d'évaluation du projet (voir section 1.3), la note que vous aurez dépend davantage de la compréhension du sujet et de la connaissance de l'implantation que vous manifestez plutôt que de la qualité "brute" de cette implantation. Notez également que des outils automatisés de détection de fraude seront utilisés dans le cadre de ce projet.

1.2.3 Conseils et consignes

Le projet ne consiste pas seulement à programmer pendant 2 semaines. Il est conseillé de réfléchir avant de taper : un temps important passé à la conception du programme réduit le temps de développement/débogage du programme, et permet d'éviter les impasses. Ceci dit, avant de passer à la conception du programme, est d'abord nécessaire de bien comprendre les formats (Vorbis et Ogg) et les étapes du décodage ; pas une petite tâche ! Ensuite il vous faudra comprendre la structure générale du code proposé et les spécifications des modules.

Après ceci vous pourrez vous lancer dans la conception de votre projet : premiers modules à implémenter, choix des structures de données intermédiaires le cas échéant, découpage modulaire de votre code (fonctions, fichiers), etc.

Il faut prévoir une décomposition du développement de manière à pouvoir tester et corriger le programme au fur et à mesure que vous l'écrivez. Sinon, vous risquez d'avoir un programme très difficile à corriger, ou vous risquez de devoir réécrire de grandes portions de code. De manière générale, on code d'abord les cas les plus généraux et/ou les plus simples, avant de coder les cas particuliers et/ou compliqués. La programmation modulaire permet en outre d'avoir un code plus concis et donc plus facile à déboguer. Programmez de manière **défensive** : pour les cas que votre programme ne devrait jamais rencontrer si vous avez programmé correctement, mettez un message compréhensible du type `Erreur interne, fonction bidule` et arrêtez proprement le programme, afin de pouvoir déboguer plus facilement. Placez des traces d'exécutions dans vos programmes de manière à pouvoir suivre le déroulement du programme. Utilisez les macros C pour faire afficher ou supprimer ces traces facilement (cf. cours de C au début du projet).

Vous allez aussi **travailler à trois** (toujours... instructif !) : gérez bien les interfaces entre vos tâches et réfléchissez à comment tester votre propre travail même si le module du trinôme n'est pas encore fini (ceci est valable pour les trois membres du trinôme !).

Enfin, **définissez vos priorités !** Est-ce la peine de s'attaquer à une extension alors qu'un module de

la spécification minimale n'a pas été implanté ? Gardez aussi comme ligne directrice d'avoir le plus tôt possible **un programme qui fonctionne**, même s'il ne gère pas tout. Ensuite, améliorez-le au fur et à mesure.

1.2.4 Styles de codage

Indépendamment de la correction des algorithmes, un code de bonne qualité est aussi un code facile, et agréable à lire. Dans un texte en langue naturelle, le choix des mots justes, la longueur des phrases, l'organisation en chapitres et en paragraphes peuvent rendre la lecture fluide, ou bien au contraire très laborieuse.

Pour du code source, c'est la même chose : le choix des noms de variables, l'organisation du code en fonctions, et la disposition (indentation, longueur des lignes, ...) sont très importants pour rendre un code clair. La plupart des projets logiciels se fixent un certain nombre de règles à suivre pour écrire et présenter le code, et s'y tiennent rigoureusement. Ces règles (Coding Style en anglais) permettent non seulement de se forcer à écrire du code de bonne qualité, mais aussi d'écrire du code *homogène*. Par exemple, si on décide d'indenter le code avec des tabulations, on le décide une bonne fois pour toutes et on s'y tiens, pour éviter d'écrire du code dans un style incohérent comme :

```
if (a == b) {
  printf("a == b\n");
} else
{
        printf ( "a et b sont différents\n");
}
```

Pour le projet C, les règles que nous vous imposons sont celles utilisées par le noyau Linux. Pour vous donner une idée du résultat, vous pouvez regarder un fichier source de noyau au hasard (a priori, sans comprendre le fond). Vous trouverez un lien vers le document complet sur EnsiWiki, lisez-le. Certains chapitres sont plus ou moins spécifiques au noyau Linux, vous pouvez donc vous contenter des Chapitres 1 à 9. Le chapitre 5 sur les `typedef`s et le chapitre 7 sur les `goto`s sont un peu complexe à assimiler *vraiment*, et sujets à discussion. Vous pouvez ignorer ces deux chapitres pour le projet C.

Nous rappelons ici le document dans les grandes lignes :

— Règles de présentation du code (indentation à 8 caractères, pas de lignes de plus de 80 caractères, placements des espaces et des accolades, ...)
— Règles et conseils pour le nommage des fonctions (trouver des noms courts et expressifs à la fois).
— Règles de découpage du code en fonction : faire des fonctions courtes, qui font une chose et qui le font bien.
— Règles d'utilisations des commentaires : en bref, expliquez *pourquoi* votre code est comme il est, et non *comment*. Si le code a besoin de beaucoup de commentaire pour expliquer comment il fonctionne, c'est qu'il est trop complexe et qu'il devrait être simplifié.

Certaines de ces règles (en particulier l'indentation) peuvent être appliquées plus ou moins automatiquement.

Le chapitre 9 vous présente quelques outils pour vous épargner les tâches les plus ingrates : GNU Emacs et la commande `indent` (qui fait en fait un peu plus que ce que son nom semble suggérer).

Pour le projet C, nous vous laissons le choix des outils, mais nous exigeons un code conforme à toutes ces directives.

1.2.5 Outils

Les outils pour développer et bien développer en langage C sont nombreux. Nous en présentons ici quelques-uns, mais vous en trouverez plus sur EnsiWiki (les liens appropriés sont sur la page du projet), et bien sur, un peu partout sur Internet !

— `Emacs` ou `Vim` sont d'excellents éditeurs de texte, qui facilite la vie du développeur en C (et pas seulement) : indentation automatique, coloration syntaxique, navigation de code ...
— `hexdump`, qui permet de *dumper*[2] sur le terminal sous différents formats le contenu d'un fichier. On

2. Néologisme pas si récent, *c.f.* http://www.infoclick.fr/dico/D/dump.html, de l'*homo-informaticus* signifiant « copier le contenu d'une mémoire vers un autre support sans aucune transformation de son contenu ».

pourra apprécier l'option `-C`, qui non contente de rappeler la forme classique des années 80, permet de voir les caractères binaires et leur interprétation ASCII.
Pour les amateurs, `od` permet des affichages du même ordre et est en 2 lettres au lieu de 7.

— `xxd`, en trois lettre cette fois, est une sorte de `hexdump -C` mais qui permet aussi de régénérer un fichier binaire à partir de sa représentation hexadécimale avec l'option `-r`.

— `gdb` le debugger ou son interface graphique `ddd` permettent de tracer l'exécution. Son utilisation est très intéressante, mais il ne faut pas espérer réussir à converger vers un programme correct par approximations successives à l'aide de cet outil, ...

— `valgrind` sera votre compagnon tout au long de ce projet. Il vous permet de vérifier à l'exécution les accès mémoires faits par vos programmes. Ceci permet de détecter des erreurs qui seraient passées inaperçues autrement, ou bien d'avoir un diagnostic pour comprendre pourquoi un programme ne marche pas. Il peut également servir à identifier les fuites mémoires (i.e., vérifier que les zones mémoires allouées sont bien désallouées). Pour l'utiliser :
`valgrind [options] <executable> <paramètres de l'exécutable>`

— Deux programmes peuvent vous servir lors de vos tests, ou pour l'optimisation de votre code. `gprof` est un outil de profiling du code, qui permet d'étudier les performances de chaque morceau de votre code. `gcov` permet de tester la couverture de votre code lors de vos tests. L'utilisation des deux programmes en parallèle permet d'optimiser de manière efficace votre code, en ne vous concentrant que sur les points qui apporteront une réelle amélioration à l'ensemble. Pour savoir comment les utiliser, lisez le manuel.

— Pour vous aider à travailler à plusieurs en même temps, des outils peuvent vous aider. Les *gestionnaires de versions* permettent d'avoir plusieurs personnes travaillant sur le même code en parallèle, et de fusionner automatiquement les changements. Votre formation Git du premier semestre devrait vous servir.

— Finalement, pour tout problème avec les outils logiciels utilisés, ou avec certaines fonctions classiques du C, les outils indispensables restent l'option `--help` des programmes, le manuel (`man <commande>`), et en dernier recours, Google ! [3]

1.2.6 Aspects matériel

Tout le travail de programmation s'effectuera sur des machines avec un système d'exploitation Linux. Tous les outils nécessaire sont installés à l'école.

Vous pouvez naturellement travailler sur votre ordinateur personnel si vous en possédez un, mais attention **le projet final doit fonctionner correctement à l'école, sous Linux, le jour de la soutenance** ! Soyez prudent si vous développez sous Windows ou MacOs, le portage est loin d'être toujours immédiat...

1.3 Evaluation du projet et organisation de la soutenance

L'évaluation de votre projet se fera sur les fichiers du projet et lors d'une soutenance. La note du projet est la note de soutenance, elle intègre une évaluation des fichiers de votre projet. La chronologie est la suivante :

— 1 ou 2 jours avant la fin du projet, vous rendez les tous vos fichiers sur TEIDE (voir ci-dessous),

— 1 jour avant la fin du projet, vous préparez vos soutenances (voir ci-dessous),

— le jour de la fin du projet, vous présentez votre projet en soutenance.

1.3.1 Rendu des fichiers de votre projet

L'archive de votre projet devra être déposée sur Teide. Elle contiendra :

— tous les sources et fichiers entêtes (y compris les en-têtes qui vous sont fournis),

— les fichiers de tests créés (Si vous avez des fichiers trop volumineux pour TEIDE, montrez-les le jour de la soutenance),

— un fichier `README.txt` décrivant le contenu du rendu, son organisation, son utilisation et l'intérêt des tests fournis,

— un fichier Makefile avec à minima un cible (`clean`).

Le code rendu doit :

3. 7g de CO_2 par requête

— respecter les directives de codage imposées (voir aussi l'introduction du polycopié).
— compiler avec la commande make
— et sans warning(s) problématiques (l'option -Werror de GCC peut vous aider)
— marcher sur telesun
— et s'y exécuter correctement (là, c'est valgrind qui vous aidera)

Attention, les projets qui ne compilent pas à l'Ensimag le jour de la soutenance ou qui terminent de manière quasi-systématique par un `Segmentation Fault` auront une note inférieure à 8 ! Un moyen simple d'éviter ce type de problème est de compiler et de déboguer le programme au fur et à mesure.

Réaliser parfaitement ce projet pendant le temps imparti est **difficile** pour un trinôme standard d'étudiants en première année. Il n'est donc pas nécessaire d'avoir tout fait pour avoir une bonne note. Par contre, il est nécessaire que ce qui a été fait, ait été bien fait. Pour cela, il faut bien tester au fur et à mesure du développement, en laissant les aspects les plus avancés du projet pour plus tard.

1.3.2 Soutenance

Les modalités vous seront spécifiées en détail le moment venu. Mais le shéma est le suivant :

— Les soutenances durent 1/2h.
— Pendant les *dix* premières minutes, le trinôme expose brièvement un *bilan* du projet et présente une *démonstration* du fonctionnement du programme. Le bilan doit préciser :
  — l'état du programme vis-à-vis du cahier des charges : ce qui a été réalisé, ce qui n'a pas été réalisé, les bugs non corrigées, etc.
  — les principaux choix de conception du programme : structures de données choisies, architecture du programme, etc.
  — les facilités/difficultés rencontrées, les bonnes et mauvaises idées,
  — l'organisation du projet dans le trinôme (répartition des tâches, synchronisation, etc.).

  La démonstration illustre le fonctionnement du programme sur quelques exemples, afin de montrer son adéquation vis-à-vis des spécifications. Il est conseillé d'utiliser plusieurs exemples courts et pertinents pour illustrer les différents points de la spécification. La démonstration pourra contenir 1 ou 2 exemples plus longs pour montrer "le passage à l'échelle".
— Pendant les 20 minutes suivantes, l'enseignant teste vos programmes et vous interroge sur le projet. Les questions peuvent porter sur tous les aspects du projet, mais plus particulièrement sur des *détails* de votre implémentation, comment vous procéderiez pour terminer les *fonctionnalités manquantes*, et comment vous procéderiez pour ajouter une *nouvelle fonctionnalité*.

Vous aurez au moins une journée pour préparer votre soutenance entre la date de rendu des fichiers et votre créneau de soutenance. Préparez la sérieusement ! Il serait dommage d'avoir fait du bon travail sur le projet, mais perdre des points à cause d'une soutenance mal préparée. Il n'est pas demandé des *transparents*, mais répétez plusieurs fois la soutenance pour vous assurer :

— de faire une présentation de *10 minutes* (plus ou moins 1 minute),
— d'aborder tous les sujets demandés (voir ci-dessus),
— de répartir le temps de parole dans le trinôme,
— de vous exercer aux démonstrations.

Chapitre 2

Préambule

2.1 À propos des formats audio - Vorbis

Il existe une multitude de formats audio, certains étant plus utilisés que d'autres. Des formats ne sont utilisés qu'associé à la vidéo (AC3, DTS, AAC,...), d'autres sont extrêmement répandus pour le stockage de la musique au format numérique. Parmi ceux-ci, on en distingue principalement deux types : les formats sans pertes, comme le WAV, ou le FLAC, et les formats avec pertes, comme le MP3, ou le Vorbis. [1]

Le Vorbis est un format audio libre, ouvert, et sans brevet. Il fournit de meilleures performances en termes de rapport compression/qualité audio que le MP3. Cependant, il reste moins utilisé que son concurrent qui lui n'est pas un format libre. Il fait partie de la suite de codecs [2] libre développée par Xiph.org. Une dernière caractéristique intéressante est son extensibilité : tout est prévu dans le Vorbis pour permettre l'insertion de nouvelles méthodes sans remaniement profond de la norme, ce qui en fait un format très intéressant pour les groupements de recherche. Cette caractéristique évite également un gros effort de redéveloppement en cas de nouvelle méthode : si l'implémentation du codec est bien pensée, l'ajout d'une nouvelle fonctionnalité n'implique pas une réécriture complète, mais simplement l'implémentation de cette nouvelle fonctionnalité.

Le Vorbis n'est qu'un format d'encodage, qui requiert un format d'encapsulation, de la même manière qu'en vidéo, l'AVI ou les formats MATROSKA (mkv) encapsulent des vidéos compressées avec certains formats tels que le DivX ou le H.264 (ou le Theora, codec vidéo libre, ouvert, et sans brevet, appartenant aussi à la suite Xiph.org). Le format d'encapsulation généralement utilisé avec le Vorbis est l'Ogg, d'où l'extension généralement trouvée pour les fichiers utilisant Vorbis (.ogg). Le Vorbis s'encapsule également très bien dans le conteneur RTC, prévu pour le streaming audio.

2.2 Objectif

L'objectif du projet est de développer intégralement un décodeur Vorbis conforme à la norme actuelle du Vorbis. Votre décodeur prendra en entrée un fichier Ogg qui contient un flux Vorbis, et produira en sortie les échantillons représentant le son.

Pour des raisons pratiques, l'encodeur traduira le son décompressé en fichiers WAV (qui représentent le son brut), la cacophonie résultant de l'écoute des sons dans les salles machines de l'Ensimag se révélerait trop éprouvante pour vous comme pour nous.

Cependant, nous vous fournissons également de quoi écouter directement le son produit sur vos machines persos, et les fichiers WAV sont lisibles par n'importe quel lecteur audio. Nous vous fournissons également diverses méthodes pour admirer le résultat de vos décodeurs.

2.3 Principe

Le principe du projet est simple. Nous avons réalisé pour vous l'architecture du décodeur, et donc découpé le projet en modules de taille raisonnable. Ces modules vous sont fournis sous la forme d'objets,

1. On ne parle évidemment ici que des formats numériques.
2. Dispositif capable de compresser et/ou de décompresser un signal numérique : COde-DECode en anglais.

et de fichiers d'en-tête C. Vous pouvez donc les utiliser, mais ne disposez pas du code source. Le but du jeu est de reprogrammer tous les modules, et de remplacer ainsi nos modules par vos modules, jusqu'à obtenir un décodeur que vous aurez intégralement développé.

Ce document est normalement suffisant pour réaliser le projet entièrement. Nous vous avons retranscrit et traduit la norme Vorbis. Le rôle et contenu des différents modules est décrit dans une documentation navigable DOXYGEN, qui vous sera fournie en même temps que les objets. Les personnes intéressées par un complément d'informations peuvent consulter la bibliographie.

Bon courage à vous et bienvenue dans le monde du Vorbis !

Bibliographie

[1] Jack E. Bresenham, "Algorithm for computer control of a digital plotter", IBM Systems Journal, Vol. 4, No.1, January 1965, pp. 25 30

[2] http://www.xiph.org/vorbis/

[3] Xiph.org Foundation, "Vorbis I specification", February 2010

[4] Ross N. Williams, "A painless guide to CRC error detection algorithms", http://www.ross.net/crc/crcpaper.html

Chapitre 3

Le codec Vorbis

Ce chapitre a pour objectif de présenter les grands principes du format Vorbis, ainsi que l'aspect haut niveau de la chaîne de codage et décodage. L'aspect pratique du décodage, la séquence d'opérations, et le format final du fichier sont présentés au chapitre 5. On appelle abusivement *fichier Vorbis* le flux Vorbis dans ce chapitre, qui est en réalité encapsulé dans un conteneur (le Ogg, généralement, décrit chapitre 4). Le fichier n'est donc pas un fichier Vorbis mais un fichier Ogg.

3.1 Représentation d'un signal sonore

D'un point de vue physique, le son est un phénomène ondulatoire de bousculement des molécules de l'air. Il est perçu chez l'homme via une vibration du tympan, transformée par les cellules ciliées en un signal électrique transmis au cerveau. Les fréquences auditives perceptibles chez l'homme varient en moyenne de 20 Hz à 20 kHz.

Le son peut être représenté sous forme temporelle (on parle alors de signal, même si cette notation est abusive, puisqu'il ne s'agit en fait que d'une représentation) ou sous forme fréquentielle (figure 3.1) ; on parle alors de représentation fréquentielle, ou de spectre. Cette dernière permet de considérer le signal sonore comme une somme de fonctions sinusoïdales. La représentation temporelle et le spectre sont deux représentations d'un même signal. On passe de l'une à l'autre en utilisant une fonction de transformée, la plus classique étant la transformée de Fourier (dans notre cas c'est une transformée en cosinus discrète, qui est une application de la transformée de Fourier dans le cas réel discret).

Le signal sonore temporel est représenté sur un ordinateur par des échantillons, aussi appelés *PCM* (Pulse Coded Modulation), qui sont le résultat de la discrétisation du signal sonore (qui est lui continu). Pour faire simple, les échantillons représentent l'amplitude du signal sonore à intervalle de temps régulier. Cet intervalle permet de définir la fréquence d'échantillonage, qui est classiquement 44.1 kHz, soit 44100 échantillons par seconde. L'amplitude des échantillons est de plus quantifiée.

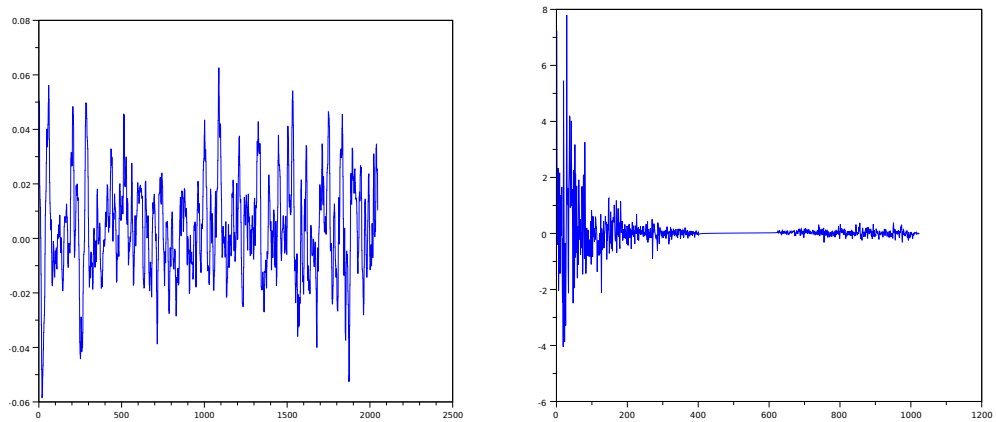


FIGURE 3.1 – Représentations temporelle (à gauche) et frequentielle d'un signal sonore. Avec la transformation MDCT du Vorbis, un signal temporel de N échantillons (PCM) a un spectre de $N/2$ raies.

3.2 Compression audio en Vorbis

Le but de la compression est de réduire la taille des données représentant le son, soit les échantillons PCM en représentation temporelle. L'opération de décompression permet de restituer les échantillons PCM à partir des données compressées. La compression est dite *sans pertes* si le signal initial est retrouvé à l'identique après décompression. Le codec Vorbis est au contraire un format *avec pertes* : le signal décompressé est différent de l'original, même s'il reste proche. La perte d'information doit être acceptable et contrôlée.

La compression repose sur deux principes :

— réduire le nombre d'informations nécessaires. Comme la plupart des applications de traitement du signal la compression audio fait alors appel à la représentation fréquentielle du signal, qui donne de plus grandes opportunités de gain. Le but est de jouer sur les gammes de fréquences audibles pour éliminer une partie de l'information à laquelle l'oreille est la moins sensible.

— travailler sur la représentation des données et leur stockage, pour leur faire prendre moins de place. Des techniques de théorie de l'information sont utilisées, notamment la dégradation de certaines valeurs par quantification et le codage entropique.

Les étapes de la chaîne de compression/décompression sont résumées figure 3.2. Le Vorbis est un format hautement configurable : même si la séquence d'opérations génériques à appliquer pour le codage/décodage est fixée, chaque opération peut utiliser différents algorithmes.

Les sections suivantes décrivent les différentes étapes de la chaîne de compression d'un bloc temporel, puis la section 3.3 décrit justement cette notion de décomposition du signal en blocs.

```
===== COMPRESSION =====

  Pour chaque fenêtre temporelle (ou bloc):
    1. Pour chaque canal
        1.1 Multiplication par la fonction de fenêtrage
        1.2 Changement de domaine (MDCT): signal temporel => spectre
        1.3 Décomposition du spectre en une représentation floor + residues
    2. Couplage des canaux: opérations sur les residues afin de mutualiser
       les informations communes aux différents canaux (par exemple gauche et
       droite)
    3. Quantification des données et codage entropique
    4. Stockage => un paquet Vorbis du conteneur ogg


===== DECOMPRESSION =====

  Pour chaque paquet Vorbis:
    1. Lecture des données (floor + 1 residue par canal)
    2. Découplage des residues
    3. Pour chaque canal
        3.1 Multiplication floor .* residue => spectre
        3.2 Changement de domaine iMDCT: spectre => signal temporel
        3.3 Multiplication par la fonction de fenêtrage
        3.4 Recouvrement: somme avec les PCMs des paquets en recouvrement
```

FIGURE 3.2 – Etapes de la décompression d'un signal.

3.2.1 Changement de domaine : MDCT

La méthode de changement de domaine utilisée dans le Vorbis est une transformée en cosinus discrète (puisqu'on travaille dans le cas discret), qui est en fait une version discrète et réelle de la transformée de Fourier (qui est continue et complexe). La variante de la DCT (Discrete Cosine Transform) utilisée est la

MDCT (Modified DCT) qui a été introduite en 1987. [1] Elle est définie par la fonction suivante :

$$X_k = \frac{4}{N} \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{2N}\left(2n + 1 + \frac{N}{2}\right)(2k+1)\right],$$

avec :

— x le signal sous forme temporelle de taille N (*après* multiplication par la fontion de fenêtrage ; voir section 3.3) ;

— X le signal sous forme fréquentielle, de taille $N/2$.

Elle convertit donc N échantillons temporels (dans le cas du Vorbis, donc, des PCMs) en $N/2$ échantillons représentant le signal dans le domaine fréquentiel.

L'inverse de la MDCT est définie par la fonction suivante (pour le Vorbis) :

$$x_n = \sum_{k=0}^{\frac{N}{2}-1} X_k \cos\left[\frac{\pi}{2N}\left(2n + 1 + \frac{N}{2}\right)(2k+1)\right].$$

L'iMDCT permet donc de repasser des échantillons fréquentiels aux PCMs. Cependant, le signal récupéré après l'inversion de la MDCT n'est pas le même que le signal d'origine. En effet, les plus perspicaces auront noté que le passage dans le domaine fréquentiel avec la MDCT génère une *perte d'information* : on obtient $N/2$ échantillons du spectre à partir de N échantillons temporels . En réalité le signal est compressé par blocs, par fenêtres, et cette perte d'information est gérée en envoyant plusieurs fois l'information via le recouvrement des fenêtres. La section 3.3 est dédié à l'explication de ce mécanisme.

3.2.2 Représentation du spectre : *floor* + *residues*

La représentation du spectre de l'onde sonore obtenu par MDCT se base sur une double représentation : une représentation à gros grain, appelée *floor*, qui définit l'aspect général de la courbe, et une représentation précise à partir de la représentation gros grain, appelée *residue*, qui définit les détails de la courbe. Pour obtenir une valeur du spectre, il faut multiplier la valeur correspondante du *floor* par celle du *residue*. L'oreille humaine est moins sensible aux détails, l'utilisation d'une telle séparation autorise plus de pertes sur les détails, et donc une meilleure compression.

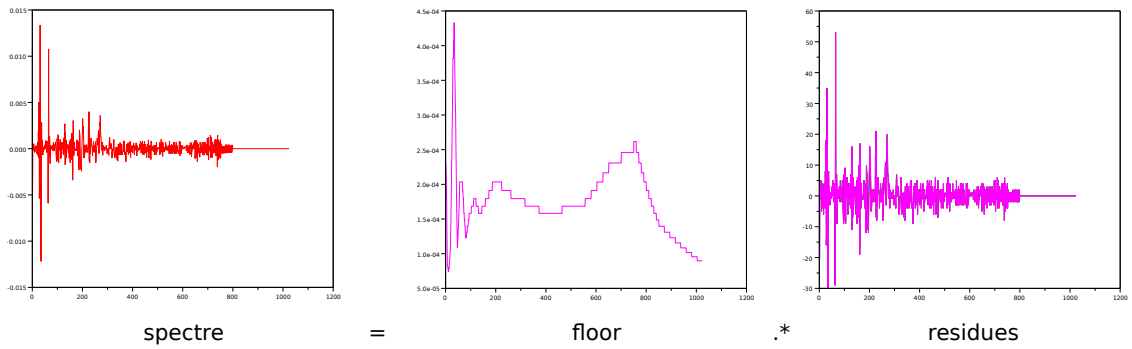


FIGURE 3.3 – Décomposition du spectre comme le produit terme à terme d'une courbe de tendance à gros grains (*floor*) par une courbe de raffinement (*residue*).

3.2.2.a *Floors*

Concernant les *floors*, deux types de représentation sont proposés. Les deux types sont interchangeables, c'est-à-dire qu'on peut passer de la représentation de type 0 à la représentation de type 1 avec une fonction mathématique connue. Dans les deux cas, la récupération de l'aspect général du spectre se fait à partir d'une méthode dite Linear Predictive Coding (LPC). On ne donne ici qu'un aperçu de cette méthode. Le

1. le but recherché de la MDCT est de s'affranchir de problèmes fréquents dans le traitement audio tels que le crênelage (aliasing), que l'on ne traitera pas ici

LPC s'appuie sur un modèle de génération du son à base d'une source et d'un filtre. Les effets du filtre sont appelés les *residues*. Lorsqu'on supprime les effets du filtre sur la source, on obtient la forme générale du spectre, qui correspond aux *floors* du Vorbis. Les *floors* ne peuvent pas être compressés avec pertes, puisqu'ils représentent l'aspect global du signal.

Les représentations type 0 et type 1 sont en fait une représentation différente des coefficients issus du LPC. Le type 0 utilise une représentation LSP (Line Spectral Pair). Cette représentation n'est plus utilisée (mais doit tout de même être supportée dans un décodeur de référence). Elle se base sur des transformées en Z, et est particulièrement complexe à décoder. Nous ne vous demandons pas d'implémenter ce type, sauf si cela vous intéresse, ou si vous avancez particulièrement vite dans le développement du décodeur. Ce type de *floor* n'est pas décrit dans ce document.

Le type 1 utilise une interpolation linéaire par segments pour représenter la courbe. Cette représentation est beaucoup moins lourde pour le décodeur, et est actuellement la plus utilisée. Concrètement, la représentation se base sur une reconstruction étape par étape de la courbe. On commence par générer une courbe d'aspect grossier (en fait, une ligne) entre les bornes de la courbe, puis on précise certains points de la courbe un par un. Plus précisément, voici les étapes de l'induction qui permet de récréer la courbe :

1. initialiser l'itération en générant une ligne droite entre le premier et le dernier point ;
2. sélectionner une abscisse (la sélection est faite à partir du flux, et non pas arbitrairement), calculer la nouvelle ordonnée à partir de l'ordonnée actuelle et d'une différence (récupérée dans le flux) ;
3. à partir de l'abscisse et de la nouvelle ordonnée, on peut reconstruire deux nouveaux segments de droite ;
4. recommencer à l'étape 2 tant qu'il reste des points d'interpolation.

L'exemple donné figure 3.4 est extrait de la norme Vorbis. La courbe des *floors* est représentée par une liste d'abscisses : $0, 128, 64, 32, 96, 16, 48, 80, 112$, et la liste d'ordonnées correspondantes : $110, 20, -5, -45, 0, -25, -10, 30, -10$. On applique l'algorithme en initiant l'induction, on trace donc la première ligne à partir des points $(0, 110)$ et $(128, 20)$. L'ordonnée du point suivant, d'abscisse 64, est codée comme une différence par rapport à la courbe résultant de l'itération précédente. L'ordonnée correspondant à $x = 64$ était $^{110+20}/_{2} = 65$, elle vaut maintenant 60, et on a donc deux lignes. L'itération suivante transforme le point $(32, 85)$ en $(32, 40)$, et ainsi de suite jusqu'à terminer la liste d'abscisses, ce qui produit la courbe finale (voir figure 3.4). L'algorithme correspondant est présenté dans le chapitre 5.

Il est important de noter qu'un « modèle » de *floor* est généralement commun à plusieurs blocs. En particulier les abcisses des points restent identiques pour une même taille de bloc. L'ensemble (limité) de ces configurations, contenant les abcisses, est donc stocké au préalable dans l'entête du fichier. Dans chaque bloc, seule une listes des ordonnées sera stockée ainsi qu'un identifiant de la configuration associée.

3.2.2.b *Residues*

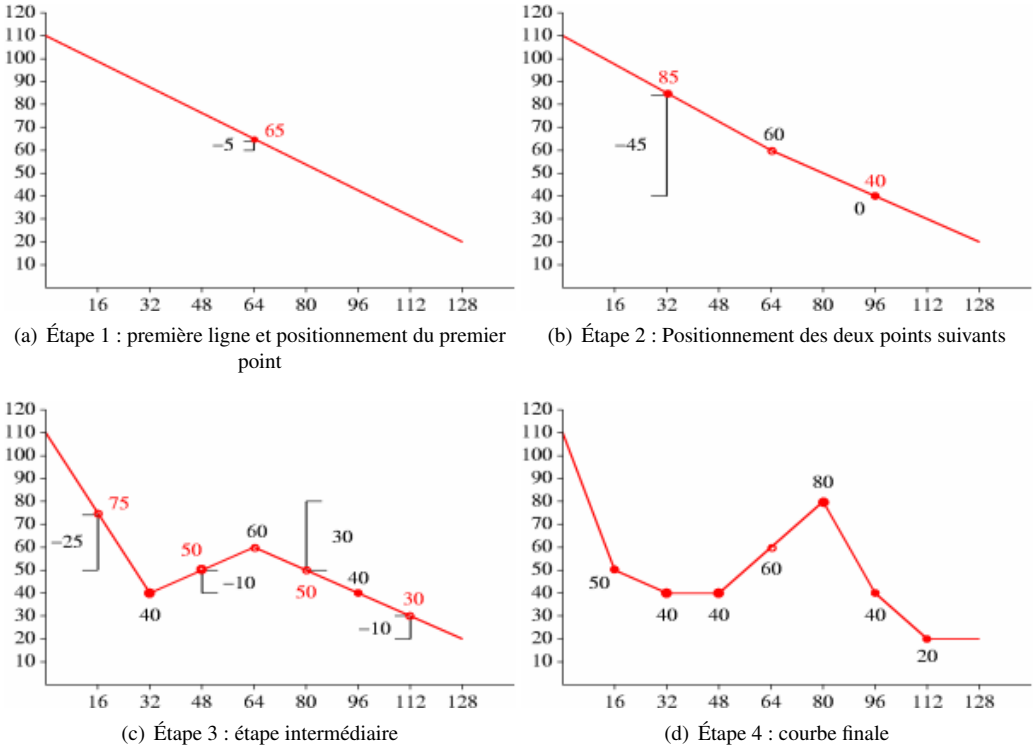
Les *residues* sont, dans le codage LPC, les résultats du filtre appliqué à la source. Ils représentent en fait les variations minimes du signal.

Le vecteur de *residues* est donc un vecteur d'une taille égale à la taille du domaine fréquentiel courant. En pratique, il n'est pas toujours intéréssant de retenir la totalité de ces *residues*. Ainsi le début (basse fréquence) et/ou la fin (haute fréquences) de ce vecteur peuvent parfois être écartées. Le codage des *residues* est donc caractérisé par le début et la fin effective du vecteur retenu. Le vecteur effectif des *residues* illustrés sur la figure 3.5 se termine avant la fin, puisque l'ensemble des dernières valeurs est nul.

Le codage des coefficients de *residues* est ensuite effectué en tenant compte de leur dynamique. Pour celà, le vecteur de *residues* est divisé en partitions. La taille des différentes partitions peut bien sûr être différente. La première zone des *residues* de la figure 3.5 a une grande amplitude, alors que la seconde une amplitude faible. On encode donc ce vecteur sous la forme de deux partitions (A et B sur la figure).

Les valeurs de ces partitions sont ensuite quantifiées en amplitude. La quantification, action de passer d'un domaine continu ou d'une précision très importante vers un domaine de précision plus restreinte, induit une erreur. Afin de minimiser cette erreur, nous profitons de ces dynamiques différentes pour encoder les valeurs de manières différentes. On utilisera un pas de quantification plus important sur la première partition (`quantif A` sur la figure) et un plus fin sur la seconde (`quantif B`).

Cependant, encoder toutes les valeurs quantifiées resterait encore trop coûteux en terme de taille d'informations. Les valeurs quantifiées de ces partitions sont alors encodées à leur tour par groupe. En effet,



(a) Étape 1 : première ligne et positionnement du premier point

(b) Étape 2 : Positionnement des deux points suivants

(c) Étape 3 : étape intermédiaire

(d) Étape 4 : courbe finale

FIGURE 3.4 – Étapes de la création de la courbe des *floors* (figure tirée de la norme Vorbis)

on utilise des vecteur de quantification (*VQ*). Les vecteurs de quantifications ne sont pas codés directement dans le flux ; on utilise le codage entropique pour encoder un numéro de vecteur, une famille d'encodage entropique pour chaque partition. Tous les vecteurs de quantification d'une même famille ont la même taille. Ainsi `A1`, `A2`, `A3` et `A4` ont la même taille (nombre de coefficient de *residue*). Par contre, les familles utilisées pour différentes partitions n'ont pas forcément la même taille. `A1` et `B1` ne sont donc pas forcément de même taille. Le vecteur est quant à lui fixé pour l'ensemble du flux (forme de configuration). L'idée est alors de réutiliser les vecteurs à plusieurs moment du flux afin de diminuer les informations du flux encodé.

Afin d'optimiser la réutilisation de ces vecteurs, il est préférable de créer des vecteurs plus « génériques » et de les mixer pour atteindre la valeur de *residue* souhaitée. Ainsi, le calcul de ces *residues* se fait en 8 passes, où on utilise des *VQ* différents pour chaque passe de chaque partition ; on appelle celà la classification. Le pas de quantification est alors plus fin et on accumule les valeurs des 8 passes.

Pour finir, un flux contient rarement un seul canal (2 pour un flux stéréo, 6 pour un 5.1). L'encodage des *residues* n'est pas forcément effectué canal par canal mais peut aussi être effectué sur la totalité des canaux. Il existe trois possibilités dans la norme pour associer les *residues* des différents canaux pour l'encodage, ils s'appellent types 0, 1 et 2 (le concepteur du Vorbis n'est pas franchement original). Pour les types 0 et 1 on encode bien canal par canal. Ces deux types fonctionnent de manière similaire, seul change l'ordre des coefficients de *residue* dans une partition. Le type 2 fonctionne quant à lui comme un type 1, mais travaille sur l'ensemble des canaux entrelacés. En clair, les types 0 et 1 travaillent canal par canal, alors que le type 2 fonctionne sur un canal virtuel qui contient l'ensemble des canaux entrelacés. Le fonctionnement exact est décrit dans le chapitre 5, lors des informations spécifiques à la lecture du flux Vorbis.

L'ensemble de l'algorithme de décodage des *residues* est détaillé au chapitre 5.

Le Vorbis utilise également une opération de couplage des canaux qui peut modifier les *residues*. Dans le décodeur, le décodage des *residues* a lieu avant l'opération de découplage.

3.2.3 Couplage des canaux

Dans un environnement multicanal, chaque canal produit un son différent, qui dépend de l'emplacement : gauche, droite, surround... Cependant, même si le son est différent, le son d'origine est généralement le même, et il existe donc une corrélation entre les différents canaux. Encoder tous les canaux indépen-

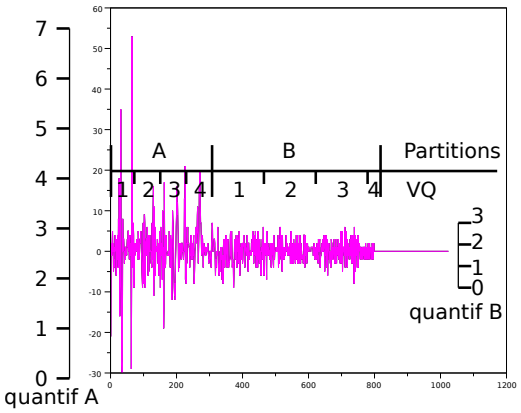


FIGURE 3.5 – Encodage des *residues*

demment génère donc de la redondance. Le couplage (*coupling* en anglais) est une opération qui permet de limiter cette redondance dans le flux encodé. Le format Vorbis permet d'utiliser ou non le couplage canal.

Dans le cas où on utilise du couplage, deux mécanismes sont mis à disposition par le format Vorbis : le *channel interleaving* (entrelacement de canal) et le *square polar mapping* (qui se traduirait par "association cartésien/polaire). On peut utiliser l'une ou l'autre des deux méthodes, ou les deux de manières conjointes. Ces deux mécanismes permettent d'obtenir divers modèles de couplage, du modèle sans pertes avec une équivalence parfaite entre la version couplée et la version non couplée, au modèle avec pertes, qui vise à réduire encore le débit binaire, en supprimant des éléments hors du domaine de l'audible, ou qui n'influent que légèrement sur le rendu du son.

On ne présente que sommairement la théorie associée aux deux méthodes. Elle est basée sur une représentation de l'espace audio à n dimensions, avec n le nombre de canaux. Le signal sonore à l'instant t est le point de coordonnées le vecteur d'échantillons fréquentiels : $s(f) = (s_{c1}(f), s_{c2}(f), \ldots, s_{cn}(f))$. Le *square polar mapping* permet de passer de la représentation cartésienne à une représentation polaire, dans laquelle l'information est représentée de manière asymétrique (le module contient beaucoup plus d'informations que l'argument). L'entrelacement de canal permet de jouer sur les méthodes de quantification à base de *codebooks*, présentées en 3.2.4, en entrelaçant les canaux dans le flux final. L'utilisation conjointe de ces différents mécanismes permet de dégager différents modèles. On notera que les *residues* pour les différents canaux présentés précédement ne sont pas les *residues* du son original, mais ceux issu de l'étape courante, *i.e.,* les canaux couplés. Pour les personnes intéressées, toutes les informations utiles sont sur le site de référence du Vorbis, plus précisément à l'adresse `http://www.xiph.org/vorbis/doc/stereo.html` .

L'opération de découplage est donc un mélange entre du désentrelacement (remise en forme des canaux à partir du flux entrelacé), et des rotations pour repasser le signal sonore en coordonnées cartésiennes. Elle s'applique sur les *residues* uniquement.

3.2.4 *Codebook* : représentation entropique des informations

Les *codebooks* du format Vorbis sont utilisés pour représenter tous types d'informations à stocker dans le flux sous forme entropique. Il peut s'agir d'entiers scalaires, comme les valeurs en ordonnée des *floors*, ou encore les vecteurs de quantification des *residues*.

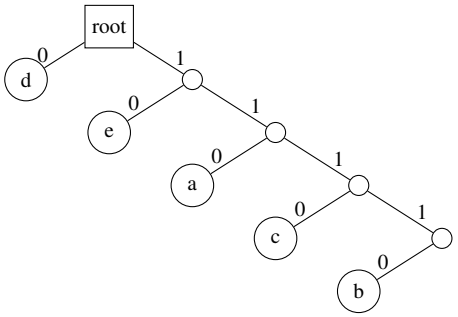
Le codage entropique est une opération qui consiste à coder un dictionnaire de mots de taille fixe à l'aide de codes de taille variable, en privilégiant les codes de petite taille pour les mots fréquents. Le codage utilisé dans le Vorbis est le codage de Huffman, qui est un codage préfixé.

Le principe du codage entropique est le suivant : quand on travaille avec un alphabet de taille finie, il est possible d'étudier la fréquence d'apparition de certains symboles. Par exemple, avec l'alphabet latin, le *'e'* est la lettre la plus utilisée dans la langue française, alors que le *'w'* est la lettre la moins utilisée (si on ne considère pas les accents). On associe alors aux symboles les plus fréquents un code de taille courte, et aux mots les moins fréquents les codes de grande taille. Le code de Huffman est préfixe car par construction, aucun code n'est le préfixe d'un autre code, ce qui évite d'avoir besoin de séparateurs. Les codes utilisés ici sont binaires.

Les codes de Huffman sont généralement représentés comme des arbres binaires, les symboles se trou-

vant sur les feuilles de l'arbre et chaque nœud représentant un bit du mot de code ('0' pour un nœud gauche et '1' pour un nœud droit). Prenons par exemple le code suivant :

Symbole	Code
a	110
b	11110
c	1110
d	0
e	10



Le décodage du bitstream `0101110110010` produit la suite de symboles « `decade` ». Il aurait fallu 3 bits par symbole pour distinguer 5 symboles pour un code de taille fixe (tous les codes sont alors équivalents), et donc la suite de 6 symboles aurait requis 18 bits, alors que 13 seulement sont nécessaires ici.

La quantification est une autre technique de compression, qui consiste à réduire la précision sur les coefficients pour réduire leur taille. Par exemple, des coefficients codés sur 16 bits peuvent être quantifiés en les codant sur 15 bits. Dans le cas du Vorbis, on utilise pour quantifier des vecteurs de quantifications. Ils sont définis dans l'en-tête du fichier et sont utilisés pour coder les *residues*. Ils sont associés aux arbres de Huffman : à chaque feuille de l'arbre est associé un vecteur de quantification. Les *codebooks* pourront donc être utilisés, à chaque fois, de deux manières : soit en scalaire, auquel cas on retourne l'entier associé, soit en « *VQ* », auquel cas on s'intéressera au vecteur de quantification.

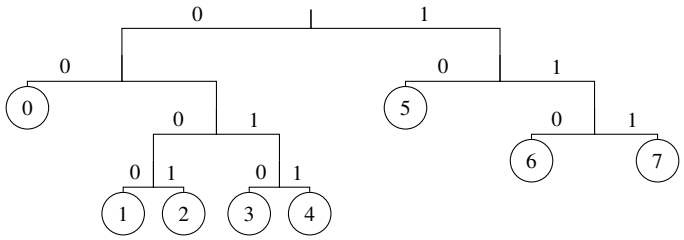
La représentation des dictionnaires dans le Vorbis implique donc une représentation de l'arbre de Huffman et une des vecteurs de quantification (VQ). Celle des VQ est fournie dans le chapitre 5.

Les codes de Huffman sont des codes préfixés. Une fois qu'un mot de code a été utilisé, il ne peut plus servir comme début à un autre mot. Si on fixe comme règle que les codes sont assignés avec d'abord le 0 (arbre gauche) puis le 1 (arbre droit), et que l'on donne les symboles et leur longueur dans l'ordre où ils sont rangés dans l'arbre on peut reconstruire l'arbre assez facilement. Prenons l'exemple suivant, tiré de la norme Vorbis.

Symbole	0	1	2	3	4	5	6	7
Longueur	2	4	4	4	4	2	3	3

La construction des mots de code associés s'effectue symbole par symbole. Le premier symbole est le 0, et a un code de longueur 2. Comme on assigne toujours le bit 0 d'abord, le mot de code associé à 0 est `00`. Puis l'entrée 1 a un mot de code de longueur 4. On ne peut plus commencer par `00` (sinon cela créerait un préfixe), le mot commence donc par `01`, puis on assigne les 0 d'abord. On obtient donc `0100`. En continuant ainsi, on obtient le codage suivant :

Symbole	Longueur	Code
0	2	`00`
1	4	`0100`
2	4	`0101`
3	4	`0110`
4	4	`0111`
5	2	`10`
6	3	`110`
7	3	`111`



Cela revient donc à créer les feuilles d'un arbre binaire où l'on cherche toujours à créer la feuille le plus à gauche possible, à une profondeur égale à la longueur du mot de code. Les symboles Vorbis sont toujours dans l'ordre, la seule information à récupérer dans le fichier est donc les longueurs associées aux symboles. Les algorithmes pour les récupérer sont donnés en section 5.4.1.

3.3 Codage par blocs

Un signal audio réel, par exemple un morceau de musique, n'est jamais compressé en entier, d'un seul tenant, selon la chaîne présentée en 3.2. Tout d'abord, la quantité de données serait bien trop importante

pour toutes les opérations de base (MDCT notamment), les rendant impraticables. Ensuite, un fichier devrait être décompressé en totalité, avec un temps de calcul potentiellement très important selon sa taille, avant de pouvoir commencer la lecture du morceau. Enfin, les fichiers numériques sont très souvent utilisés comme des flux, par exemple pour des applications en *streaming*. Il faudrait donc attendre que la totalité des informations aient été transmises puis décodées avant de lire le son, ce qui n'est pas possible dans ce contexte.

En pratique un signal est donc traité par *blocs*, c'est-à-dire comme une séquence de données temporelles de taille fixe (en nombre d'échantillons PCM). Les opérations de codage/décodage vues en 3.2 sont en fait appliquées à l'échelle d'un bloc. Un bloc peut être traité (codé ou décodé, donc lu) dès lors que toutes ses informations sont disponibles (qu'il a été transmis).

Un bloc, ou une *fenêtre*, est donc une *portion du signal temporel*. Vous remarquerez une similitude entre la notion de fenêtres et la notion de *paquets* Vorbis (section 4.1). La fenêtre est une portion de signal sous forme temporelle, alors que le paquet est la portion de flux Vorbis encodant les informations d'une fenêtre.

3.3.1 Difficultés du codage par blocs

Une solution simple serait de multiplier le signal temporel par une porte rectangulaire de taille N pour extraire un bloc de N échantillons PCM. Par translation de cette porte de $t = 0$ à $t = kN$, le signal serait découpé en un ensemble de blocs successifs indépendants.

Une première limite est que la multiplication par une porte introduit des composantes à hautes fréquences dans le spectre du signal.[2] Après décompression, ces perturbations du spectre font que le signal initial n'est pas retrouvé ; en particulier les frontières des blocs ne correspondent plus forcément. Pour une image, par exemple *jpeg*, ceci reviendrait à une « pixelisation » où des blocs carrés apparaissent. L'oreille humaine étant par contre beaucoup plus sensible que l'œil, ces parasites sonores ne sont pas acceptables.

Une seconde limite est liée à la transformée de changement de domaine utilisée par le Vorbis. Comme vu section 3.2.1, la MDCT génère un spectre de $N/2$ raies à partir de N échantillons temporels. Une partie de l'information est donc perdue.

3.3.2 Compression/décompression par recouvrement de blocs

Le signal est donc découpé en blocs qui se recouvrent deux à deux. Chaque bloc partage donc une moitié des données avec le bloc précédent et une moitié avec le suivant.

La figure 3.6 illustre la compression du signal. Les blocs de N échantillons génèrent par MDCT des spectres de $N/2$ raies qui seront chacun compressé et encodé dans un paquet du flux Vorbis. Le premier bloc est spécifique, équivalent à une fenêtre temporelle centrée en 0 où les $N/2$ premiers PCMs sont nuls. Il sert à l'initialisation du flux.

La figure 3.7 illustre la décompression des paquets. Chaque paquet fournit $N/2$ raies, qui donneront N échantillons temporels après passage au domaine temporel par iMDCT. En raison de la perte d'informations par les transformées, les échantillons reconstruits depuis un seul paquet ne correspondent pas au signal d'origine sur la même fenêtre temporelle. Ils doivent être additionnés (on parle aussi d'accumulation) avec les échantillons reconstruits depuis les paquets en recouvrement. Dans le flux, le processus de décodage est le suivant (avec une seule taille de fenêtre N, pour l'exemple) :

— le décodage du premier paquet fournit N échantillons temporels mais aucun n'est encore complet. La première moitié, nulle, ne sera pas utilisée. Il s'agit en fait de l'initialisation du flux de décodage, aucun PCM n'est produit.

— le décodage d'un paquet fournit ensuite à nouveau N échantillons. La somme des $N/2$ premiers échantillons d'un paquet avec les $N/2$ derniers du paquet précédent fournit bien des PCMs complets. Au décodage d'un nouveau paquet, $N/2$ échantillons PCM sont donc produits.

— le décodage du dernier paquet ne sert qu'à compléter les échantillons incomplets du paquet précédent. Le nombre de PMCs produits par le dernier paquet peut éventuellement être tronqué à une valeur $\leq N/2$, selon la longueur du signal. Les données restantes sont jetées.

3.3.3 Fonction de fenêtrage

Le recouvrement fait donc que toute l'information temporelle est envoyée deux fois. Cependant, si on envoie le signal tel quel deux fois, l'information en sortie du décodeur ne sera plus exacte.

2. Multiplier un signal temporel par une porte revient à convoluer son spectre par un sinus cardinal.

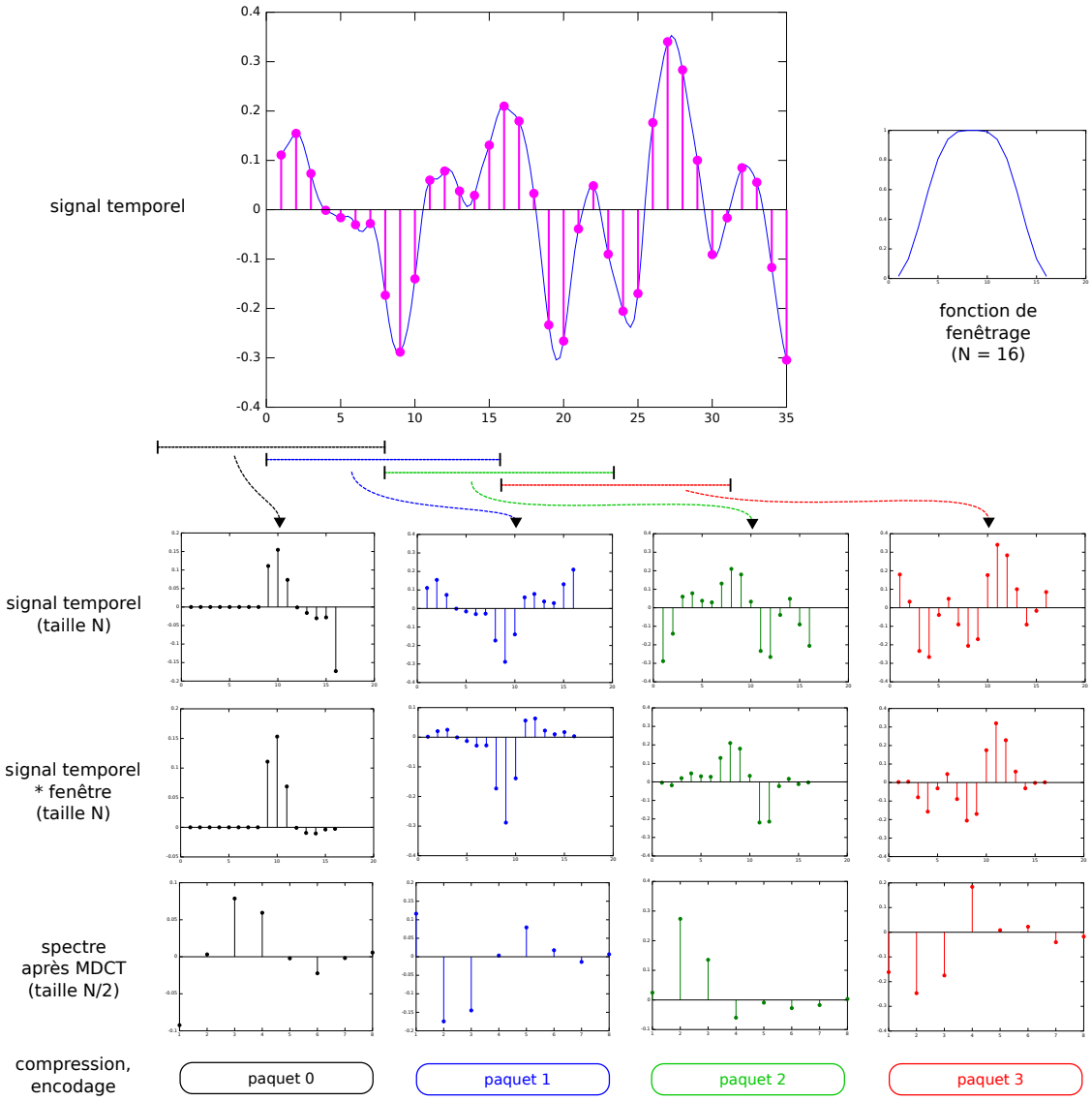


FIGURE 3.6 – Illustration du codage par blocs, ici sur un signal mono-canal avec une unique fenêtre de taille $N = 16$. Les blocs de N échantillons PCMs sont extraits, avec recouvrement, puis multipliés par la fonction de fenêtrage. Le passage au domaine fréquentiel par MDCT réduit l'information à $N/2$ raies par bloc. Après compression, chaque bloc est finalement encodé dans un paquet du flux Vorbis.

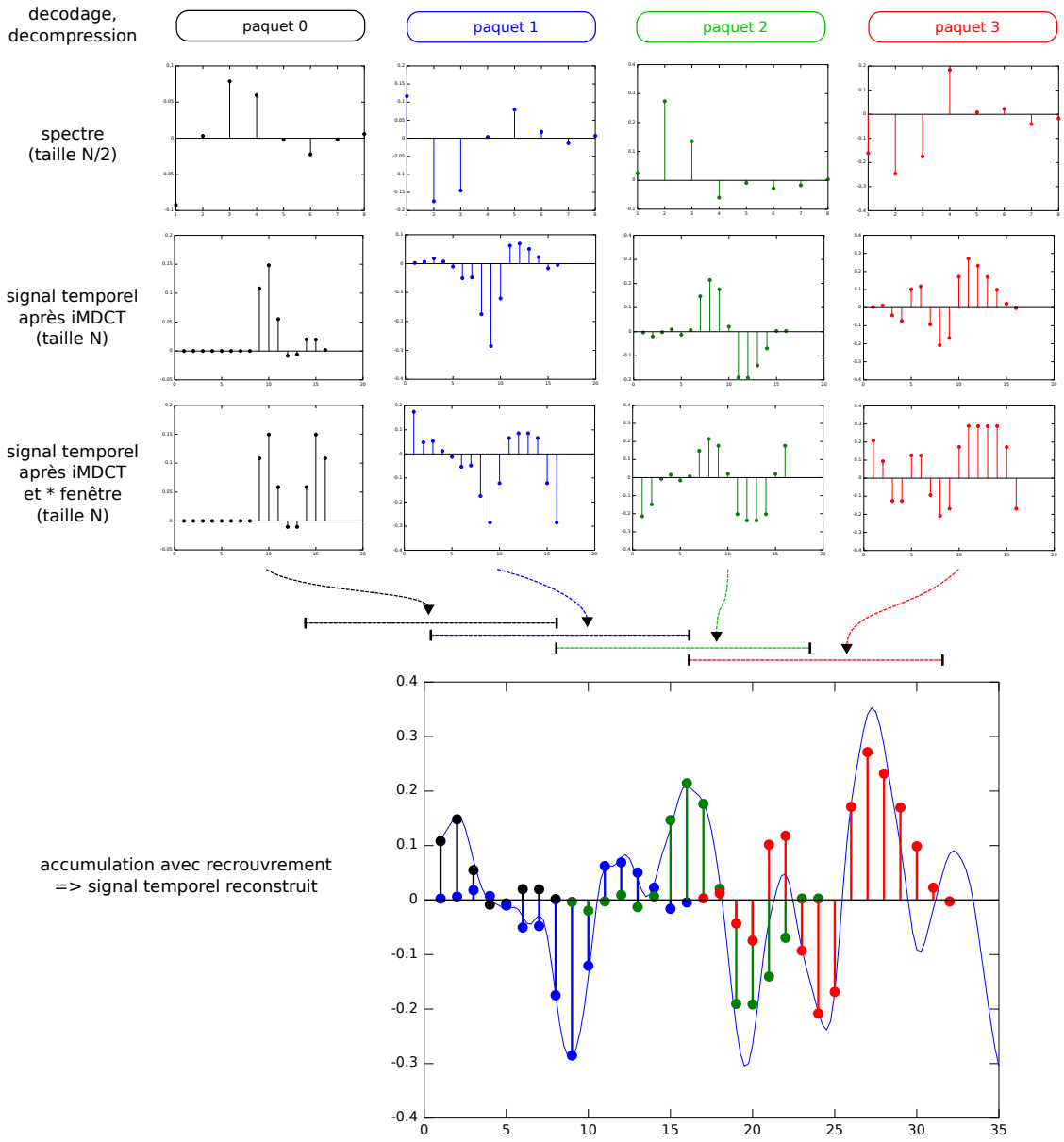


FIGURE 3.7 – Illustration du décodage par blocs, avec accumulation sur les fenêtres temporelles en recouvrement. Le décodage d'un paquet fournit $N/2$ raies. Après passage au domaine temporel, puis multiplication par la fonction de fenêtrage, N échantillons PCMs *incomplets* sont reconstuits. Les échantillons PCMs finaux sont obtenus en sommant les PCMs reconstuits depuis deux blocs en recouvrement. On voit bien sur le dernier bloc (en rouge) que les $N/2$ derniers PCMs reconstruits ne correspondent pas encore au signal. Il faudra leur ajouter les $N/2$ premiers échantillons reconstuits à partir du prochain bloc à décoder. Sur cet exemple la compression est sans perte, donc le signal initial de la figure 3.6 est retrouvé à l'identique. (*NDLR : ce shéma est à regarder sur la version électronique .pdf du sujet, en gros et en couleur !*)

On définit donc une *fonction de fenêtrage* pour pallier ce problème. Cette fonction, de même taille qu'une fenêtre temporelle, est multipliée coefficient à coefficient avec la représentation temporelle du signal, et c'est le résultat qui est transformée par la MDCT (figure 3.6). Les fonctions de fenêtrage du Vorbis sont présentées figure 3.8. On remarque le recouvrement des fenêtres (la fenêtre en pointillés et la fenêtre en trait plein se recoupent).

i) Fonction de fenêtrage La fonction de fenêtrage du Vorbis est définie mathématiquement comme suit, avec N la taille de la fenêtre dans le domaine temporel :

$$\forall n \in [0; N-1], w_n = \sin\left(\frac{\pi}{2} \times \sin^2\left(\frac{n+\frac{1}{2}}{N} \times \pi\right)\right)$$

La fonction de fenêtrage doit vérifier la relation : $w_n^2 + w_{n+\frac{N}{2}}^2 = 1$. En effet, la fenêtre est appliquée deux fois aux échantillons, une fois lors de l'encodage, et une fois lors du décodage, et les échantillons sont additionnés pour le recouvrement. On va donc, pour encoder puis décoder un échantillon, appliquer la fenêtre 4 fois à un seul échantillon :

— une fois à l'encodage du premier bloc : $s_n \times w_{n+\frac{N}{2}}$
— une fois à l'encodage du second bloc : $s_n \times w_n$
— une fois au décodage du premier bloc : $s_n \times w_{n+\frac{N}{2}}^2$
— une fois au décodage du second bloc : $s_n \times w_n^2$

Lors de la correction du recouvrement, on obtient alors le signal reçu

$$r_n = s_n \times w_{n+\frac{N}{2}}^2 + s_n \times w_n^2 = s_n \times (w_n^2 + w_{n+\frac{N}{2}}^2) = s_n$$

ii) Fenêtres de tailles différentes Un inconvénient de la fonction de fenêtrage est qu'elle induit un lissage lors du passage au domaine fréquentiel. Ceci est particulièrement gênant en cas de variabilité hétérogène des composantes fréquentielles (dans le temps). C'est le cas par exemple lors de silences, ou au démarrage/arrêt d'un instrument au sein d'un morceau. Une fenêtre de petite taille permet de bien gérer la dynamicité des données, mais offre une mauvaise compression. A l'inversion, une fenêtre de grande taille amène à une compression plus efficace mais avec un lissage des données plus important.

En pratique, le Vorbis autorise donc deux tailles de fenêtre différentes par flux (la grande et la petite fenêtre). La petite fenêtre est utilisée lors de variations rapide du spectre, la grande le reste du temps.

Comme l'intersection de ces deux fenêtres doit être adaptée, on calcule une fenêtre avec l'algorithme suivant. On note N_i la taille de la fenêtre i. Le mécanisme d'adaptation fait correspondre le point d'abscisse $\frac{3N_{i-1}}{4}$ avec le point d'abscisse $\frac{N_i}{4}$, et le point d'abscisse $\frac{3N_i}{4}$ avec le point d'abscisse $\frac{N_{i+1}}{4}$. Plus concrètement, l'algorithme calcule les indices de début et de fin de pente autour de ces points de correspondance, génèrant ainsi 5 parties disctinctes. On génère ensuite l'enveloppe w comme suit :

$$w_i = \begin{cases} 0 & & \text{si } 0 \leq i < i_0 \\ \sin\left(\frac{\pi}{2} \times \sin^2\left(\frac{i-i_0+\frac{1}{2}}{N} \times \frac{\pi}{2}\right)\right) & \text{avec } N = \min(N_{i-1}, N_i) & \text{si } i_0 \leq i < i_1 \\ 1 & & \text{si } i_1 \leq i < i_2 \\ \sin\left(\frac{\pi}{2} \times \sin^2\left(\frac{i-i_2+\frac{1}{2}}{N} \times \frac{\pi}{2} + \frac{\pi}{2}\right)\right) & \text{avec } N = \min(N_{i+1}, N_i) & \text{si } i_2 \leq i < i_3 \\ 0 & & \text{si } i_3 \leq i < N_i \end{cases}$$

— i_0 et i_1 délimitent la pente de gauche
— i_2 et i_3 délimitent la pente de droite

i_0, i_1, i_2, et i_3 sont récupérées à partir du flux lors du décodage, l'algorithme correspondant à ce calcul est présenté dans le chapitre 5. Cependant, la méthode de calcul est triviale (*cf.* figure).

iii) Bilan Finalement, l'opération inverse de la MDCT se décompose comme suit :
— application de l'iMDCT à un spectre pour récupérer les signaux temporels ;
— application de la fonction de fenêtrage (multiplication) ;
— addition des signaux se recouvrant, qui forment alors le signal sous forme temporelle finale.

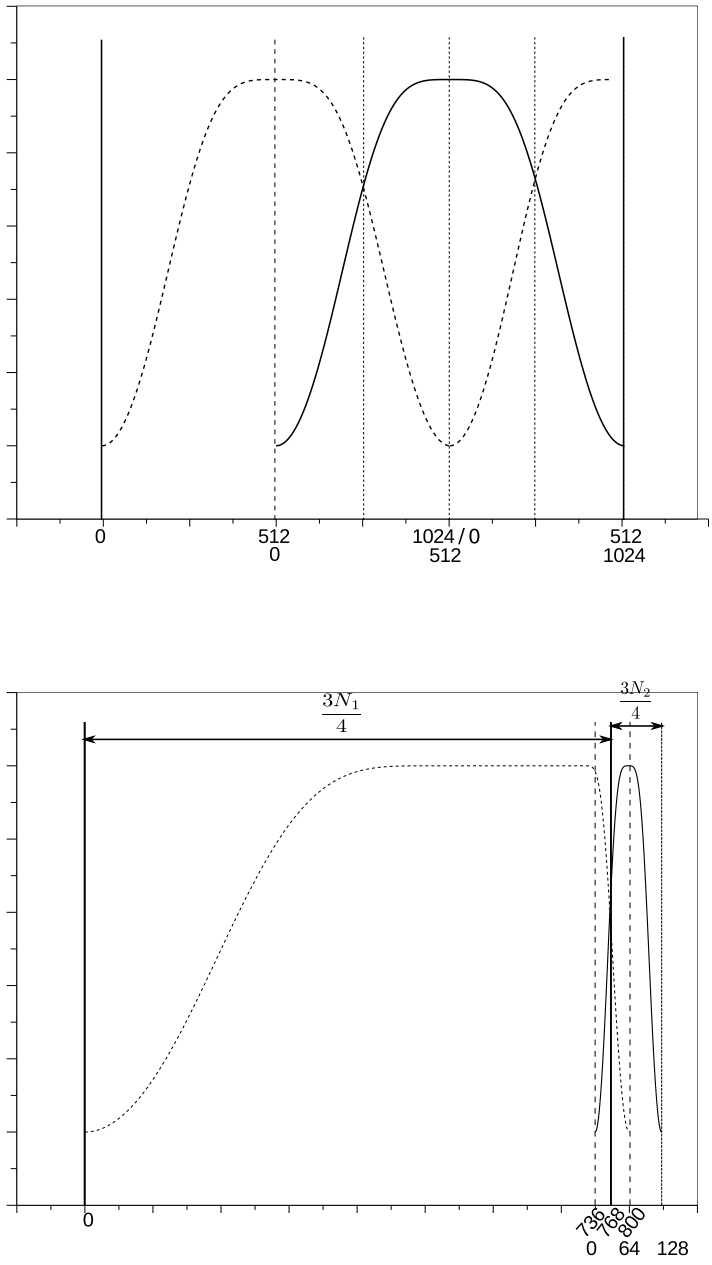


FIGURE 3.8 – Exemple de fonctions de fenêtrage. En haut, recouvrement avec des fenêtres de taille identique $N = 1024$. En bas, cas où les fenêtres ne sont pas de même taille ($N_1 = 1024, N_2 = 128$).

On notera bien que pour une taille de bloc N, les données fréquentielles contiendront $N/2$ coefficients, alors que les données temporelles en contiendront N. Également, le nombre d'échantillons PCMs produits au décodage d'un paquet Vorbis n'est pas toujours identique, il dépend des tailles de fenêtre des deux paquets en recouvrement.

3.4 Configuration du format Vorbis : *mode* et *mapping*

La décompression d'un flux Vorbis est réalisée en deux étapes principales :

— une partie d'initialisation, qui récupère dans le fichier l'ensemble des données dites de *configuration*. Ces données sont lues dans les premiers paquets d'en-têtes du flux Vorbis. Il s'agit en particulier des informations sur les *floor*, les *residues*, les modes de couplage, les fenêtres, ainsi que les dictionnaires (*codebooks*) utilisés pour le morceau.
— une partie de décodage, qui applique les opérations de décompression aux données pour produire les PCMs. Le corps du flux Vorbis est lu et traité paquet par paquet (bloc par bloc). Pour chaque paquet, la configuration du paquet est d'abord lue (son *mode*) puis les données spécifiques permettant de reconstuire les *floor* et *residues* correspondant au paquet.

Comme précisé précédemment, le Vorbis est un format hautement configurable, ce qui permet d'étendre très facilement le format normalisé. En effet, même si la séquence d'opérations à appliquer est définie, plusieurs algorithmes peuvent être définis pour une même opération. Ces différents algorithmes peuvent être utilisés à l'intérieur d'un même flux sur des paquets différents. Certains paramètres sont aussi fixés par paquet, par exemple la taille du paquet courant.

Lors de l'initialisation du décodage (lecture des en-têtes), toutes les ***configurations*** utilisées par un flux Vorbis sont définies. Puis pour chaque paquet on lira la configuration qui lui est associée.

On appelle ici configuration un jeu de valeurs pour l'ensemble des paramètres. Ainsi, si nos paramètres sont la taille du paquet et la représentation des floors, on peut avoir une configuration où la taille vaut 1024 et le floor est de type 0, et une autre configuration où la taille vaut 512 et le floor est de type 1.

Dans le cadre du Vorbis, le mécanisme de configuration est fourni par les *modes* et les *mappings*.
— À chaque paquet est associé un ***mode***, qui spécifie :
  — le type de la fenêtre (petite ou grande) ;
  — la transformée utilisée ;
  — le *mapping* associé à ce paquet.
— Une configuration de ***mapping*** spécifie les informations sur la représentation des données :
  — le type de *floor* à utiliser (0 ou 1) et les dictionnaires (*codebook*) associés pour lire les valeurs ;
  — les informations sur les *residues* (type 0, 1 ou 2) pour chaque canal, et les *codebooks* pour la lecture des données ;
  — les informations de couplage des canaux.

Notez que le Vorbis offre la possibilité de spécifier le type de transformée utilisé, mais cette possibilité n'est pour l'instant jamais exploitée, puisque la norme ne spécifie qu'un type de transformée (MDCT). Il en est de même pour le type de la fenêtre (0 ou 1 seulement).

Durant le décodage, la première information récupérée dans un paquet Vorbis est donc le *mode* utilisé pour ce paquet, ce qui permet de déduire les paramètres et algorithmes à utiliser pour le décoder.

Le schéma 3.9 représente l'ordonnancement des opérations à effectuer sur un paquet. Les flèches en trait plein représentent les données, alors que les flèches en pointillés représentent le paramètrage des blocs. Les opérations se classifient en deux types, certaines permettant de récupérer des données dans le paquet (*mode*, *codebook*, *floor*), alors que d'autres ne font que réaliser des calculs, sans lire le flux. Avant la transformée, les données sont une représentation fréquentielle du signal ; après la transformée, le signal est sous forme temporelle.

Les différentes opérations sont présentées plus en détail dans le chapitre 5.

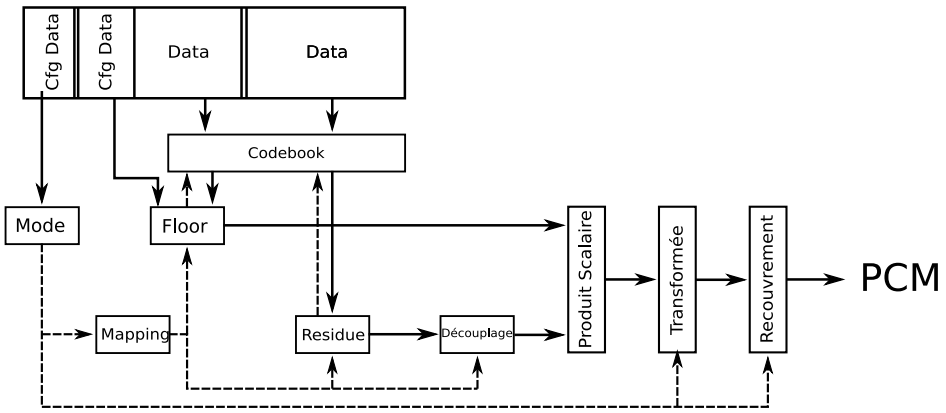


FIGURE 3.9 – Opérations de décompression d'un paquet Vorbis

Chapitre 4

Le conteneur Ogg

Le format *Ogg* est un conteneur multimédia (de même que les formats Avi ou Mkv). Il sert à stocker ou à transmettre ensemble différents médias (par exemple de la vidéo, de l'audio, des images ou du texte). Ogg est un format qui permet d'encapsuler et d'entrelacer différents flux/fichiers médias appelés *flux logiques*.

Un flux/fichier Ogg est constitué de un ou plusieurs *flux physiques* qui sont mis bout à bout. Ces flux physiques sont complètement indépendants. Dans la suite nous ne traiterons donc que d'un seul flux physique.

4.1 Flux logique et paquets

Pour qu'un flux logique puisse être encapsulé dans un flux Ogg il faut qu'il soit organisé en *paquets*. Ce découpage n'est pas fait par Ogg, il est dépendant du format du flux logique. Lors de la lecture d'un flux logique, les paquets sont restitués l'un après l'autre au décodeur approrié. Le format vidéo *Theora*, le format audio *Vorbis*, le format d'image *Png* sont des exemples de flux logiques supportés par le Ogg.

Un flux logique doit commencer par un ou plusieurs paquets d'en-tête qui permettent d'initialiser le décodeur approprié. Ceux-ci ne doivent pas contenir de données. Le premier paquet doit de plus permettre de trouver facilement de quel format est le flux logique. La manière la plus classique pour cela est que le premier paquet commence par une séquence particulière spécifique au format.

4.2 Entrelacement

Un flux physique est lui même constitué de plusieurs flux logiques entrelacés. Au sein d'un flux physique, chaque flux logique est identifié par un numéro différent. Dans Ogg, l'unité d'entrelacement est la *page*. Une page appartient à un unique flux logique et en contient une portion. Au sein d'un flux logique, les pages sont ordonnées et numérotées par ordre croissant.

La première page d'un flux logique est appelée page *BOS* (*Begin Of Stream*) et ne doit contenir que le premier paquet du flux. La dernière est appelée page *EOS* (*End Of Stream*). Les pages dans un flux physique doivent respecter l'ordre suivant :

— ensemble des pages BOS de tous les flux logiques ;
— pages éventuelles contenant les paquets d'en-tête supplémentaires [1] ;
— les pages contenant les paquets de données. [2]

FIGURE 4.1 – Entrelacement des pages dans un flux physique

Il n'y a pas de contraintes sur les pages EOS. Elles n'ont pas à être forcément à la fin du flux physique. Une page BOS peut même être aussi une page EOS si le flux logique ne contient qu'une page.

1. Bien que Ogg impose que les paquets d'en-tête soient mis avant tout paquet de données, il n'y a aucun mécanisme pour différencier ces deux catégories de paquets dans Ogg.
2. Aucun paquet de données ne doit apparaître dans les pages contenant des paquets d'en-tête.

4.3 Segments

Les paquets ne sont pas directement encapsulés dans les pages d'un flux logique. Chaque paquet est d'abord divisé en un ou plusieurs morceaux appelés *segments* comme illustré sur la FIGURE 4.2.

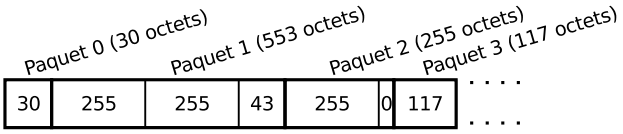


FIGURE 4.2 – Subdivision des paquets d'un flux logique en segments

Un segment fait au maximum 255 octets, ainsi sa taille peut se coder sur un octet. Le premier segment d'un paquet contient ainsi les 255 premiers octets, le deuxième les 255 suivants, etc. Cela jusqu'à atteindre la fin du paquet. Le dernier segment d'un paquet contient obligatoirement moins de 255 octets : si un paquet contient un multiple de 255 octets, alors son dernier segment contiendra 0 octets. Cela permet de repérer les fins de paquets : dès qu'un segment a une taille différente de 255, cela signifie que c'est le dernier du paquet en cours. Les paquets de taille nulle ne sont pas interdits, ils ne seront formés que d'un segment de taille nulle.

Ces segments sont ensuite encapsulés dans les pages du *flux logique* correspondant dans l'ordre du flux : le premier segment du premier paquet, suivi du deuxième segment du premier paquet, etc.

4.4 Pages

Dans un flux physique, les pages se succèdent les unes aux autres. Chacune contient jusqu'à 255 segments. [3] Une page est formée d'un en-tête, qui indique en particulier le nombre et la taille des segments qu'elle contient, suivie des segments mis bout à bout (voir FIGURE 4.3). Il n'y a aucune séparation entre les différents segments puisque la taille de chacun d'entre eux est indiquée dans l'en-tête de la page.

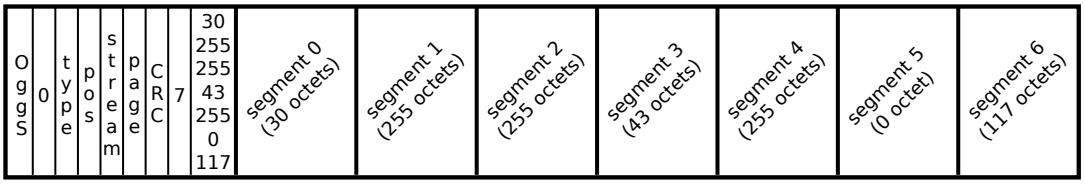


FIGURE 4.3 – Exemple de page

4.4.1 En-tête

L'en-tête de la page contient 8 champs de taille fixe et un champ de taille variable. Les champs utilisent la convention petit-boutiste (*little-endian*) (octet de poids faible d'abord) pour coder les champs de plusieurs octets. Le tableau 4.1 récapitule les différents champs et leur taille.

— ***capture pattern*** (4 octets)
Indique le début d'une page. Ce champ doit contenir les 4 caractères ASCII « OggS ».
— ***stream structure version*** (1 octet)
Indique la version du format du fichier Ogg, la version décrite ici est la 0.
— ***header type flag*** (1 octet)
Renseigne les attributs de la page. Seuls les 3 premiers bits sont utilisés, les 5 autres doivent être mis à 0.
  — Bit 0 : est à 1 si la page continue le paquet non terminé dans la page précédente. [4]
  — Bit 1 : est à 1 si la page est la première du flux logique (page BOS).
  — Bit 2 : est à 1 si la page est la dernière du flux logique (page EOS).

3. Il n'est pas interdit qu'une page ne contienne aucun segment.
4. Ce bit est obligatoire, mais redondant puisque la taille du dernier segment de la page précédente nous indique si le paquet est terminé.

Nom	Taille (en octets)
Capture pattern	4
Stream structure version	1
Header type flag	1
Granule position	8
Bitstream serial number	4
Page sequence number	4
CRC checksum	4
Page segments	1
Segment table	variable

TABLE 4.1 – En-tête d'une page

— ***granule position*** (8 octets)
Informe sur la position absolue dans le flux logique une fois que le dernier paquet complet de la page est décodé (le dernier paquet de la page, s'il n'est pas terminé, n'est pas comptabilisé). Sa valeur dépend du format du flux logique. Néanmoins la valeur -1 (en complément à 2) [5] indique qu'aucun paquet ne se termine sur la page (elle ne contient qu'une portion d'un paquet). Pour le format *Vorbis*, il indique le nombre d'échantillons décodés.
— ***bitstream serial number*** (4 octets)
Numéro identifiant le flux logique. Il doit être unique dans chaque flux physique, mais il est complètement aléatoire.
— ***page sequence number*** (4 octets)
Numéro identifiant la page dans le flux logique. Les pages doivent être numérotées en ordre croissant. Ainsi si un numéro est sauté, cela indique que le flux est corrompu.
— ***CRC checksum*** (4 octets)
Code de contrôle permettant de vérifier que la page n'est pas corrompue. La méthode pour le calculer est exposée dans la partie 4.4.2.
— ***page segments*** (1 octet)
Indique le nombre de segments que contient la page (de 0 à 255).
— ***segment table***
Liste d'octets indiquant la taille de chacun des segments (de 0 à 255). La taille de ce champ est donc variable et est égale à la valeur du champ précédent.

4.4.2 Calcul du CRC

Le Contrôle de Redondance Cyclique (Cyclic Redundancy Check, CRC) est une méthode utilisée pour détecter les erreurs dans un message. Ils permettent de calculer un résumé du message (qu'on appelle *hash*, ou abusivement et plus couramment *checksum* ou tout simplement CRC). Concrètement, il sert à vérifier l'intégrité des données. Des explications sur la théorie peuvent être trouvés dans l'article de Ross Williams [4], ou dans les livres sur la théorie de l'information pour les plus perfectionnistes.

Le calcul des CRC est basé sur la division polynomiale. Dans le cas du Ogg, on utilise un résumé sur 32 bits, avec comme polynôme générateur [6] $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+0$. On note ce polynôme 0x04c11db7 : les bits à 1 correspondent aux coefficients présents, le coefficient le plus haut n'est pas représenté.

Nous vous proposons l'algorithme suivant pour le calcul du CRC de l'Ogg, qui est une méthode simple mais peu performante.

```
polynome := 0x04c11db7 (32 bits)
crc      := 0x00000000 (32 bits)
boucle pour tous les bits 'bit' du message
   si (bit xor bit31(crc) = 1)
      crc := (crc << 1) xor polynome
   sinon
```

5. Toutes les machines sauf les plus exotiques travaillent en complément à 2 pour les nombres signés, vous n'avez donc pas à vous en préoccuper dans le cadre du projet.
6. Le polynôme est en fait celui de la norme CRC-32-IEEE-802.3.

```
      crc := crc << 1
   fin si
fin boucle
return crc
```

Dans notre cas, le message est une page complète dont le champ CRC de l'en-tête aura été mis à 0 (on ne l'a pas encore calculé). Les octets sont traités dans l'ordre (en-tête, puis les segments) ; chaque octet est traité du bit de poids fort au bit de poids faible.

Chapitre 5

Lecture d'un flux Vorbis

L'objectif de cette section est de vous présenter l'emplacement des différentes informations requises pour décoder un flux Vorbis. Il s'agit de la description du *flux logique*, découpé en paquets. Les aspects liés au stockage dans le conteneur Ogg (en pages, segments... voir la section 4) sont totalement encapsulés : en gros le flux est disponible sous forme de paquets Vorbis, chacun étant une suite continue de bits.

Le flux Vorbis est divisé en deux parties :

— un en-tête, ou *header*, qui contient toutes les informations utiles au décodage : les différentes configurations, les codebooks, les méthodes de représentation, etc. Il y a en fait trois en-têtes dans le Vorbis.

— le corps du flux, qui contient les ondes sonores compressées, découpé en paquets représentants les différentes fenêtres temporelles.

5.1 Préambule

Le Vorbis est un fichier binaire, sans alignement. Ceci pose certains problèmes lorsqu'on programme. En effet, les processeurs classiques sont prévus pour travailler sur des octets, ou plus généralement sur des ensembles d'octets (des mots de 32 bits ou 64 bits). Les langages de programmation ne permettent donc pas, ou peu, de travailler sur des éléments binaires qui ne rentrent pas dans les tailles habituelles. De même, pour la lecture du fichier, on ne peut pas utiliser les bibliothèques habituelles, qui lisent les fichiers octets par octets (comme les fichiers de texte). Toute lecture du fichier se fera donc à travers le module d'entrée-sortie spécifiquement développé pour le projet. Le Vorbis est un format little-endian.

Pensez aussi à utiliser les opérateurs binaires du C et les masques :

— & pour le AND

— | pour le OR

— ~ pour le NOT

— ^ pour le XOR

De même, **pensez à annuler les bits non utilisés**, lorsque vous travaillez en binaire. Par exemple, si vous représentez le bit '1' sur un octet, les 7 bits de poids forts ne sont pas utilisés. Si vous effectuez des opérations, il peut arriver qu'ils soient quand même modifiés. Si vous testez ensuite la valeur, pour savoir si elle vaut 1 ou 0, elle ne vaudra ni l'un, ni l'autre ! Pour rappel, un AND avec le bit '0' fait toujours '0', un OR avec le bit '1' fait toujours '1'.

Les machines sur lesquelles vous développez étant pour la plupart également en little-endian (sauf si certains développent sur des machines Sun©, par exemple, ou PowerPC© pour certains Apple©), il n'y aura pas de problème d'endianness à gérer. Par contre, les outils utilisés pour afficher les fichiers binaires font automatiquement la conversion en big endian, pour plus d'aisance de lecture. En effet, tout le monde apprend à compter en big endian, et par habitude, nous sommes nous-même big-endian (on écrit 10 pour dix, et non pas 01). Pensez donc bien, lorsque vous lisez deux bits successivement dans un octet, que les valeurs que vous voyez à l'écran sont à inverser. Par exemple, si vous voyez l'octet $0x80$, et que vous voulez lire 1 bit, le bit en question sera un '0' (le plus à droite sur le visualiseur, mais le plus à gauche dans la machine), et non pas un '1'. Enfin, une dernière recommendation à propos des manipulations de types en C : d'une manière générale, les *cast* vers les types signés étendent le signe : si vous passez un `unsigned char` en `signed short`, et que ce caractère était plus grand que 127, le `short` résultant sera négatif. Ceci s'applique

aux shifts binaires aussi. D'une manière générale, n'utilisez les valeurs signées qu'avec précaution, et quand c'est nécessaire.

Dans la suite du document, les différentes sections du flux Vorbis vous sont présentées sous la forme de pseudo-code. Pensez bien à lire tous les bits précisés, même si l'information n'est pas utilisée.

Dans ces notations de pseudo-code, vous trouverez entre crochets la notation représentant les variables. Cela ne veut cependant pas dire que vous devrez avoir une variable du même nom dans votre code, il peut s'agir d'un champs d'une structure aussi. Par exemple, `[ mapping_submap_residue [i]]` ne veut pas uniquement dire nom de variable `mapping_submap_residue` de type tableau. Il peut tout aussi bien s'agir d'un champ `submap_residue` de type tableau dans la structure de *mapping* courant. Il faudra analyser judicieusement les algorithmes pour faire le bon choix.

Par ailleurs, si dans l'algorithme, il est précisé que le flux est non décodable, vous devez arrêter le décodage. Pour cela, une simple commande `exit` sera suffisante, nous ne vous demandons pas de contrôle d'erreurs avancé. Par contre, vous verrez que le Vorbis résiste bien aux erreurs dans les paquets, puisque la contre-mesure dans ce cas est de finir le traitement avec les données déjà obtenues et passer au paquet suivant.

5.2 Décodage des en-têtes Vorbis

Les trois headers du Vorbis commencent tous par 7 octets ayant la même signification :

```
[packet_type] = lire 1 bit, doit valoir 1 pour les headers
[header_type] = lire 7 bits
[identification] = lire 6 fois 8 bits, doit valoir :
        0x76,0x6f,0x72,0x62,0x69,0x73 (\og vorbis \fg)
```

5.2.1 Header 1 : Identification

Le premier header du Vorbis identifie le flux comme étant un flux Vorbis. Il est identifié par la valeur `header_type` 0. On y trouve toutes les informations globales sur le flux en cours : nombre de canaux, débit, taux d'échantillonage, grande taille et petite taille des fenêtres, etc.

```
[vorbis_version]    = lire 32 bits comme un entier non-signé
[audio_channels]    = lire 8 bits comme un entier non-signé
[audio_sample_rate] = lire 32 bits comme un entier non-signé
[bitrate_maximum]   = lire 32 bits comme un entier signé
[bitrate_nominal]   = lire 32 bits comme un entier signé
[bitrate_minimum]   = lire 32 bits comme un entier signé
[blocksize_0]       = 2 puissance (lire 4 bits comme un entier non-signé)
[blocksize_1]       = 2 puissance (lire 4 bits comme un entier non-signé)
[framing_flag]      = lire 1 bit, doit valoir 1 sinon erreur
```

`[vorbis_version]` doit être égal à 0, sinon le flux n'est pas décodable. `[audio_channels]` et `[audio_sample_rate]` doivent être supérieur à 0. `[blocksize_i]` représentent les tailles de fenêtres, et doivent vérifier $64 \leq$ `[blocksize_0]` $\leq$ `[blocksize_1]` ≤ 8192.

Les valeurs de *bitrate* ne sont que des indications, et ne sont pas utilisées dans notre décodeur.

Enfin le fanion `framing_flag` ne sert qu'à vérifier le bon déroulement de l'opération de lecture de l'entête. Il suffit de vérifier sa valeur (1), et il peut être jeté. Dans tous les cas, si vous avez un `framing_flag` à lire, il ne sert qu'à vérifier le positionnement dans le fichier, et n'est pas utilisé lors du décodage.

5.2.2 Header 2 : Commentaires

Le deuxième header a comme `[header_type]` 1. Il sert à encoder les méta-informations sur le flux Vorbis en cours, comme par exemple le titre de la chanson, la date, la durée ou l'auteur. N'importe quel information peut être incluse, mais certaines informations sont normalisées, pour faciliter le décodage.

Le format de stockage est le suivant :

```
[vendor_length] = lire 32 bits comme un entier non-signé
[vendor_string] = lire [vendor_length] caractères
[user_comment_list_length] = lire 32 bits comme un entier non-signé
pour i de 0 à [user_comment_list_length - 1] {
    [length_i] = lire 32 bits comme un entier non-signé
    [user_comment_i] = lire [length_i] caractères
}
[framing_bit] = lire 1 bit, doit valoir 1 sinon erreur
```

Chacun des commentaires `[user_comment_i]` est de la forme *"FIELD=value"*. Pour rappel, '=' en ASCII se code `0x3D`. La première moitié, le champ (`FIELD`) est insensible à la casse, et indique quelle sera l'information qui suit. Ce champ peut valoir n'importe quoi, mais certaines valeurs sont réservées :

TITLE Titre de la piste ;

VERSION Info sur la version de la piste (remix ou autre) ;

ALBUM Album dont est extraite la piste ;

TRACKNUMBER Le numéro de la piste dans l'album ;

ARTIST L'artiste qui interprète la piste ;

COPYRIGHT Propriétaire des droits ;

LICENSE Information légale ;

ORGANIZATION Organisation qui produit la piste, *i.e.,* le label ;

DESCRIPTION Description de la piste ;

GENRE Genre musical ;

DATE Date d'enregistrement ;

LOCATION Lieu d'enregistrement ;

CONTACT Personne à contacter en cas de besoin ;

ISRC International Standard Recording Code, information de référencement.

Ici encore le `framing_flag` doit être jeté après usage.

5.2.3 Header 3 : Configuration

Le troisième et dernier header contient toutes les informations de configuration requises pour décoder le flux Vorbis. Il est constitué de six sous-parties consécutives, une pour chacun des éléments du Vorbis. Il s'agit, dans l'ordre, de : *codebook*, *transformée*, *floors*, *residue*, *mapping* puis *mode*.

Chacune des sous-parties contient d'abord le nombre de configurations, puis l'ensemble des configurations disponibles. Les configurations sont stockées dans l'ordre de leur identifiant. Les détails de configuration de chaque élément sont décrits dans les sous-sections qui leur sont dédiées (section 5.4).

5.2.3.a Codebook

Contient d'abord le nombre de *codebooks* à décoder, puis chacun des *codebooks* :

```
[vorbis_codebook_count] = lire 8 bits comme un entier non-signé
[vorbis_codebook_count] = [vorbis_codebook_count] + 1
pour [codebook_id] dans 0 .. [vorbis_codebook_count - 1]
    décoder le codebook [codebook_id]
```

Le décodage d'un *codebook* est présenté en section 5.4.1.

5.2.3.b Time Domain Transforms

Cette partie de l'en-tête 3 contient des informations sur la méthode de changement de domaine temps-fréquence. Elles sont ignorées dans la version actuelle de Vorbis, car la MDCT est systématiquement utilisée, mais permettent des extensions possibles dans les futures révisions de la norme. Ces données doivent tout de même être lues pour conserver la synchronisation dans le flux et pouvoir décoder la suite de l'en-tête.

```
[time_count] = lire 6 bits comme un entier non signé et ajouter 1
pour [i] dans 0 .. [time_count - 1] faire
      [val] = lire 16 bit;
      si [val] != 0 alors
         flux non decodable
      fin si
fin pour
```

Aucune sous-section spécifique n'est liée à la transformation dans ce document (rien de plus à décrire). Vous veuillerez cependant à initialiser les structures internes éventuelles dans votre code, le cas échéant.

5.2.3.c Floor

Contient d'abord le nombre de configurations de *floor*, puis pour chacune des configurations, son type suivi de la configuration elle-même (voir section 5.4.2.a).

```
[floor_count] = lire 6 bits comme un entier non signé et ajouter 1
pour [i] dans 0 .. [floor_count - 1] faire
     [floor_type] = lire 16 bits comme un entier non signé
     si [floor_type] == 0 alors
        [floor_configurations[i]] = décoder entête de floor type 0
     sinon si [floor_type] == 1 alors
        [floor_configurations[i]] = décoder entête de floor type 1
     sinon
        flux non decodable
     fin si
fin pour
```

5.2.3.d Residue

Idem pour la lecture des configurations de *residue*, on récupère le nombre de configurations à lire, puis pour chacune des configurations, on identifie son type et on décode le type correspondant (section 5.4.3.a).

```
[residue_count] = lire 6 bits comme un entier non signé et ajouter 1
pour [i] dans 0 .. [residue_count - 1] faire
   [residue_type] = lire 16 bits comme un entier non signé
   si [residue_type] == 0 alors
      [residue_configurations[i]] = décoder un residue de type 0
   fin si
   si [residue_type] == 1 alors
      [residue_configurations[i]] = décoder un residue de type 1
   fin si
   si [residue_type] == 2 alors
      [residue_configurations[i]] = décoder un residue de type 2
   fin si
   si [residue_type] > 2 alors
      flux non décodable
   fin si
fin pour
```

5.2.3.e Mapping

Là encore, on récupère le nombre de configurations de *mapping* à lire, puis pour chacune on identifie son type et on décode le type correspondant.

```
[mapping_count] = lire 6 bits comme un entier non signé et ajouter 1
pour [i] dans 0 .. [mapping_count - 1] faire
```

```
    [mapping_type] = lire 16 bits comme une entier non signé
    si [mapping_type] == 0 alors
       décoder un mapping de type 0
    sinon
       flux non décodable
    fin si
fin pour
```

Il n'existe qu'un seul type de *mapping* dans la norme actuelle, mais la norme est extensible. Le décodage du type 0 de *mapping* est présenté section 5.4.4.a.

5.2.3.f Mode

Idem, le nombre de configurations est à lire en premier, puis chacune des configurations est décodée.

```
[mode_count] = lire 6 bits comme un entier non signé et ajouter 1
pour [i] dans 0 .. [mode_count - 1] faire
   [mode_configurations[i]] = décoder un mode
fin pour
```

Le décodage d'un mode est décrit section 5.4.5.a.

5.2.3.g Framing flag

La lecture de l'en-tête 3 se finit par la lecture d'un bit dit de *framing* qui permet de vérifier que l'on a bien conservé la synchronisation du flux pendant l'ensemble de la lecture de l'en-tête. Il peut être jeté une fois l'opération terminée.

```
[framing_flag] = lire 1 bit
si [framing_flag] != 1 alors
   flux non décodable
fin si
```

La lecture de l'en-tête de configuration est maintenant finie.

5.3 Décodage des paquets audio Vorbis

Cette section présente les étapes de décodage d'un paquet du flux Vorbis : la lecture des informations liées au paquet, puis les différentes opérations du décodage (décrites au chapitre 3).

5.3.1 Décodage du flux : récupération des informations

Le décodage d'un paquet audio commence par la lecture du flux pour récupérer les données nécessaires au décodage du paquet. Vous trouverez dans cette section l'ensemble des lectures illustrées sur la figure 3.9 par des flèches pleines.

5.3.1.a Type de paquet

La première lecture dans le flux vorbis est le type de paquet. On procède comme suit.

```
[packet_type] = lire 1 bit
si [packet_type] != 0 alors
   paquet incorrect
fin si
```

5.3.1.b Mode

L'étape suivante de la lecture du paquet est la récupération du mode à utiliser pour le paquet courant, ainsi que les informations complémentaires sur la fenêtre s'il y a lieu.

```
[mode_number] = lire ilog([mode_count - 1]) bits
si [mode_blockflag] == 0 alors
   [n] = [blocksize_0]
sinon
   [n] = [blocksize_1]
fin si

si [mode_blockflag] == 1 alors
   [previous_window_flag] = lire 1 bit
   [next_window_flag] = lire 1 bit

   si [previous_window_flag] == 0 alors
      fenetre hybride à gauche
   fin si
   si [next_window_flag] == 0 alors
      fenetre hybride à droite
   fin si
fin si
```

5.3.1.c Mapping ⇒ *floor*, *residue*

Les prochaines étapes du décodages sont faites sous le contrôle du *mapping* courant (déterminé par le mode du paquet), car seul lui connait les informations qui permettront de réaliser les lectures dans le flux (les *floors* à utiliser, les *residues*, *etc*). Son travail se résume en les tâches suivantes.

i) Floor Les *floors* sont codés dans l'ordre dans le flux et sont donc lus un par un.

```
pour [i] dans 0 .. [audio_channels - 1] faire
  [submap_number] = [mapping_mux[i]]
  [floor] = [submap_floor[submap_number]]
  [floor_type] = [floor_types[floor_number]]

  [ret] = décoder le floor pour le canal [i] en fonction de son type.

  si [ret] == <unused> alors
     [no_residue[i]] = true
  fin si
fin pour
```

En cas de fin de paquet durant le décodage des *floors*, tous les canaux sont nuls, et on passe directement à l'étape de recouvrement.

Le décodage d'un *floor* selon son type est décrit section 5.4.2.b.

ii) Non-zero propagate De manière générale, lorsque le *floor* d'un canal est non utilisé, le flux ne contient pas de *residue* codé pour ce canal. Cependant pour des soucis de couplage, on doit maintenir des vecteurs pour l'ensemble des couples. En effet, le couplage peut utiliser de l'entrelacement de canal. Dans ce cas, on pourrait entrelacer un canal nul avec un canal non-nul, ce qui donnerait deux canaux non nuls. Cependant, les canaux marqués comme non-utilisés ne sont pas décodés, on aurait alors une partie des *residues* manquante au moment du désentrelacement.

```
pour [i] dans 0 .. [mapping_coupling_steps - 1] faire
   [magnitude_channel] = [mapping_magnitude[i]]
```

```
   [angle_channel] = [mapping_angle[i]]
   si ([no_residue[magnitude_channel]] == false) OU
      ([no_residue[angle_channel]] == false) alors
      [no_residue[magnitude_channel]] = false
      [no_residue[angle_channel]] = false
   fin si
fin pour
```

iii) Residues Contrairement aux *floors*, les *residues* ne sont pas codés dans l'ordre des canaux dans le flux, mais dans l'ordre des *submap*.

```
pour [i] dans 0 .. [mapping_submaps - 1] faire
   [ch] = 0
   pour [j] dans 0 .. [audio_channels - 1] faire
      si [mapping_mux[j]] == [i] alors
         si [no_residue[j]] == true
            [do_not_decode_flag[ch]] = true
         sinon
            [do_not_decode_flag[ch]] = false
         fin si
         incrémenter [ch]
      fin si
   fin pour

   [residue_number] = [mapping_submap_residue[i]]
   [residue_type] = [residue_types[residue_number]]

   décoder [ch] vecteurs en utilisant [residue_type] et [do_not_decode_flag]
          dans [decoded_residues]

   [ch] = 0
   pour [j] dans 0 .. [audio_channels - 1] faire
      si [mapping_mux[j]] == [i] alors
         [residues[j]] = [decoded_residues[ch]]
         incrémenter [ch]
      fin si
   fin pour
fin pour
```

Le décodage des *residues* est détaillé section 5.4.3.b.

5.3.2 Traitement des informations

Une fois que toutes les informations ont été extraites du flux Vorbis, il faut appliquer les opérations permettant de regénérer les PCMs.

5.3.2.a Mode

À l'issue de la lecture des informations du *mode* on peut préparer le filtre qui sera utilisé dans l'iMDCT. La préparation de l'enveloppe (fenêtre) se fait à partir de l'algorithme présenté en section 3.3.3. Les indices de début et de fin de pente sont calculés comme suit, avec `[n]` qui représente la taille de la fenêtre courante (`blocksize_0` ou `blocksize_1`).

```
si fenêtre hybride à gauche alors
    [debut_pente_gauche] = [n] / 4 - [blocksize_0] / 4
    [fin_pente_gauche] = [n] / 4 + [blocksize_0] / 4
sinon
    [debut_pente_gauche] = 0
```

```
    [fin_pente_gauche] = [n] / 2
fin si
si fenêtre hybride à droite alors
    [debut_pente_droite] = [n] * 3/4 - [blocksize_0] / 4
    [fin_pente_droite] = [n] * 3/4 + [blocksize_0] / 4
sinon
    [debut_pente_droite] = [n] / 2
    [fin_pente_droite] = [n]
fin si
```

5.3.2.b Mapping

Toujours sous le contrôle du mapping, un certain nombre de traitements doivent être effectués.

i) Floor Après lecture des informations dans le flux, la courbe à gros grain du *floor* peut être générée selon le principe de la figure 3.3. L'algorithme est donné section 5.4.2.c.

ii) Inverse coupling La première opération sert à inverser le couplage des canaux, pour regénérer les *residues* propres à chaque canaux plutôt que les *residues* couplés. On utilise pour cela l'algorithme suivant (qui représente une rotation dans le référentiel utilisé).

```
pour [i] dans [vorbis_mapping_coupling_steps - 1] .. 0
    [magnitude_vector] = [residues[[vorbis_mapping_magnitude[i]]]
    [magnitude_angle] = [residues[vorbis_mapping_angle[i]]]
    pour [j] dans 0 .. taille([magnitude_vector]) - 1 faire
        [M] = [magnitude_vector[j]];
        [A] = [angle_vector[j]];
        si [M] > 0 alors
            si [A] > 0 alors
                [angle_vector[j]] = [M] - [A]
            sinon
                [angle_vector[j]] = [M]
                [magnitude_vector[j]] = [M] + [A]
            fin si
        sinon
            si [A] > 0 alors
                [angle_vector[j]] = [M] + [A]
            sinon
                [angle_vector[j]] = [M]
                [magnitude_vector[j]] = [M] - [A]
            fin si
        fin si
    fin pour
fin pour
```

5.3.2.c Dot product

Après l'inverse coupling, chaque canal dispose de ses informations de *floor* et de *residues*. L'opération suivante consiste en une reconstruction du signal fréquentiel à partir de ces deux éléments. L'algorithme utilisé est un produit scalaire (*dot product* en anglais), qui consiste en une multiplication coefficient à coefficient des *floors* avec les *residues*.

5.3.2.d iMDCT + fonction de fenêtrage

Une fois que le signal fréquentiel est récupéré en entier, il doit être converti en temporel. Ceci se fait en utilisant l'iMDCT, présentée précédement. Le N de l'iMDCT représente bien ici la taille de la fenêtre.

N'oubliez pas de multiplier ensuite par la fonction de fenêtrage !

5.3.2.e Overlap add

La dernière opération pour obtenir le signal temporel final, tel qu'il sera écouté ou presque, est le recouvrement. Cette opération consiste en une addition des données issues de la fenêtre précédente avec les données issues de la fenêtre courante (figure 3.7). Une fois que les indices qui se recouvrent sont calculés (*cf.* section 3.3.3), il s'agit simplement d'une addition de vecteurs.

Attention cependant, les seules données valides après cette opération sont les données qui se recouvraient. Les données non encore additionnées sont à conserver et à garder pour la fenêtre suivante. La première fenêtre ne sert qu'à initialiser, rien n'est transmis après son décodage.

5.3.2.f Production des échantillons PCM

Les PCMs sont disponibles après l'opération de recouvrement. La dernière étape à prendre en compte concerne les canaux, et l'ordre défini dans la norme :

1 canal : son monophonique ;

2 canaux : son stéréophonique, gauche puis droite ;

3 canaux : son « 1d-surround », gauche, centre, droite ;

4 canaux : son quadriphonique, avant gauche, avant droite, arrière gauche, arrière droite ;

5 canaux : son surround, avant gauche, centre, avant droite, arrière gauche, arrière droite ;

6 canaux : son 5.1, même ordre que 5 canaux, le 6ème représente le LFE (Low Frequency Effect, la basse) pour le caisson de basse ;

7 canaux : son 6.1, avant gauche, centre, avant droite, côté gauche (dans le 6.1, on a 3 enceintes devant, 2 directement sur le coté, et une derrière), côté droit, arrière centre, LFE ;

8 canaux : son 7.1, même ordre que le 6.1 sauf que l'arrière est remplacé par un arrière gauche et un arrière droite (dans l'ordre).

5.4 Description détaillée des principales opérations

Cette section décrit en détail les principales sous-parties : *codebook*, *floors*, *residue*, *mapping* et *mode*. Pour chacune, il s'agit :

— de la partie « initialisation », en fait réalisée lors de la lecture de l'en-tête 3 (section 5.2.3) ;
— le cas échéant, de la partie de « décodage » réalisée lors de la lecture de chaque paquet (section 5.3) ;
— de tous les éléments spécifiques.

5.4.1 Codebook

5.4.1.a Décodage de l'en-tête

Le décodage des *codebooks* dans le flux Vorbis est complexe de par le nombre de différents cas possibles. Le décodage d'un *codebook* se découpe en deux parties : la récupération de l'arbre de Huffman et sa construction, puis la génération des vecteurs de quantification associés.

Tous les *codebooks* commencent par un schéma de synchronisation, 0x564342. Avant de commencer à décoder, il faut vérifier la validité de ce champ :

```
[vorbis_sync_pattern] = lire 24 bits comme un entier non signé
si [vorbis_sync_pattern] != 0x564342 alors  flux non décodable
```

On récupère ensuite les deux paramètres communs à tous les modes de représentation des *codebooks* : `[codebook_entries]` et `[codebook_dimensions]`, qui représentent respectivement le nombre d'éléments dans le *codebook*, et la taille du vecteur de quantification associé.

```
[codebook_dimensions] = lire 16 bits comme un entier non-signé
[codebook_entries] = lire 24 bits comme un entier non-signé
```

Comme décrit en 3.2.4, un arbre de Huffman est complètement représenté si on donne l'association valeur/longueur de l'ensemble de l'alphabet concerné. On distingue trois variantes de cette représentation dans le Vorbis :

— le mode ordonné, dans lequel les entrées sont stockées dans l'ordre des longueurs associées ;
— le mode non-ordonné plein, dans lequel toutes les entrées sont représentées, et toutes les longueurs sont données dans l'ordre des entrées ;
— le mode non-ordonné creux, dans lequel certaines entrées peuvent ne pas être représentées dans l'arbre.

Dans le flux, on procède alors comme suit, avec `[codeword_lengths]` un vecteur de taille `codebook_entries` :

```
[ordered] = lire 1 bit
si [ordered] == 0 alors
    [sparse] = lire 1 bit
    pour [entry] dans 0 .. [codebook_entries - 1] faire
        si [sparse] == 1 alors
            [used] = lire 1 bit
            si [used] == 1 alors
                [length] = lire 5 bits comme un entier non signé
                [codeword_lengths[entry]] = [length + 1]
            sinon
                [codeword_lengths[entry]] = <unused>
            fin si
        sinon // sparse == 0
            [length] = lire 5 bits comme un entier non signé
            [codeword_lengths[entry]] = [length + 1]
        fin si
    fin pour
sinon // ordered == 1
    [entry] = 0
    [length] = lire 5 bits comme un entier non signé
    tant que [entry] < [codebook_entries] faire
        [length] = [length + 1]
        [num_bits] = ilog([codebook_entries] - [entry])
        [entry_number] = lire [num_bits] bits comme un entier non signé
        pour [i] dans [entry] .. [entry + entry_number - 1] faire
            [codeword_lengths[i]] = [length]
        fin pour
        [entry] = [entry] + [entry_number]
    fin tant que
    si [entry] > [codebook_entries] alors  flux non décodable
fin si
```

Une fois qu'on a récupéré `[codeword_lengths]`, on peut donc construire l'arbre de Huffman. Nous vous laissons libre de choisir l'algorithme à utiliser pour construire l'arbre.

La représentation des vecteurs de quantification se fait en deux étapes. D'abord on récupère les paramètres nécessaires au décodage des vecteurs, puis on décode l'ensemble de ces vecteurs. Concernant la représentation, on distingue trois types :

— le type 0, dans lequel il n'y a pas de vecteur (type scalaire) ;
— le type 1 construit sa table à partir des informations contenues dans le fichier ;
— le type 2 lit sa table complètement dans le flux.

La récupération des paramètres nécessaires au décodage des vecteurs se fait selon l'algorithme suivant. Les fonctions `float32_unpack` et `lookup1_values` sont définies après.

```
[codebook_lookup_type] = lire 4 bits comme un entier non signé
si [codebook_lookup_type] > 0 alors
    [codebook_minimum_value] = float32_unpack(lire 32 bits comme un entier non
signé)
    [codebook_delta_value] = float32_unpack(lire 32 bits comme un entier non
```

```
signé)
    [codebook_value_bits] = lire 4 bits comme un entier non-signé et ajouter 1
    [codebook_sequence_p] = lire 1 bit
    si [codebook_lookup_type] == 1 alors
        [codebook_lookup_values] =
lookup1_values([codebook_entries],[codebook_dimensions] )
    sinon
        [codebook_lookup_values] = [codebook_entries] * [codebook_dimensions]
    fin si
    pour [value] dans 0 .. [codebook_lookup_values - 1] faire
       [codebook_multiplicands[value]] = lire [codebook_value_bits] bits comme
                un entier non signé
    fin pour
fin si
```

La fonction `float32_unpack` permet de récupérer un nombre flottant à partir de sa représentation entière dans le flux Vorbis. La représentation utilisée se base sur une représentation mantisse/exposant : $m \times 2^e$, avec m la mantisse et e l'exposant. Ceci est fait avec l'algorithme suivant, avec `x` l'entier à transformer.

```
[mantissa] = [x] AND 0x001fffff (non signé)
[sign] = [x] and 0x80000000 (non signé)
[exponent] = [x] AND 0x7fe00000 >> 21 (non signé)
si [sign] != 0 alors
    [mantissa] = -[mantissa]
fin si
return [mantissa] * (2 ^ ([exponent] - 788))
```

La fonction `lookup1_values(a,b)` renvoie la valeur définie comme étant : « le plus grand entier r tel que $r^b \leq a$ ».

Une fois que l'on a extrait tous les paramètres nécessaires, on peut construire les vecteurs de quantification chacun étant associé à une entrée de l'arbre de Huffman. Ces vecteurs `[vq_vector]` sont de taille `[codebook_dimensions]`.

Dans le cas d'un type 1, on a l'algorithme suivant, qui s'applique pour chaque entrée `[entry]` :

```
[last] = 0
[index_divisor] = 1
pour [i] dans 0 .. [codebook_dimensions - 1] faire
    [index] = ([entry] / [index_divisor]) modulo [codebook_lookup_values]
    [vq_vector[i]] = [codebook_multiplicands[index]] * [codebook_delta_value] +
        [codebook_minimum_value] + [last]
    si [codebook_sequence_p] == 1 alors
        [last] = [vq_vector[i]]
    fin si
    [index_divisor] = [index_divisor] * [codebook_lookup_values]
fin pour
```

Dans le cas d'un type 2, l'algorithme est légérement différent :

```
[last] = 0
[index] = [entry] * [codebook_dimensions]
pour [i] dans 0 .. [codebook_dimensions - 1] faire
    [vq_vector[i]] = [codebook_multiplicands[index]] * [codebook_delta_value] +
        [codebook_minimum_value] + [last]
    si [codebook_sequence_p] == 1 alors
        [last] = [vq_vector[i]]
    fin si
    [index] = [index + 1]
fin pour
```

5.4.2 Floors

Nous ne présentons ici que le *floor* de type 1 car les *floors* de type 0 ne font pas partie du sujet, ils ne sont pas à réaliser.

5.4.2.a Décodage de l'en-tête

Les *floors* de type 1 sont rangés dans le paquet par partition, et chaque partition est d'une classe donnée. La première information présente dans le descripteur dans l'en-tête pour un *floor* de type 1 est la liste des classes pour chacune des partitions.

De plus, chaque classe représente un certain nombre de points significatifs pour la construction de la courbe gros grain. Ce nombre de points et les sous classes qui permettront de les lire dans les paquets audio sont ensuite lus dans l'en-tête. On notera la présence des `[masterbooks]` et `[subclass_books]` qui sont des indices de *codebook*.

Pour finir, dans les paquets audio seules les ordonnées de la courbe sont présentes. La dernière partie du descripteur contient les abscisses correspondantes.

```
[floor1_partitions] = lire 5 bits comme un entier non signé
[maximum_class] = -1
pour [i] dans 0 .. [floor1_partitions - 1] faire
   [floor1_partition_class_list[i]] = lire 4 bits comme un entier non signé
fin pour

[maximum_class] = valeur maximum du vecteur [floor1_partition_class_list]

pour [i] dans 0 .. [maximum_class] faire
   [floor1_class_dimensions[i]] = lire 3 bits comme un entier non signé et
       ajouter 1
   [floor1_class_subclasses[i]] = lire 2 bits comme un entier non signé
   si ([floor1_class_subclasses[i]] != 0) alors
      [floor1_class_masterbooks[i]] = lire 8 bits comme un entier non signé
   fin si
   pour [j] dans 0 .. (2 puissance [floor1_class_subclasses[i]]) - 1 faire
      [floor1_subclass_books[i][j]] = lire 8 bits comme un entier non signé et
          soustraire un
        (attention la lecture peut rendre 0)
   fin pour
fin pour

[floor1_multiplier] = lire 2 bits comme un entier non signé et ajouter 1
[rangebits] = lire 4 bits comme un entier non signé
[floor1_X_list[0]] = 0
[floor1_X_list[1]] = 2 puissance [rangebits];
[floor1_values] = 2

pour [i] dans 0 .. [floor1_partitions - 1] faire
   [current_class_number] = [floor1_partition_class_list[i]]
   pour [j] dans 0 .. [floor1_class_dimensions[current_class_number] - 1] faire
      [floor1_X_list[floor1_values]] = lire [rangebits] bits comme un entier
          non signé
      incrémenter [floor1_values] de un
   fin pour
fin pour
```

5.4.2.b Décodage du paquet

Le décodage d'un *floor* de type 1 dans un paquet audio commence par la lecture d'un bit qui permet de savoir si le *floor* courant est utilisé pour ce canal.

```
[nonzero] = lire 1 bit comme un booleen
```

Si le bit [nonzero] est égal à zéro, il n'y a aucune donnée à décoder et le canal en court de décodage n'a pas de *floor*. La fonction de décodage doit donc retourner <unused> (*cf* 5.3.1.c). Sinon le décodage des données se déroule de la manière suivante :

```
[range] = élement [floor1_multiplier - 1] du vecteur { 256, 128, 86, 64 }
[floor1_Y[0]] = lire ilog([range - 1]) bits comme un entier non signé
[floor1_Y[1]] = lire ilog([range - 1]) bits comme un entier non signé
[offset] = 2;

pour [i] dans 0 .. [floor1_partitions - 1] faire
   [class] = [floor1_partition_class_list[i]]
   [cdim] = [floor1_class_dimensions[class]]
   [cbits] = [floor1_class_subclasses[class]]
   [csub] = (2 puissance [cbits]) - 1
   [cval] = 0
   si ([cbits > 0) alors
      [cval] = lire dans le paquet en utilisant le codebook scalaire numéro
               ([floor1_class_masterbooks[class]])
   fin si
   pour [j] dans 0 .. [cdim - 1] faire
      [book] = [floor1_subclass_books[class][cval AND csub]]
      [cval] = [cval] décalé à droite de [cbits] bits
      si ([book] >= 0) alors
         [floor1_Y[[j]+[offset]] = lire dans le paquet en utilisant le codebook
                        scalaire [book]
      sinon
         [floor1_Y[[j]+[offset]] = 0
      fin si
   fin pour
   [offset] = [offset] + [cdim]
fin pour
```

Les données rangées dans le paquet audio pour un *floor* de type 1 correspondent aux ordonnées de l'ensemble des points décrits dans la configuration de *floor* courante. On lit tout d'abord les deux premières valeurs de manière systématique, pour rappel elles correspondent à la première et à la dernière valeur de la courbe. On lit ensuite, pour chaque partition, une valeur qui nous permettra de connaître les sous classes à utiliser en utilisant le *codebook* `masterbooks[class]` en mode scalaire. Les ordonnées sont ensuite lues grâce au *codebook* des sous classes ainsi décrites en mode scalaire toujours.

5.4.2.c Synthèse de la courbe

La première phase de la synthèse de la courbe est le calcul en amplitude des ordonnées. En effet, seules la première et la dernière valeur de la courbe sont placées tel quel dans le paquet audio. Seules des valeurs différentielles sont ensuite placées pour les autres points de la courbe. Cette étape consiste donc à rétablir les valeurs en lieu et place des valeurs différentielles.

```
[range] = vector { 256, 128, 86, 64 } element [floor1_multiplier - 1]
[floor1_step2_flag[0]] = set
[floor1_step2_flag[1]] = set
[floor1_final_Y[0]] = [floor1_Y[0]]
[floor1_final_Y[1]] = [floor1_Y[1]]

pour [i] dans 2 .. [floor1_values - 1] faire
   [low_neighbor_offset] = low_neighbor([floor1_X_list], [i])
   [high_neighbor_offset] = high_neighbor([floor1_X_list], [i])

   [predicted] = render_point( [floor1_X_list[low_neighbor_offset]],
```

```
                                [floor1_final_Y[low_neighbor_offset]],
                                [floor1_X_list[high_neighbor_offset]],
                                [floor1_final_Y[high_neighbor_offset]],
                                [floor1_X_list[i]] )
   [val] = [floor1_Y[i]]
   [highroom] = [range] - [predicted]
   [lowroom] = [predicted]
   si ( [highroom] < [lowroom] ) alors
      [room] = [highroom] * 2
   sinon
      [room] = [lowroom] * 2
   fin si

   si ([val] != 0) alors
      [floor1_step2_flag[low_neighbor_offset]] = set
      [floor1_step2_flag[high_neighbor_offset]] = set
      [floor1_step2_flag[i]] = set

      si ([val] >= [room]) alors
         si ( [highroom] > [lowroom] ) alors
            [floor1_final_Y[i]] = [val] - [lowroom] + [predicted]
         sinon
            [floor1_final_Y[i]] = [predicted] - [val] + [highroom] - 1
         fin si
      sinon
         si ([val] est impair) alors
            [floor1_final_Y[i]] = [predicted] - (([val] + 1) / 2)
         sinon
            [floor1_final_Y[i]] = [predicted] + ([val] / 2)
         fin si
      fin si
   sinon
      [floor1_step2_flag[i]] = unset
      [floor1_final_Y[i]] = [predicted]
   fin si
fin pour
```

Pour calculer la valeur en amplitude de chacun des points de la courbe, cette fonction doit interpoler entre ses deux voisins les plus proches (un de chaque côté) dont on connait déjà la valeur (dans l'ordre dans lequel sont rangées les valeurs). On fait appel pour cela à deux fonctions `low_neighbor` et `high_neighbor`. L'interpolation est réalisée par la fonction `render_point`. Muni de la valeur prédite et de la valeur lue on peut donc déduire la valeur réelle pour le point en question. On notera que le codage des valeurs est un peu différent lorsqu'on approche des valeurs extrêmes. On notera aussi que la lecture d'une valeur nulle dans le paquet abouti à un cas où on utilise directement la valeur prédite conformément à la description du format.

La seconde phase de la synthèse de la courbe gros grain est l'interpolation linéaire entre chacun des couples de points décrits par la configuration de *floor* courante. Vous aurez remarqué jusqu'alors que les points de la courbes ne sont pas ordonnés correctement pour cette étape. La première chose à faire est donc d'ordonner les vecteurs `[floor1_final_Y]`, `[floor1_step2_flags]` et `[floor1_X_list]` dans l'ordre des valeurs de `[floor1_X_list]` croissantes. Les vecteurs ordonnés sont notés `[floor1_final_Y_o]`, `[floor1_step2_flags_o]` et `[floor1_X_list_o]`. À partir de ces trois vecteurs ordonnés, on peut faire appel à la fonction `render_line` pour chacun des couples de deux points afin de peupler le vecteur `[floor]`. Ce vecteur est de taille `n2` (ou faut-il lire $\frac{N}{2}$). Au besoin, on étend la valeur jusqu'à la fin du vecteur. Dans le cas contraire, si le *floor* courant est plus long que la fenêtre en cours, il faudra songer à ne pas déborder du vecteur, on tronque le *floor*. L'étape finale de cette fonction est la production des valeurs finales, puisque l'ensemble des valeurs produites jusqu'alors sont des indices dans la table `[floor1_inverse_dB_static_table]` (cette table vous est donnée dans le fichier d'exemple `floor_type1_exemple.c`).

```
[hx] = 0
[lx] = 0
[ly] = [floor1_final_Y_o[0]] * [floor1_multiplier]
```

```
pour [i] dans 1 .. [floor1_values - 1] faire

   si ([floor1_step2_flag_o[i]] is set ) alors
      [hy] = [floor1_final_Y_o[i]] * [floor1_multiplier]
      [hx] = [floor1_X_list_o[i]]
      render_line( [lx], [ly], [hx], [hy], [floor] )
      [lx] = [hx]
      [ly] = [hy]
   fin si
fin pour

si ( [hx] < [n2] ) alors
   render_line( [hx], [hy], [n2], [hy], [floor] )
fin si

si ( [hx] > [n2] ) alors
   tronquer [floor] à [n] élements
fin si

pour [i] dans 0 .. [n2 - 1] faire
   [v[i]] = [floor1_inverse_dB_static_table[floor[i]]]
fin pour
```

5.4.2.d Fonctions pratiques

i) `low_neighbor` La fonction `low_neighbor(v, x)` renvoie la position `n` tel que `v[n]` soit le plus grand élément inférieur à `v[x]` et que `n` soit inférieur à `x`.

ii) `high_neighbor` La fonction `high_neighbor(v, x)` renvoie la position `n` tel que `v[n]` soit le plus petit élément supérieur à `v[x]` et que `n` soit inférieur à `x`.

iii) `render_point` Cette fonction permet de calculer l'interpolation de la valeur en un point précis.

```
[dy] = [y1] - [y0]
[adx] = [x1] - [x0]
[ady] = absolute value of [dy]
[err] = [ady] * ([X] - [x0])
[off] = [err] / [adx] using integer division
si ([dy] < 0) alors
   [Y] = [y0] - [off]
sinon
   [Y] = [y0] + [off]
fin si
```

iv) `render_line` Cette fonction est une extension de l'algorithme de Bresenham[1] pour le tracé de ligne.

```
[dy] = [y1] - [y0]
[adx] = [x1] - [x0]
[ady] = valeur_absolue([dy])
[base] = [dy] / [adx] (division entière)
[x] = [x0]
[y] = [y0]
[err] = 0
si ([dy] < 0) alors
   [sy] = [base] - 1
```

```
sinon
   [sy] = [base] + 1
fin si
[ady] = [ady] - valeur_absolue([base]) * [adx]
[v[x]] = [y]
pour [x] dans [x0 + 1] .. [x1 - 1] faire
   [err] = [err] + [ady];
   si ( [err] >= [adx] ) alors
      [err] = [err] - [adx]
      [y] = [y] + [sy]
   sinon
      [y] = [y] + [base]
   fin si
   [v[x]] = [y]
fin pour
```

5.4.3 Residues

Les trois types (0, 1 et 2) de *residue* sont relativement similaires. Une bonne partie de leur traitement est donc commun.

Étant donné que la grosse différence entre les trois types de *residue* actuels de la norme Vorbis réside essentiellement dans l'organisation des données, l'en-tête est commun aux trois types. D'un autre côté, même si l'organisation des données dans le paquet audio est un peu différente (type 0 ou 1) ou encore que les canaux sont tous codés ensemble (type 2), une partie non négligeable du décodage du paquet est elle aussi commune.

Les sections suivantes décrivent d'abord les parties communes aux trois types de *residue*, puis les particularités de chacun.

5.4.3.a Décodage de l'en-tête (partie commune)

Le descripteur de *residue* (type 0, 1 ou 2) présent dans l'en-tête 3 se présente de la manière suivante. Ce descripteur commence par des informations globales, à savoir, le début et la fin de la portion de vecteur codé par ce descripteur de *residue*. Cette portion se subdivise en partitions dont la taille est donnée par le troisème champ. Ces partitions ont une classification (le nombre maximum est donné par le quatrième champ). On trouve enfin la référence au *codebook* qui permettra de connaître les classifications réellement utilisées par lecture en mode scalaire dans le paquet audio.

La lecture du fichier, et la production des valeurs de *residue* se fait en plusieurs passes (8). On lit donc ensuite pour chaque classification un mot de 8 bits qui représente par chaque bit si pour cette passe la classification code des valeurs.

La dernière étape est enfin de récupérer pour chaque combinaison utilisée (comme précédemment expliqué précédement) le *codebook* qui permettra de lire les valeurs en mode VQ.

```
[residue_begin] = lire 24 bits comme un entier non signé
[residue_end] = lire 24 bits comme un entier non signé
[residue_partition_size] = lire 24 bits comme un entier non signé et ajouter 1
[residue_classifications] = lire 6 bits comme un entier non signé et ajouter 1
[residue_classbook] = lire 8 bits comme un entier non signé

pour (i] dans 0 .. [residue_classifications - 1] faire
   [high_bits] = 0
   [low_bits] = lire 3 bits comme un entier non signé
   [bitflag] = lire 1 bit comme un booleen
   si ( [bitflag] est vrai ) alors
      [high_bits] = lire 5 bits comme un entier non signé
   fin si
   [residue_cascade[i]] = [high_bits] * 8 + [low_bits]
fin pour
```

```
pour[i] dans 0 .. [residue_classifications - 1] faire
   pour [j] dans 0 .. 7 faire
      si ( [residue_cascade[i]] bit [j] == 1 ) alors
         [residue_books[i][j]] = lire 8 bits comme un entier non signé
      sinon
         [residue_books[i][j]] = <unused>
      fin si
   fin pour
fin pour
```

5.4.3.b Décodage du paquet audio (partie commune)

La partie commune aux trois types de *residue* pour le décodage des paquet audio correspond au squelette de récupération des partitions une par une.

En premier lieu, un certain nombre de variables peuvent être utiles pour réaliser ce décodage. comme la taille du vecteur de sortie ou encore la taille réellement codée par ce *residue* (en effectuant une intersection entre la taille totale du vecteur et la taille totale du *residue* codé).

On récupère la taille des *codebook*, on calcule aussi le nombre de valeurs à lire et enfin le nombre de partitions à lire.

La lecture se déroule donc en 8 passes. La première passe possède un traitement supplémentaire qui consiste à décoder les classifications à utiliser pour lire les valeurs à suivre. Ce décodage s'effectue par division euclidienne par `[residue_classifications]` où le reste est l'indice de la classification de la i^{eme} partition.

On décode enfin les partitions une par une en utilisant les informations fraichement produites qui nous permettent de récupérer le *codebook* de référence. On notera que cette lecture de partition est spécifique au type de *residue*, elle sera donc expliquée dans les partie suivantes.

```
[actual_size] = current blocksize/2;
si ([residue_type] == 2) alors
   [actual_size] = [actual_size] * [ch]
fin si
[limit_residue_begin] = minimum ([residue_begin], [actual_size]);
[limit_residue_end] = minimum ([residue_end], [actual_size]);


[classwords_per_codeword] = [codebook_dimensions] du codebook [
    residue_classbook]
[n_to_read] = [limit_residue_end] - [limit_residue_begin]
[partitions_to_read] = [n_to_read] / [residue_partition_size]

mettre à zéro les vecteurs de sortie

si ([n_to_read] == 0) alors
   Stop. Pas de residue à lire.
fin si

pour [pass] dans 0 .. 7 faire
   [partition_count] = 0
   tant que ([partition_count] < [partitions_to_read]) faire
      si ( [pass] == 0 ) alors
         pour [j] dans 0 .. [ch - 1] faire
            si (non  [do_not_decode[j]]) alors
               [temp] = lire dans le paquet en utilisant le codebook [
                   residue_classbook]
                                    en mode scalaire
               pour [i] dans [classwords_per_codeword - 1] ... 0 faire
                  [classifications[j][i + [partition_count]] =
                                   [temp] modulo [residue_classifications]
```

```
                                (modulo entier)
                [temp] = [temp] / [residue_classifications]
                         (division entière)
             fin pour
          fin si
       fin pour
    fin si
    pour [i] dans 0 .. [classwords_per_codeword - 1] et     | Faire une seule
    tant que [partition_count] < [partitions_to_read] faire | boucle ici
       pour [j] dans 0 .. [ch - 1] faire
          si (non [do_not_decode[j]]) alors
             [vqclass] = [classifications[j][partition_count]]
             [vqbook] = [residue_books[vqclass][pass]]
             si ([vqbook] non <unused>) alors
                décoder la partition dans le vecteur numéro [j] en commençant
                   à la case [limit_residue_begin]+[partition_count]*[
                       residue_partition_size]
                   en utilisant le codebook [vqbook] en mode VQ
             fin si
          fin si
       fin pour
       incrémenter [partition_count] de un
    fin pour
 fin pour
fin pour
```

5.4.3.c Spécificités du type 0

La seule spécificité du *residue* de type 0 réside dans la lecture des partitions. Dans ce *residue*, les valeurs d'un même canal sont entrelacées dans les vecteurs de VQ comme suit.

```
          vecteur de residue original: [0 1 2 3 4 5 6 7 ]
codebook dimensions = 8  encodé comme: [0 1 2 3 4 5 6 7 ]
codebook dimensions = 4  encodé comme: [0 2 4 6], [1 3 5 7]
codebook dimensions = 2  encodé comme: [0 4], [1 5], [2 6], [3 7]
codebook dimensions = 1  encodé comme: [0], [1], [2], [3], [4], [5], [6], [7]
```

Le décodage d'une partition se présente donc comme suit. On produit donc `[n]` valeurs dans le vecteur `[v]` en commençant à la case `[offset]`.

```
[n] = [residue_partition_size]
[v] = vecteur de sortie
[offset] décalage dans le vecteur [v]

[step] = [n] / [codebook_dimensions]
pour [i] dans 0 .. [steps - 1]
   [entry_temp] = lire le vecteur à partir du paquet en utilisant le codebook
       courant en mode VQ
   pour [j] dans 0 .. [codebook_dimensions - 1] faire
      [v[offset+i+j*step]] = [v[offset+i+j*step] + [entry_temp[j]]
   fin pour
fin pour
```

5.4.3.d Spécificités du type 1

La seule spécificité du *residue* de type 1 est là aussi la lecture des partitions. Dans ce *residue*, les valeurs d'un même canal sont codées dans l'ordre dans les vecteurs de VQ comme suit.

```
          vecteur de residue original: [0 1 2 3 4 5 6 7]
codebook dimensions = 8  encodé comme: [0 1 2 3 4 5 6 7]
codebook dimensions = 4  encodé comme: [0 1 2 3], [4 5 6 7]
codebook dimensions = 2  encodé comme: [0 1], [2 3], [4 5], [6 7]
codebook dimensions = 1  encodé comme: [0], [1], [2], [3], [4], [5], [6], [7]
```

Là aussi le décodage produit `[n]` valeurs dans le vecteur `[v]` en commençant à la case `[offset]`. Ces valeurs sont lues de la manière suivante.

```
[n] = [residue_partition_size]
[v] = vecteur de sortie
[offset] décalage dans le vecteur [v]

[i] = 0
tant que ([i] < [n]) faire
   [entry_temp] = lire le vecteur à partir du paquet en utilisant le codebook
       courant en mode VQ
   pour [j] dans 0 .. [codebook_dimensions - 1] faire
      [v[offset+i]] = [v[offset+i]] + [entry_temp[j]]
      incrémenter [i] de 1
   fin pour
fin tant que
```

5.4.3.e Spécificités du type 2

La dernier type de *residue* est legèrement différent des deux autres en ce sens que la différence ne réside pas uniquement dans la manière de lire les partitions.

En effet, ce type de *residue* code l'ensemble des canaux en un seul canal entrelacé. On peut donc voir ce type de *residue* comme un *residue* de type 1 avec un vecteur de taille `ch*n2`

Une autre spécificité de ce *residue* vient de sa manière de gérer les marquages 'do_not_decode'. Il n'y a aucun décodage à faire si l'ensemble des canaux est marqué.

On finit par désentrelacer les valeurs.

```
1. si (do_not_decode[0 .. [ch-1]] == vrai) alors
      produire un vecteur de taille n2*ch de zéro et passer à 3
   fin si

2. Décoder un residue de type 1 pour un canal de taille n2*ch

3. pour [i] dans 0 .. [n - 1] faire
      pour [j] dans 0 .. [ch - 1] faire
         vecteur de sortie [j] élément [i] = [v[i*ch + j]]
      fin pour
   fin pour
```

5.4.4 Mapping

5.4.4.a Décodage de l'en-tête

Il n'existe qu'un type de *mapping* dans la norme actuelle, le *mapping* de type 0. Son décodage dans l'en-tête 3 se présente comme suit :

```
[flag] = lire 1 bit comme un booleen
si [flag] == 1 alors
   [mapping_submaps] = lire 4 bits comme un entier non signé et ajouter 1
sinon
```

```
   [mapping_submaps] = 1
fin si
[flag_square_polar] = lire 1 bit comme un booleen
si [flag_square_polar] == 1 alors
   [mapping_coupling_steps] = lire 8 bits comme un entier non signé et ajouter
       1
   pour [j] dans 0 .. [mapping_coupling_steps - 1] faire
       [mapping_magnitude[j]] = lire ilog([audio_channels - 1]) bits comme un
entier non signé
       [mapping_angle[j]] = lire ilog([audio_channels - 1]) bits comme un
           entier
non signé
       si ([mapping_magnitude[j]] > [audio_channels - 1]) OU
          ([mapping_angle[j]] > [audio_channels - 1]) alors
          flux non décodable
       fin si
    fin pour
sinon
   [mapping_coupling_steps] = 0
fin si

[reserved] = lire 2 bits
si [reserved] != 0 alors
   flux non décodable
fin si

si [mapping_submaps] > 1
   pour [j] dans 0 .. [audio_channels - 1] faire
       [mapping_mux[j]] = lire 4 bits comme un entier non signé
       si [mapping_mux[j]] > [mapping_submaps - 1] alors
          flux non décodable
       fin si
   fin pour
sinon
   pour [j] dans 0 .. [audio_channels - 1] faire
       [mapping_mux[j]] = 0
   fin pour
fin si

pour [j] dans 0 .. [mapping_submaps - 1] faire
   [discard] = lire 8 bits (non utilisés)
   [mapping_submap_floor[j]] = lire 8 bits comme un entier non signé
   si [mapping_submap_floor[j]] > [floor_count - 1] alors
      flux non décodable
   fin si
   [mapping_submap_residue[j]] = lire 8 bits comme un entier non signé
   si [mapping_submap_residue[j]] > [residue_count - 1] alors
      flux non décodable
   fin si
fin pour
```

On remarque donc que dans le décodage du type 0 de *mapping*, on décode dans l'ordre les informations contenues dans le mapping (*cf.* 3.4). On commence par décoder les éventuelles informations relatives au couplage canal, uniquement le *square_polar*. Le couplage canal se réalisant en étape, pour chaque étape, on récupère l'indice du canal contenant le module (*magnitude*) et l'argument (*angle*).

Dans un second temps on récupère les informations relative au *submap* à utiliser pour chacun des canaux dans le mapping courant. Les *submaps* représentent une association floor/residue.

Enfin, on récupère les descriptions des différents *submap*, à savoir, le *floor* et le *residue* à utiliser pour chaque *submap*.

5.4.5 Mode

5.4.5.a Décodage de l'en-tête

Il n'existe pas d'extension possible sur ce point dans la norme Vorbis, on lit donc systématiquement le même type d'entrée pour toutes les configurations de *mode*.

Le décodage d'un mode se présente comme suit.

```
[mode_blockflag] = lire 1 bit
[mode_windowtype] = lire 16 bits comme un entier non signé
[mode_transformtype] = lire 16 bits comme un entier non signé
[mode_mapping] = lire 8 bits comme un entier non signé
si ([mode_windowtype] != 0)           OU
   ([mode_transformtype] != 0)        OU
   ([mode_mapping] >= [mapping_count]) alors
   flux non décodable
fin si
```

Il est à noter que la variable `mode_mapping` affectée ici sert à identifier le mapping utilisé. La liste des mappings associés aux flux Vorbis est maintenue dans une structure **`mappings_setup_t`** du module `mapping`. On récupère donc dans le bloc de code ci-dessus un index dans le tableau `maps` de cette structure.

Il existe tout de même des extensions possibles dans cette partie de la norme, mais sous la forme de champs non utilisés, c'est le cas du `[mode_windowtype]` et du `[mode_transformtype]`, qui doivent être à zéro puisque aujourd'hui la norme n'utilise qu'un type de fenêtre et un seul type de transformée (la MDCT). Ces champs sont présents uniquement pour que les flux actuels soient compatibles avec les versions futures de la norme. Nous ne prendrons donc pas en compte ces champs dans ce projet (mais ils doivent être lus dans l'en-tête).

Chapitre 6

Travail attendu

Vous devez implanter les différentes étapes menant d'un fichier Ogg contenant un flux Vorbis à la production d'échantillons représentant le son en brut dans le format WAV décrit en annexe A. Pour vous permettre de mener à bien un sujet de cette taille dans le temps imparti, une implantation de référence du décodeur est fournie. Elle définit l'architecture générale du projet, via un découpage en modules de tailles raisonnables et de complexités progressives. Chaque module est fourni sous la forme :

— d'un fichier d'en-tête en C (`.h`), décrivant les structures de données et spécifiant les fonctions fournies par ce module ;
— d'un fichier objet (`.o`) contenant le code compilé de l'implantation de référence du module.

Votre travail consistera à reprogrammer les différents modules, pour remplacer les fichiers objets de référence par ceux que vous aurez vous-même programmé.

6.1 Description des modules

L'architecture générale du projet est fournie. [1] La décomposition est *modulaire*, organisée de la manière suivante : [2]

i) Modules des opérations principales du décodage des données Cinq modules correspondent aux opérations détaillées section 5.4 : `mode`, `mapping`, `floor`, `residue` et `codebook`. Ils comportent tous une opération d'initialisation, qui sera invoquée lors de la lecture de l'en-tête 3 (section 5.2.3), et une ou plusieurs fonctions correspondant à la lecture et/ou au traitement des données d'un paquet. Les autres modules gèrent les étapes supplémentaires du décodage.

— `mode` : ce module implante la gestion des *modes*, lecture de l'entête [5.2.3.f, 5.4.5.a] et lecture du paquet audio [5.3.1.b] ;
— `mapping` : ce module implante la gestion des *mappings*, entête et paquets. [5.2.3.e, 5.4.4.a (entête), 5.3.1.c, 5.3.2.b (paquet audio)] ;
— `codebook` [5.2.3.a, 5.4.1.a] ;
— `floor` : ce module n'implante que la base commune de gestion des *floor*. Pour la phase de lecture des configurations, il faut penser à initialiser les données de traitement internes aux deux types (`data0` et `data1`). Il faut appeler la méthode `new()` avant la lecture des configurations. Il faut ensuite appeler la méthode `allocate()` après la lecture de l'ensemble des configurations. [5.2.3.c, 5.4.3.a(entête), 5.4.3.b(paquet audio)].
  La partie spécifique de chaque type de floor est déléguée à deux modules dédiés :
  — `floor_type1` : ce module est lui responsable de la gestion spécifique du type 1 de *floor* ; [5.4.2.a (entête), 5.4.2.b, 5.4.2.c (paquet audio)].
  — `floor_type0` : ce module est lui responsable de la gestion spécifique du type 0 de *floor*. [pas de référence, demandez aux enseignants (module optionnel)].

1. Dommage ? Peut-être car concevoir et implémenter ce projet *from scratch* aurait été très intéressant. Par contre, ce n'est simplement pas possible dans le cadre du projet C, en trinôme, sur la durée impartie. Si vous en doutez, on en reparle à la fin du projet !
2. Les références aux sections précédentes de la documentation sont fournies entre crochets. L'ensemble des informations décrites dans l'ensemble des sections référencées sont à intégrer dans le module.

— `residue` : ce module implante la gestion des *residues* [5.2.3.d, 5.4.3.a(entête), 5.4.3.b(paquet audio)] ;
— `dot_product` : Ce module gère l'opération de produit scalaire entre les *floors* et *residues* obtenus à l'issue des étapes de synthèse de courbe gros grain et de découplage [5.3.2.c] ;
— `envelope` : ce module s'occupe de la partie de génération de la fonction de fenêtrage et du recouvrement (`overlap_add`) [5.3.2.a(fenêtre), 5.3.2.e(recouvrement)] ;
— `time_domain_transform` : Ce module gère lui l'implantation de l'iMDCT pour le passage du domaine fréquentiel au domaine temporel. [5.2.3.b(entête), 5.3.2.d(paquet audio)]. Dans le traitement du paquet, c'est aussi lui qui multiplie le résultat de l'iMDCT avec la fonction de fenêtrage ;
— `fast_imdct` : Ce module gère lui une implantation plus spécifique de l'iMDCT pour le passage du domaine fréquentiel au domaine temporel [pas de référence, demandez aux enseignants (module optionnel)] ;
— plusieurs modules gèrent les en-têtes : `common_header` [5.2] pour la partie commune, ainsi que `header1` [5.2.1], `header2` [5.2.2] et `header3` [5.2.3].

ii) Modules de décodage du flux Ces modules correspondent aux différentes étapes de la gestion du flux :

— `main` : programme principal : ouvre un fichier *Ogg* ;
— `ogg_core` : gère la décompression générale d'un fichier Ogg et appelle la fonction principale de décodage du flux Vorbis `vorbis_main` ;
— `ogg_packet` : gère spécifiquement la reconstruction des paquets Vorbis ;
— `vorbis_main` : fournit la fonction principale de décodage du flux : lecture des entêtes, décompression des paquets et génération des PCMs. C'est aussi ici que sont définies les principales structures de données permettant de stocker les informations de configuration lues dans les en-têtes ;
— `vorbis_packet` : gestion de l'enchainement des lectures et traitements pour la gestion des paquets Vorbis. Ce module définit aussi les principales structures de données représentant les données en cours de décodage ;
— `vorbis_io` : permet principalement d'avancer dans les paquets du flux Vorbis et de lire des bits (n, pas forcément un multiple de 8) dans le paquet courant.

iii) Autre modules

— `pcm_handler` : gère la production des échantillons PCMs dans un fichier au format *.wav* [5.3.2.f] ;
— `helpers` : fonctions "utilitaires", utilisées notamment pour les *floors* [5.4.2.d].

6.2 API fournie

L'API (*Application Programming Interface*), c'est-à-dire les spécifications des structures de données et fonctions fournies par chaque module, est **disponible sur l'Ensiwiki** sous forme d'un fichier `.pdf` navigable ou via une page web. Ces spécifications sont en fait directement inclues entre des balises spécifiques dans les fichiers d'en-tête `.h`. La documentation est ensuite générée à l'aide d'un analyseur (ici, *doxygen*) qui extrait les informations contenues dans le code source pour créer une documentation formatée.

Tous les modules que vous développerez devront suivre les spécifications imposées, afin de rester compatibles avec les modules de références encore utilisés. De manière générale, **il est imposé de conserver les fichiers d'en-tête distribués**, sans modifier ni les structures de données ni les signatures des fonctions.

Pour certains modules, il peut être utile voire nécessaire de rajouter des champs aux structures de données définies dans les en-tête. Vous devrez alors utiliser un mécanisme dit d'« *extension de type* », qui permet de spécialiser une structure générique déja existante. C'est le cas par exemple dans le module *mapping*, où une structure **`mapping_t`** définit les champs communs à tous les types de mapping et une structure **`mapping_type0_t`** y ajoute les données spécifiques au mapping de type 0. Le fonctionnement de ce mécanisme sera détaillé sur une page de l'Ensiwiki.

6.3 Travail demandé

Nous avons associé à chacun des modules un niveau de difficulté ainsi que des activités différentes :
— « D » pour conception (*Design*) : liberté de conception interne du module ;
— « C » pour compréhension : nécessite une bonne compréhension de la chaîne de décompression.

`dot_product`	☆		Obligatoires
`mode`	☆☆☆		
`floor`	☆☆☆	C	
`time_domain_transform`	☆☆☆		
`mapping`	☆☆☆☆☆	C	
`common_header`	☆		À la carte
`header1`	☆		
`header2`	☆☆		
`header3`	☆☆☆		
`vorbis_io`	☆☆☆☆	D	
`helpers`	☆☆		
`envelope`	☆☆☆☆	C	
`main`	☆☆☆		
`floor_type1`	☆☆☆☆☆	D/C	
`codebook`	☆☆☆☆☆	D	
`residue`	☆☆☆☆☆	D/C	
`vorbis_packet`	☆☆☆☆	C	
`vorbis_main`	☆☆☆	C	
`ogg_core`	☆☆☆☆	D++	
`ogg_packet`	☆☆☆☆	D++	
`pcm_handler`	☆☆☆		
`fast_imdct`	☆☆☆☆☆☆	C++	Extensions
`floor_type0`	☆☆☆☆☆☆	D/C	

Vous devez développer durant le projet au minimum les modules obligatoires, ainsi que l'équivalent de dix étoiles en modules à la carte. C'est la quantité de travail minimum attendue. La quantité de travail sera évaluée par le nombre d'étoiles obtenues sur l'ensemble des modules réalisés. Il va de soi que seuls les modules terminés et fonctionnels comptent ! Pour finir, ceux qui auront terminé l'ensemble des modules à la carte pourront s'attaquer aux modules d'extensions. Ces modules sont des modules super-bonus.

L'ordre de traitement des modules est libre : à vous de choisir selon votre niveau de compréhension, vos envies...

Bien que la quantité de travail soit un élément déterminant dans la notation, gardez à l'esprit qu'il y en a d'autres (qualité des modules, respect des consignes, cf section 1.3).

D'un point de vue pratique, pour vous aider à développer plus facilement votre décodeur, nous vous proposons différents outils et méthodes qui vous permettront de valider votre code. Ces outils et méthodes sont présentés sur le Wiki. Dans tous les cas, n'hésitez pas à faire appel aux encadrants si vous restez bloqués trop longtemps, ils pourront vous aider sur des problèmes courants ou moins courants, et ils pourront clarifier les points obscurs de la norme.

Il ne nous reste plus qu'à vous souhaiter bon courage !

Annexe A

Description du format WAV

Le format WAV stocke les échantillons de manière non compressée et est donc très simple. Il est constitué d'une partie donnant des informations sur l'audio (nombre et taille des échantillons, nombre de canaux, etc.) suivie des échantillons.

A.1 Structure principale

Le format WAV utilise le format de stockage *RIFF* qui utilise des *chunks* comme structure pour organiser un fichier. Un *chunk* est constitué d'un court en-tête et de son corps (voir Table A.1). Tous les champs dans ces *chunks* sont little-endian.

Nom	Taille (en octets)	Description
ID	4	4 caractères identifiants le *chunk*
Size	4	taille du corps du *chunk* (sans les champs *ID* et *Size*)
Data	*Size*	corps du *chunk* de taille *Size*

Table A.1 – Structure d'un chunk

Un fichier WAV est formé d'un *chunk* principal : *RIFF*.

Nom	Taille (en octets)	Description
ID	4	« `RIFF` »
Size	4	taille du fichier moins 8 octets
Type	4	les 4 caractères : « `WAVE` »
Format	24	un sous-*chunk* donnant les caractéristiquent de l'audio
Data	variable	un sous-*chunk* contenant tous les échantillons

Table A.2 – Chunk principal RIFF

A.2 Chunk décrivant les caractéristiques

Le *chunk* format donne les caractéristiques du fichier audio. Certains sont redondants, deux relations sont vérifiées entre les champs :

— `Data_rate = Slice_rate * Alignement`

— `Alignement = Channels * ceil(Depth/8)`

Le terme *Slice* est utilisé ici pour représenter le groupement des échantillons de chaque canal correspondant au même instant (la taille d'une *Slice* est donc égale à la taille d'un échantillon multiplié par le nombre de canaux).

Dans notre cas, les échantillons sont codés sur 16 bits donc `Depth` = 16 et aucune compression n'est utilisée donc `Compression` = 1.

Nom	Taille (en octets)	Description
ID	4	« `fmt␣` » (le quatrième caractère est un espace)
Size	4	16
Compression	2	indique la compression utilisée
Channels	2	nombre de canaux
Slice_rate	4	nombre d'échantillons par seconde par canal
Data_rate	4	octets par seconde
Alignement	2	octets par *Slice*
Depth	2	nombre de bits par échantillon

TABLE A.3 – Chunk de description des caractéristiques

A.3 Chunk donnant les échantillons

Le deuxième *chunk* contient tous les échantillons, ceux-ci sont mis les uns à la suite des autres dans l'ordre suivant : premier échantillon du premier canal, premier échantillon du deuxième canal, . . . , premier échantillon du dernier canal, deuxième échantillon du premier canal, deuxième échantillon du deuxième canal, etc.

Nom	Taille (en octets)	Description
ID	4	« `data` »
Size	variable	taille du fichier - 44 octets
Samples	variable	les échantillons

TABLE A.4 – Chunk contenant les échantillons