TIPE : Les mathématiques derrière la compression du son

Yoann Potiron (L2 Mathématiques)

15/06/2009

Table des matières

Introduction

Nous allons commencer par une petite introduction au son numérique.

Tout d'abord, qu'est-ce que le son (définition un peu brute) ?

Le son est une vibration de l'air, c'est-à-dire une suite de surpressions et de dépressions de l'air par rapport à une moyenne, qui est la pression atmosphérique. D'ailleurs, pour s'en convaincre, il suffit de placer un objet bruyant (un réveil par exemple) dans une cloche à vide pour s'apercevoir que l'objet initialement bruyant n'émet plus un seul son dès qu'il n'est plus entouré d'air !

La façon la plus simple de reproduire un son actuellement est de faire vibrer un objet. De cette façon un violon émet un son lorsque l'archet fait vibrer ses cordes, un piano émet une note lorsque l'on frappe une touche, car un marteau vient frapper une corde et la faire vibrer.

Et alors, c'est quoi le son numérique ?

D'une manière générale, on appelle signal analogique un signal produit par un dispositif mécanique ou électronique. Dans un tel signal, la variable est le temps qui s'écoule de manière continue. Il y a à peine quelques dizaines d'années, toute chaîne de production sonore était entièrement analogique : par exemple, le son produit par les musiciens, le signal électrique délivré par les micros, le signal transmis par ondes hertziennes ou gravé sur un disque de vinyle, le signal reçu et amplifié par votre chaîne Hi-Fi et finalement le son fourni par le haut-parleur, sont tous des signaux analogiques.

Avec la formidable augmentation de la puissance des ordinateurs est apparu un nouveau maillon dans cette chaîne : le son numérique. Une fois capté par le micro, le son est transformé en une suite de nombres binaires (formés de 0 et de 1), qui sont transmis, stockés ou gravés sous cette forme. Gràce à cette dernière, les sons musicaux peuvent être stockés sur votre ordinateur, transformés par les ingénieurs du son (ou par vous-même, quand vous voulez entendre les aigus ou alors mettre plus de basse), écoutés sur votre téléphone portable ou sur votre baladeur, échangés sur Internet... On comprend très vite pourquoi la musique est aujourd'hui davantage présente sous forme de signaux électriques numérisés, circulant dans des câbles et traités par des ordinateurs, que sous sa forme acoustique originelle.

A la frontière des mathématiques appliquées, de la physique (plus particulièrement de la psychoacoustique) et de l'informatique, ce TIPE va traiter de la numérisation et de la compression du son.

Dans un premier temps, nous allons parler de Joseph Fourier et de deux de ses découvertes qui portent son nom, les séries de Fourier et la transformée de Fourier.

Dans un second temps, nous allons nous intéresser aux deux opérations essentielles qu'a subi un signal numérique. Ce dernier a été *échantillonné* : cela consiste à prélever les valeurs $s_n = \mathrm{s}(t_n)$ du signal analogique à des instants régulièrement espacés $t_n = \tau n$, où τ est appelée la période d'échantillonnage. Par ailleurs, les échantillons s_n ont été *quantifiés* : cela consiste à approcher et remplacer ces nombres réels s_n, qui peuvent avoir une infinité de décimales instockables, par des nombres r_n pris dans un ensemble fini comportant $\mathrm{L} = 2^b$ valeurs possibles. Ces nombres r_n sont alors codés sur b bits pour être stockés ou transmis. En qualité audio, on utilise généralement un codage sur 16 bits, soit 2 octets.

Puis, nous allons voir deux types de compression, la compression *psychoacoustique*, qui consiste à utiliser les propriétés de l'ouïe et du cerveau pour réduire la quantité d'information, et la compression *entropique* ou *codage du Huffman*. C'est une compression de type statistique qui grâce à une méthode d'arbre permet de coder les octets revenant le plus fréquemment avec une séquence de bits beaucoup plus courte.

Enfin, nous allons terminer par l'invention algorithmique du siècle dernier, à savoir la FFT (*Fast Fourier Transform* ou *Transformée de Fourier rapide*), sans laquelle les deux premières étapes seraient inutiles. En effet, elle permet de passer d'une complexité de n^2 à $n * log(n)$, ce qui divise le nombre d'opérations par pratiquement 150 quand $n = 1000$ et qu'on a un log_2.

Chapitre 1

Transformée de Fourier

Commençons par parler des transformées de Fourier. En effet, elles vont nous être utiles tout au long de ce TIPE... Elles portent le nom de leur inventeur, Joseph Fourier. Il conduisait des expériences sur la propagation de la chaleur qui finalement, lui permettront de modéliser l'évolution de la température au travers de séries trigonométriques, et qui ouvriront la voie à la théorie des séries de Fourier et des transformées de Fourier.

Ce qui est exceptionnelle, c'est que 250 ans après, elles sont à l'origine de nombreuses choses autour de nous. Nous en reparlerons dans le Chapitre 4...

1.1 Séries de Fourier

En analyse, les séries de Fourier sont un outil fondamental dans l'étude des fonctions périodiques. C'est à partir de ce concept que s'est développée la branche des mathématiques connue sous le nom d'analyse harmonique. Si nous commençons par étudier les séries de Fourier, c'est parce que la transformée de Fourier peut être vue comme une généralisation de ces dernières pour des fonctions non périodiques.

L'étude d'une fonction périodique par les séries de Fourier comprend deux volets :

- l'analyse, qui consiste en la détermination de la suite de ses coefficients de Fourier ;
- la synthèse, qui permet de retrouver, en un certain sens, la fonction à l'aide de la suite de ses coefficients.

Au-delà du problème de la décomposition, la théorie des séries de Fourier établit une correspondance entre la fonction périodique et les coefficients de Fourier. **De ce fait, l'analyse de Fourier peut être considérée comme une nouvelle façon de décrire les fonctions périodiques**. Des opérations telles que la dérivation s'écrivent simplement en termes de coefficients de Fourier. La construction d'une fonction périodique solution d'une équation fonctionnelle peut se ramener à la construction des coefficients de Fourier correspondants.

Avant d'amener la définition des séries de Fourier, il faut définir ce qu'est une série trigonométrique (même si cela semble très intuitif)...

Définition . *On appelle* ***série trigonométrique*** *une série de fonctions* $\sum f_n$ *dont le terme général est de la forme* $f_n(x) = a_n cos(nx) + ib_n sin(nx)$ *avec* $x \in \mathbb{R}, \forall n \in \mathbb{N}, a_n \in \mathbb{R}, b_n \in \mathbb{R}$ *Sous forme complexe, on peut écrire*

$$\sum_{n\in\mathbb{Z}} c_n \exp inx$$

Nous sommes maintenant armés pour énoncer la définition de la série de Fourier. On pourrait le faire sous forme réelle, mais nous choisirons la forme complexe plus adaptée à notre sujet. Rappelons qu'il n'est pas nécessaire de se souvenir de la définition car nous utiliserons les transformées de Fourier. Cependant, elle est bien plus intuitive que cette dernière...

Définition . *Soit f une fonction $2-\pi$ periodique. La **série de Fourier de f** est la série trigonométrique définie par*

$$\sum_{n\in\mathbb{Z}} c_n \exp inx$$

où $\forall n \in \mathbb{Z}$, $c_n = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(x)\exp -inxdx$. Bien sûr, pour qu'elle existe, il faut que les c_n soient bien définis, sinon la définition ne marche pas !

Ce qui est particulièrement intéressant, c'est que sous certaines conditions (f continue et dérivable), la série de Fourier de f converge vers f(x). C'est le **théorème de Dirichlet-Jordan** (qui est en fait un poil plus général puisqu'il permet de ne pas avoir la continuité en un nombre fini de points, sous réserve d'être dérivable à gauche et à droite, mais qu'importe, nos signaux seront toujours continus !).

La démonstration est un peu longue et hors contexte pour être énoncée ici.

1.2 Transformée de Fourier

Comme nous le disions, la transformation de Fourier est un analogue de la théorie des séries de Fourier pour les fonctions non périodiques, et permet de leur associer un spectre en fréquences.

Il existe plusieurs définitions. Nous allons tout d'abord voir une définition plus mathématique, qui finalement ne nous servira à rien, puisque tout au long du TIPE, nous utiliserons une autre définition. Finalement, ce n'est qu'une histoire de convention, et cela ne change pas grand chose aux analyses que l'on va en faire.

Définition . *Si f est une fonction intégrable sur $\mathbb{R}$, sa **transformée de Fourier** est la fonction $\mathcal{F}(f) = \widehat{f}$ définie par*

$$\mathcal{F}(f) : \xi \mapsto \widehat{f} = \int_{-\infty}^{+\infty} \exp(-i\xi x)dx$$

Voici la définition alternative. Nous choisissons des autres noms de variables pour vous mettre en condition pour la suite du TIPE. Nous appelons maintenant la fonction s comme signal, au lieu de ξ, nous mettons f comme fréquence (attention à ne pas faire la confusion avec f une fonction, j'en ai déjà assez souffert moi-même !) et t comme temps à la place de x.

Définition . *Si s est une fonction intégrable sur $\mathbb{R}$, sa **transformée de Fourier** est la fonction $\mathcal{F}(s) = \widehat{s}$ définie par*

$$\mathcal{F}(s) : f \mapsto \widehat{s} = \int_{-\infty}^{+\infty} \exp(-2i\pi ft)dt$$

Bin sûr, il y a de quoi raconter sur les transformées de Fourier, mais étant donné que le cours est de niveau L3, nous ne dirons pas grand chose. Je pourrais faire semblant de maîtriser des objets et les énoncer les uns à la suite des autres, mais je ne pense pas que ce serait utile, surtout que je prends soin de simplifier au maximum (donc de ne pas utiliser ces objets) dans les autres chapitres. Néammoins, dans la version 2 de l'an prochain (si version 2 il y a !), je pense que je m'amuserai à compléter cette partie (histoire de garder la forme en Latex, faudrait pas perdre tout ce que je me suis donné tant de mal à apprendre !)... Il faut tout de même rajouter une chose, à savoir la transformée de Fourier inverse. En effet, si l'on va beaucoup se servir de la transformée de Fourier, on va avoir besoin de son inverse !

Définition . *Si la transformée de Fourier de f est elle-même une fonction intégrable, la formule dite de **transformation de Fourier inverse**, opération notée $\mathcal{F}^{-1}$, est celle qui permet (sous conditions appropriées) de retrouver f à partir des données fréquentielles :*

$$\mathcal{F}^{-1}(\widehat{s}) : f \mapsto s = \int_{-\infty}^{+\infty} s(x)\exp(2i\pi ft)dt$$

Nous l'avons énoncé sous la forme fréquentielle, le lecteur pourra sans difficulté l'énoncer sous la forme *mathématique*...

1.3 Transformée de Fourier Discrète

Sans rajouter de longs discours (le pourquoi du comment sera expliqué plus tard), il existe une version discrète de la Transformée de Fourier.

Définition . *Pour un signal s de N échantillons, la* ***transformée de Fourier Discrète*** *S est définie par :*

$$S(k) = \sum_{n=0}^{N-1} s(n) \cdot e^{-2i\pi k \frac{n}{N}}$$

Sans en rajouter plus, on a également la transformée de Fourier Discrète inverse :

Définition . ***la transformée de Fourier Discrète inverse*** *est donnée par :*

$$S(k) = \sum_{k=0}^{N-1} S(k) \cdot e^{2i\pi n \frac{k}{N}}$$

Chapitre 2

Echantillonage du son

Le système de conversion numérique du son a été mis au point en 1957 dans les laboratoires Bell par Max Mathews. L'échantillonnage numérique procède par prélèvements d'échantillons, c'est-à-dire de portions du signal sonore. A l'instar du cinéma, où une suite de photographies défilant à une vitesse déterminée produit l'illusion du mouvement, on mesure la pression acoustique du signal sonore à intervalles réguliers. La principale différence avec le cinéma est la cadence utilisée : si 24 images par seconde suffisent pour reproduire le mouvement, l'échantillonnage numérique d'un son requiert une cadence beaucoup plus élevée pour donner l'illusion d'un son continu. Nous allons voir pourquoi...

Définition . *On considère un son $s(t)$, où la fonction s est continue et bornée sur $\mathbb{R}$. Une periode d'échantillonage $\tau > 0$ ayant été choisie, le* ***son échantilloné*** *ou* ***échantillon*** *consiste en la suite des valeurs*

$$s_n = s(n\tau), \text{ où } n \in \mathbb{N}$$

La fréquence d'échantillonage F_e est définie par $F_e = 1/\tau$. Elle s'exprime en Hz.

La figure 2.1 représente un son de durée 0.01 seconde, échantilloné à 2000 Hz. A la quantification près (nous verrons cela plus loin), ce sont ces valeurs qui seront stockées sur le CD audio.

Une description plus élaborée du son échantillonné , abondamment utilisée en théorie du signal, consiste à le représenter sous une infinité d'impulsions de Dirac. Nous ne développerons pas plus ce point car le niveau recquis est bien trop élevé... Cependant, on peut retenir la petite idée qui se cache derrière ce nom. Au lieu d'enregistrer la valeur de l'échantillon $s_e(t)$, $\forall t$, il s'agit d'intégrer des sommes de petits triangles entre chaque t.

2.1 Critère de Nyquist et Théorème de Shannon

Comme vous allez le comprendre rapidement, nous faisons face à une première difficulté. Nous venons de voir que nous stockons les données du son $s_e(t)$. Cependant, si nous faisons varié $s(t)$ pour des $t \neq n\tau$, nous voyons bien que nous obtenons le même échantillon. L'objectif est donc de trouver une condition qui nous assure que l'échantillon suffit pour reconstruire le son de manière exacte et de manière unique. Prenons le cas d'un son pur. Enonçons donc la

Définition . *Un* ***son pur*** *est un son ne comportant qu'une seule harmonique de fréquence f. Autrement dit, c'est un signal sinuoïdal. En passant sous la forme complexe, cela donne :*

$$s(t) = \alpha \exp(2i\pi f t)$$

On peut effectuer sur ce son deux opérations de base qui ne changent pas la pureté de ce dernier :

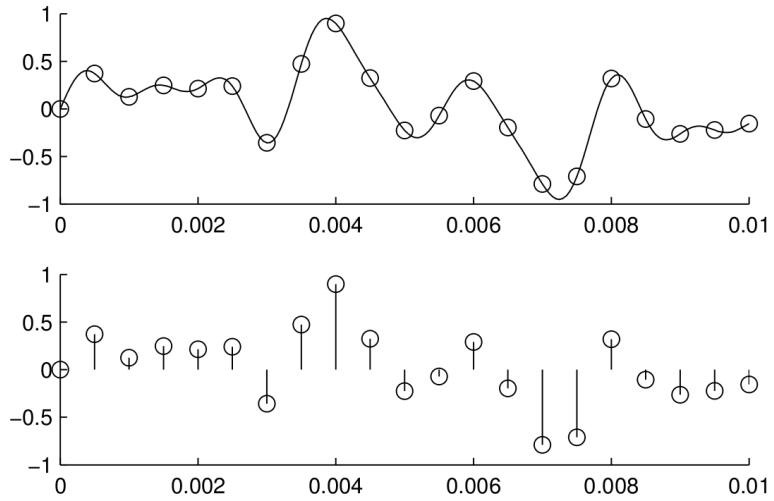

FIG. 2.1 – Son initial $s(t)$ en haut, son échantilloné $s_e(t)$ en bas

- ***L'amplification*** *d'un facteur* $a > 0$ $A := A(t) = as(t)$
- ***Le déphasage*** *d'angle* $\theta \in [0; 2\pi[$ $D := \alpha \exp(2i\pi ft - i\theta)$. *On peut aussi voir ce déphasage comme un retard* $\tau = \theta/2\pi f$

Soit donc le son pur :

$$s(t) = \alpha \cos(2\pi ft - \theta)$$

Cherchons la fréquence minimale d'échantillonage qui permette de reconstruire correctement ce son. Par exemple, on est tenté de se dire qu'on pourrait échantilloner à chaque fois que $s(t)$ passe par un maximum, donc une fois par période. Cependant, si l'on consdère le signal constant $s'(t) = \max s(t)$, on retrouverait les mêmes échantillons.

Essayons donc de prendre au moins un échantillon tous les maximums et les minimums, c'est-à-dire au moins deux échantillons par période. La période étant de $1/f$, on a donc l'hypothèse suivante :

$$\tau < \tfrac{1}{2f}$$

La fréquence d'échantillonage étant $F_e = \frac{1}{\tau}$, on peut donc énoncer le **critère de Nyquist** ou **condition de Shannon**

$$F_e > 2f$$

Avant de démontrer ce résultat, arretons-nous quelques instants pour réflechir à ce qu'il veut dire. On considère qu'une excellente oreille humaine peut percevoir les fréquences situées entre 20Hz et 20kHz, et que les sons deviennent inaudibles dehors (infra-sons ou ultra-sons). On sait que la plupart du son numérique (en tout cas de la musique, car c'est différent pour la radio et le téléphone...) est échantilloné à 44.1kHz (il suffit de regarder sur vos CDs favoris). Bien sûr, il fallait que la fréquence respecte le critère de Nyquist, donc il fallait qu'elle soit superieure à

$2*20kHz = 40kHz$. Cependant, pourquoi ne pas avoir pris simplement 40kHz ? Pour une question de conversion numérique, il faut une fréquence un peu superieure à 40kHz. Mais pourquoi 44.1kHz ?

Pour répondre à cette question, il faut revenir aux premiers jours de la recherche sur l'audio numérique. A l'époque, les disques durs avaient de la bande passante mais pas la capacité pour les longs enregistrements, on s'est donc intéressé aux enregistreurs vidéos. Ils étaient adaptés pour stocker des échantillons audios en créant une pseudo-onde vidéo qui convertissait les nombres binaires en noir et blanc. Il existait deux types de standard vidéo, le 525 lignes par image à 60 Hz (donc 60 images par seconde), et le 625 lignes par image à 50Hz.

Pour ces deux standards, une seconde de son peut être enregistrée *sur une seconde vidéo*. En effet, dans le standard 60Hz, il y a 35 lignes cachées (on ne peut pas enregistrer dessus). Ainsi il reste $525 - 35 = 490$ lignes par image. Il faut diviser par deux pour avoir le nombre de lignes par terrain, soit 245. De plus, dans chaque ligne, il y a trois échantillons.
$44100 = 60 * 245 * 3$

Dans le standard 50Hz, il y a 37 lignes cachés (ce qui nous donne $\frac{625-37}{2} = 294$ lignes enregistrables).
$44100 = 50 * 294 * 3$

Même si les CD n'ont pas de circuits vidéos, l'équipement utilisé pour faire des CD originaux est basé sur celui de la vidéo et determine donc la fréquence d'échantillonage.

Si l'on regarde bien, 44100 est un nombre assez exceptionnel (bien qu'il n'y paraisse pas à première vue), puisqu'il peut être factorisé comme le produit des carrés des 4 premiers nombres premiers ($44100 = 2^2 * 3^2 * 5^2 * 7^2$)), c'est donc pour cela qu'il a été choisi.

Attaquons-nous maintenant à la démonstration du critère. On va passer sous forme complexe, ainsi, le critère devient :

$$F_e > 2 \mid f \mid$$

Soient s_1 et s_2 deux signaux harmoniques tels que $s_1(t) = s_2(t) \forall t$ et tels que

$$s_1(t) = \alpha_1 \exp(2i\pi f_1 t)$$
$$s_2(t) = \alpha_2 \exp(2i\pi f_2 t)$$

Montrons qu'ils sont égaux. De $s_1(0) = s_2(0)$, on obtient $\alpha_1 = \alpha_2$ et on pose $\alpha = \alpha_1 = \alpha_2$. Si $\alpha = 0$, $\forall t, s_1(t) = s_2(t) = 0$. Supposons maintenant $t \neq 0$ et montrons l'égalité.
à $t = \tau$ on a $\alpha \exp(2i\pi f_1 t) = \alpha \exp(2i\pi f_2 t)$ d'où $\exp(2i\pi(f_1 - f_2)\tau) = 1$

$$\text{soit } f_1 - f_2 = \tfrac{k}{\tau} = kF_e, k \in \mathbb{Z}$$

Or, $\mid kF_e \mid = \mid f_1 - f_2 \mid <= \mid f_1 \mid + \mid f_2 \mid < \mid \frac{Fe}{2} \mid + \mid \frac{Fe}{2} \mid = F_e$, d'où $k = 0$, et donc $f_1 = f_2$. Sans le critère de Nyquist, on n'a pas l'égalité, car k peut être non nul. De plus si l'on observe les échantillons suivants, on a
$s_1(t_n) = \alpha \exp(2i\pi f_1 nt) = \alpha(\exp(2i\pi f_1 t))^n = \alpha(\exp(2i\pi f_2 t))^n = \alpha \exp(2i\pi f_2 nt) = s_2(t_n)$ ce qui fait qu'on a les mêmes échantillons pour des signaux différents.

Ainsi, le critère de Nyquist est nécessaire et suffisant pour que deux signaux harmoniques ayant les mêmes échantillons soient égaux. C'est un cas particulier du **Théorème de Shannon**. Avant de l'énoncer, il faut donner une définition.

Définition . *On dit qu'un signal $s(t)$ est à **bande limitée** $[-B, B]$ si*

$$\widehat{f} = 0 \ \forall \mid f \mid > B$$

autrement dit si le signal ne comporte aucune fréquence $\mid f \mid > B$.

Théorème de Shannon . *Soit s une fonction qui admette une transformée de Fourier $\widehat{s}$ à bande limitée $[-B, B]$. On échantillonne cette fonction à la fréquence F_e. Si F_e vérifie le critère de Nyquist, alors s est l'unique fonction à bande limitée $[-B, B]$ qui a pour échantillons les valeurs $s(\frac{n}{F_e})_{n \in \mathbb{N}}$*

Commençons par parler de démonstration... J'espère être un jour en âge de la comprendre, car ce théorème est un des plus étonnants que j'ai pu rencontré. Elle se sert de la Formule de Poisson, des sinus cardinaux, de transformées de Fourier... Revenons à l'analyse de ce théorème. Vous allez me dire que ce n'est pas vrai, que l'on peut changer les valeurs de $s(t)$ entre les échantillons sans modifier ces derniers. Justement, ce théorème montre que en faisant de telles modifications, il est impossible que s reste à bande limitée $[-B, B]$. Bien entendu, le théorème ne marchera pas, mais ce sera parce que les hypothèses ne sont pas respectées.

2.2 Quantification

Dans la partie précédente, nous étions confronter au problème qu'un son est continu et que par conséquent, il fallait stocker un nombre infini de valeurs, ce qui est impossible... Nous sommes parvenus à regler le problème en échantillonant le son et grâce au **Théorème de Shannon**, nous avons vu que nous perdions aucune information sur le son. Cependant, nous voilà confronter à un nouveau problème. Pour chaque s_n, les ordinateurs ne peuvent stocker qu'une quantité finie de valeurs. Nous savons pertinemment qu'à cause de ce problème, la reconstruction du son ne sera pas parfaite (un ordinateur ne sait même pas stocké parfaitement le nombre réel 1, alors vous pensez bien que pour un signal continu, c'est une autre histoire !). Nous devons donc nous faire à l'idée qu'il est impossible (on y croyait encore après la première partie) de reproduire le son original à 100%. Cherchons donc à minimiser cette perte.

2.2.1 Le procédé

Commençons par décrire le procédé le plus simple de quantification : la *quantification uniforme*. Soient N échantillons $s_n, n = 0...N-1$, que l'on souhaite coder sur b bits. Par exemple, si $b = 3$, on a 000, 001, 010, 011, 100, 101, 110 et 111 qui sont l'écriture en base 2 des nombres 1 à 7. Il y en a en tout $L = 2^b$. Dans le cas standard, où b = 16, cela donne $2^{16} = 65536$ possibilités. L'objectif est donc de fournir les meilleurs valeurs à chaque possibilité. Il faut déjà trouver le plus petit intervalle dans lequel on peut trouver tous les s_n. Comme $s_n, n = 0...N-1$ est fini, il suffit de prendre $M = max s_n$. Comme nous nous occupons d'un signal qui représente un son, on peut considérer que le minimum est égal à $-M$.

Maintenant, pour attribuer un nombre à chaque s_n, il y a trois étapes.

- On partage l'intervalle $[-M, M]$ en L sous-intervalles, $M_i = [-M + \frac{2M*i}{L},\ -M + \frac{2M*(i+1)}{L}[\cup [-M + \frac{2M*(L-1)}{L},\ -M + \frac{2M*((L-1)+1)}{L}], i = 0...L-2$
- On assigne à chaque s_n son code binaire b_n, qui est l'intervalle dans lequel il se trouve.
- On associe maintenant à chaque b_n le milieu de l'intervalle.

On se rend bien compte en regardant la figure 2 qu'il aurait été préférable de ne pas couper uniformément l'intervalle, mais de mettre plus de sous-intervalles près de 0. Cela s'appelle la *quantification non uniforme*, nous y reviendrons plus tard...

Le tableau ci-dessous résume les huits premiers échantillons du graphique précédent.

s_n	0.000	0.386	0.131	0.255	0.224	0.241	-0.373
r_n	0.125	2.375	0.125	0.375	0.125	0.125	-0.375
b_n	100	101	100	101	100	100	010

2.2.2 Erreur due à la quantification

Comme nous le disions, la reconstruction n'est pas parfaite. On peut la mesurer grâce à la :

Définition . *On appelle* ***erreur de quantification*** *ou* ***bruit de quantification****, pour chaque* s_n*, le nombre* $\epsilon_n = \mid s_n - r_n \mid$

Comme r_n représente le milieu de chaque sous-intervalle, si on considère h la longueur d'un sous-intervalle, on a par construction

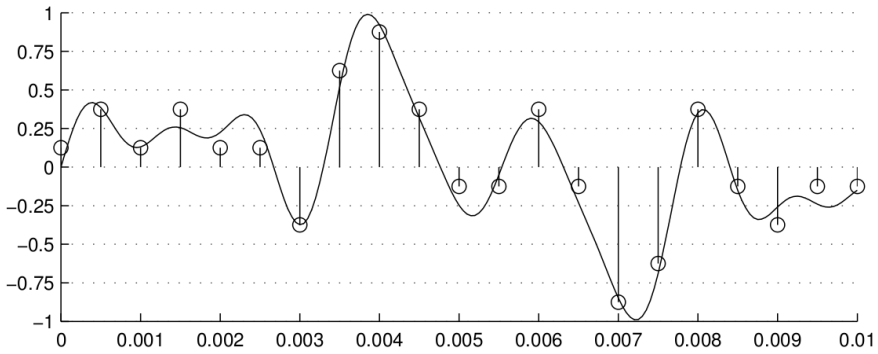

FIG. 2.2 – échantillonage et quantification sur 3 bits

$$\epsilon_n = \frac{h}{2} = \frac{2M}{L} * \frac{1}{2} = \frac{M}{2^b}$$

Sans rentrer dans l'explication théorique, en appliquant la formule sur l'intensité sonore, on trouve que d'ajouter un bit augmente de 6 dB le rapport entre l'intensité sonore du son et celle du bruit. Pour une qualité téléphonique, on utilise une quantification sur b = 12 bits, ce qui donne $6 * 12 = 72$ dB dans le rapport. Pour la musique, on utilise 16 bits, ce qui qui donne un rapport de $6 * 16 = 92$ dB.

Comme on vient de le dire, l'erreur de quantification (ou bruit de quantification) ne dépend pas de l'amplitude du signal d'entrée : elle est constante, ne dépendant que du degré de finesse de la quantification. D'un autre côté, le signal transmis aura une puissance à la réception directement proportionnelle à la puissance émise, c'est-à-dire, dans le cas du chant, à l'intensité de la voix du chanteur. Le rapport signal-sur-bruit, qui mesure la qualité du signal reçu par l'interlocuteur, correspond au rapport entre la puissance de l'erreur de quantification - constante - et la puissance du signal émis. Il est donc, du fait de ce qui précède, directement proportionnel à la puissance du signal émis. En d'autres termes, plus on chante fort (dans le cas d'une musique), moins on est perturbé. Par corollaire, plus on chante doucement, plus on sera perturbé par le bruit de quantification.

2.2.3 Quantification non uniforme

Pour remédier à l'inconvénient représenté par la quantification uniforme, on devrait imaginer un système qui rendrait le rapport signal-sur-bruit non dépendant de l'amplitude du signal émis (ou, si vous préférez, de l'intensité sonore du locuteur). Pour ce faire, il y a deux méthodes :

- Contrôler l'amplitude du signal émis (en amplifiant les faibles amplitudes relativement aux grandes amplitudes, on parle d'amplitude en terme d'intensité bien sûr, et non pas de fréquence, comme c'était le cas depuis le début), ce qui aura pour effet de favoriser les amplitudes faibles relativement aux grandes. Le bruit étant constant, on devrait ainsi être à même de rendre constant le rapport signal-sur-bruit
- Contrôler l'amplitude du signal émis (en amplifiant les faibles amplitudes relativement aux grandes amplitudes), ce qui aura pour effet de favoriser les amplitudes faibles relativement aux grandes. Le bruit étant constant, on devrait ainsi être à même de rendre constant le rapport signal-sur-bruit

2.2.4 Et le cerveau dans tout ça ?

Plus les années avancent et plus la quantité d'informations stockable sur un ordinateur est grande ! Supposons que l'on veuille enregistrer (et que soit disponible des capteurs et enregistreurs

imaginaires parfaits) un son le plus pur possible. Jusqu'où doit-on aller, sachant que le cerveau a une capacité limitée ?

Sans rentrer dans les détails d'anatomie du cerveau (bien que cela soit passionant, je ne crois pas que cela soit directement le sujet du TIPE), le débit d'une oreille est de 14Mbits/seconde. On a encore de la marge avec notre pauvre débit de $44100 * 16 = 0.7$ Mbits/seconde !

Chapitre 3

Compression

Une grosse étape vient d'être réalisée, on a réussi à stocker des *nombres infinis* dans des variables finies... Réfléchissons bien, combien cela va-t-il nous prendre de place ? Prenons les CD moyens d'aujourd'hui, à savoir 40 minutes de chanson. On veut coder l'information sur 4 octets, à 44100Hz, cela donne donc $44100 * 4 * 40 * 60 = 423.36Mo$. Cela fait un peu gros par rapport aux 50 ou 60 Mo habituels pour un album... Il est donc indispensable de compresser les chansons. On rentre dans une partie plus physique (qui va être formulée avec des mots plutôt qu'avec des formules brutes qui sortiraient un peu de nulle part pour nous mathématiciens) et informatique. Mais ne vous en faites pas trop, la dernière partie sur la FFT sera très mathématique !

Quels sont donc les deux grands types de compression ?

- La **compression psychoacoustique**, spécifique à la compression audio, qui s'appuie sur les caractéristiques de l'ouïe pour enlever de l'information non pertinente sur certaines fréquences
- la **compression entropique** (de données sans perte), type *zip* (Huffman), qui recherche des séquences afin de réduire la quantité de données qui doivent être stockées

3.1 Compression psychoacoustique

La **compression psychoacoustique** est la partie qui s'effectue avec des pertes et durant laquelle un encodeur va jeter de l'information pour réduire la taille. Elle est fondée sur un modèle mathématique qui tente de décrire ce que l'oreille humaine perçoit réellement ; elle a l'objectif de se débarasser de l'information qu'on ne peut pas entendre.

L'information exacte à éliminer dépend du codec (= COde - DECode en anglais) utilisé. Certains codecs sont faits pour enlever certaines fréquences afin que la compression soit meilleure pour les voix (la téléphonie fonctionne avec ce genre de système, c'est pourquoi les musiques d'attente ne sont jamais comme vous les entendez sur votre chaîne Hi-Fi).

Divers modèles ont étés formulés au fil des années afin de réduire la taille des fichiers audio. Le plus marquant ces dernières années est sans aucun doute le modèle psychoacoustique utilisé dans la compression mpeg1 layer 3 (mp3).

3.1.1 Motivation

Nous parlions de modèle mathématique. Dit en deux mots, cela consiste à passer dans le domaine fréquentiel, afin d'effectuer une quantification des *composantes de Fourier* sur un nombre de bits variable.

Avant de rentrer dans les détails, nous pouvons faire l'observation suivante, qui nous explique pourquoi nous devons passer dans le domaine fréquentiel et s'imposer une transformée de Fourier, puis une transformée inverse. Supposons qu'un son soit quantifié sur 16 bits ; nous avons vu dans le chapitre précédent que c'était la qualité standard. Imaginons que nous devions le compresser

car il prend trop de place ; il faut enlever un binaire, ce qui équivaut à diviser par deux. Ainsi, il ne reste que 8 bits. La modification va être trop importante et sur le signal reconstruit, il y aura un souffle largement perceptible.

Par contre, en passant sur le domaine des fréquences, autrement dit en quantifiant les coefficients de Fourier sur 8 bits, seule une petite modification des timbres des fréquences est perceptible (ou pas). Il est même possible d'aller jusqu'à 4 bits, alors qu'il ne serait même pas possible de reconnaître notre morceau favori quantifié sur 4 bits.

3.1.2 Les différentes étapes

Tout d'abord, observons les étapes nécessaires à la compression d'un fichier audio. Pour cet exemple, on décrira le Codec mp3 :

- Le signal est découpé en petites sections appellées **frames**, chacune étant entrelacées avec ses voisines.
- La section est analysée pour voir quelles fréquences sont présentes.
- Ces chiffres sont ensuite comparés à des tables de données propres au Codec, qui contiennent des informations sur les modèles psychoacoustiques. Dans le codec mp3, ces modèles sont très avancés, et une grosse partie de la modélisation se fonde sur un principe du nom de masking, que nous allons voir plus bas. Toute information qui correspond au modèle psychoacoustique est conservée et le reste est rejeté. **Ce sont les bases de la compression audio**.
- en fonction du **bitrate** (= nombre de bits par seconde), le codec utilise la taille allouée pour stocker ces données.

3.1.3 Le masking : le principe du codage psychoacoustique

Le **masquage simultané** fonctionne sur le principe que certains sons en couvrent d'autres lorsqu'ils sont joués en même temps. Dans des parties calmes d'une musique on peut parfois entendre des sons très faibles comme la respiration d'un chanteur. Dans un son plus fort, ces éléments ne peuvent plus être entendus, mais cela ne signifie pas que le son a disparu. Il a aussi été montré que si vous jouez deux sons se ressemblant, le deuxième étant à une fréquence un peu plus élevée, et avec un volume un peu plus faible, on a peu de chances d'entendre ce deuxième son. Le cerveau fait en quelque sorte son propre filtrage du son. Quand deux sons distincts sont joués simultanément, même si vous n'entendez pas l'un d'entre eux, il y a de l'information. C'est ce genre d'information qui est éliminée, et c'est le principe du masquage simultané - enlever les sons que le cerveau ne peut entendre à cause de la présence d'autres sons -.

Le **masquage temporel** fonctionne de manière similaire, mais ici l'idée n'est pas que vous ne pouvez pas entendre un son à cause d'un autre qui lui ressemble. C'est le fait que si nous jouons un son très peu de temps après un autre, nous n'entendons pas le deuxième (et vice-versa). Le cerveau ne peut pas tout analyser !

Bien que ces deux procédés aient l'air merveilleux, il ne faut pas oublier qu'il est impossible de faire de la magie. Alors oui, on peut réduire la taille sans trop s'en rendre compte. Mais cela s'entend, surtout par les musiciens ! Le tout est de trouver le bon équilibre.

3.1.4 Que se passe-t-il mathématiquement ?

Prenons donc une frame, comme ce qui était précisé dans les différentes étapes. Supposons que la découpe se fasse 512 échantillons par 512 échantillons. Appliquant la Transformée de Fourier Discrète, nous nous retrouvons avec 512 coefficients c_n. La partie significative pour l'audition est celle située au-dessus du masque M, représentée à gauche de la figure. Nous notons j les indices correspondants. La partie située en-dessous du masque, représentée à droite de la figure est à priori inaudible, car masquée (couverte) par la première. Nous notons k les indices correspondants.

La compression psychoacoustique consiste alors à quantifier les coefficients de la TFD, en utilisant moins de bits pour coder les c_k que les c_j. Par ailleurs, compte tenu de la formule qui

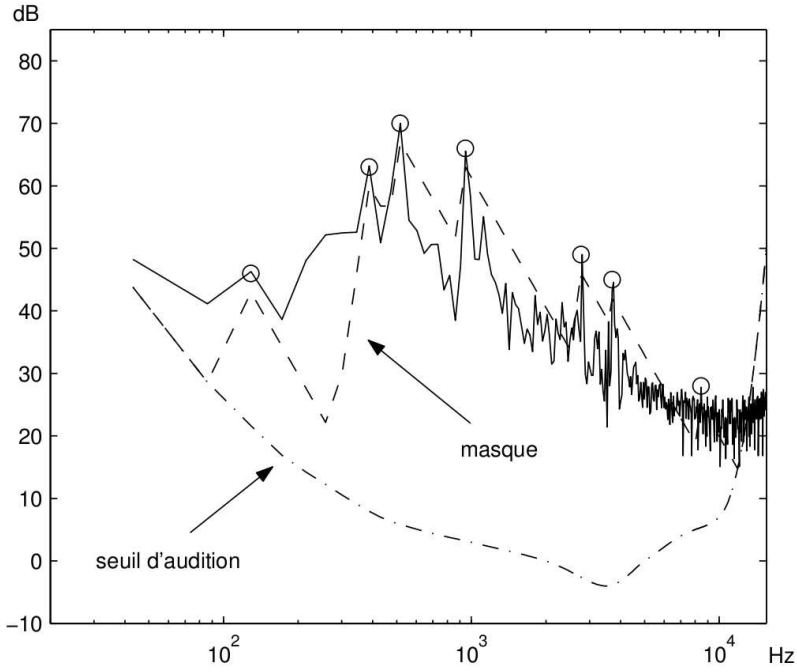

FIG. 3.1 – masque global du spectre (la TFD) des 512 échantillons u_n. Il prend en compte à la fois l'effet masquant des composantes fréquentielles de forte intensité et le seuil d'audition

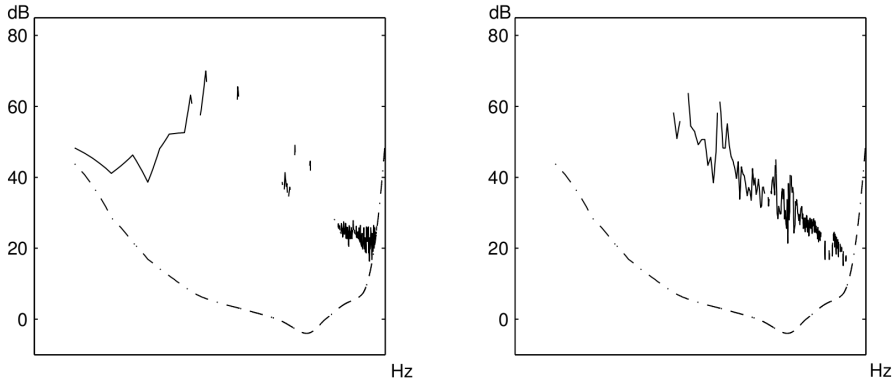

FIG. 3.2 – séparation des parties du spectre situées au-dessus (figure de gauche) et en-dessous (figure de droite) du masque, en vue d'une compression sur un nombre de bits différent

donne les c_n (par bonté d'âme, je rappelle la formule de la TFD, alors que vous devriez la connaître par coeur !)

$$c_n = \frac{1}{N} \sum_{k=0}^{N-1} r_k \exp(-2i\pi nk/N),$$

, les r_k étant les échantillons.

nous constatons que l'on a une symétrie conjuguée $c_{N-n} = \overline{c_n}$. Par conséquent, il suffit simplement de connaître la motié + 1 des coefficients $c_0 \dots c_{256}$. Ce sont donc ces 257 qui seront quantifiés puis stockés après la compression entropique de la partie suivante.

Lors de l'écoute, on recalcule l'ensemble des coefficients en utilisant leur TFD inverse (en vérité ce n'est pas la TFD mais chuut, on voit ça au chapitre suivant !), et on termine par l'addition des frames successives. Si nous n'avions rien quantifié, les coefficients seraient comme à l'entrée. En raffinant beaucoup le procédé, on arrive à des taux de 90% pour le mp3 tout en maintenant une excellente qualité sonore (vous la connaissez bien !).

3.1.5 Stocker les données : Comment cela fonctionne ?

Pour ceux qui veulent aller encore plus loin, nous terminons avec la façon de stocker, même si nous ne décrivons pas précisément le protocole. Vous pouvez sauter cette sous-partie si cela vous lasse de ne pas faire de mathématiques ! Depuis le Chapitre 2, vous savez ce que audio 16bit 44.1Khz signifie pour votre échantillon sonore ; mais que signifie 128kbps pour votre fichier compressé ?

Et bien, imaginez que vous avez seulement un certain nombre de bits pour décrire une seconde d'audio. Les données sont stockées en utilisant les mathématiques pour représenter le signal de chaque frame. En utilisant un modèle mathématique nommé Discrete Cosine Transform (DCT) (en français Transformée Cosinus Dicrète - TCD), une onde peut être représentée par une somme de fonctions cosinus. Plus on utilisera de fonctions cosinus dans l'expression mathématique, plus le résultat sera proche du signal original. On va stocker les données en fonction du bitrate, ainsi la complexité (donc l'exactitude de la chanson) sera limitée par la quantité de données qui peuvent être stockées par frame.

Plus simplement, 128kbps signifie qu'on a 128kbits de données disponibles chaque seconde pour décrire l'audio. C'est une quantité correcte pour un codec comme le mp3, et c'est un des bitrates les plus populaires pour l'encodage mp3. Comme pour la vidéo, la qualité de l'encodage audio augmentera si on alloue plus de kbits/seconde. Cependant, un mp3 à 192kbps est bien plus proche de la qualité CD que le 128kbps. Il est facile de comprendre que le bitrate est un facteur limitant dans le stockage des données.

Deux petites choses que l'on peut rajouter. On peut stocker l'audio à bitrate inconstant (cela se développe en ce moment), et ainsi descendre en kbps dans les parties peu complexes. Un dernier principe est le **Joint Stéréo**. Il s'agit de ne garder qu'un canal sur les deux dans les parties où ils sont vraiment similaires ! Cela devient donc du mono à certains endroits, mais ce n'est pas censé s'entendre. En tant que musicien, je ne crois pas du tout à une telle possibilité, je ne développe donc pas plus... La version un peu plus futée prend en compte le fait que nous avons beaucoup de mal à déterminer d'où viennent certaines fréquences. L'exemple le plus simple est de prendre son caisson de basse et de le bouger. On ne sent pas trop la différence et on a du mal à entendre que les fréquences viennent de ce caisson. On peut donc transformer les caissons de basse (et de hautes fréquences car cela marche pareil) en mono.

3.2 Compression entropique et codage de Huffmann

Comme dit précédemment, en plus d'une compression psychoacoustique, mp3 est doté d'une compression entropique. Cela ne sert donc à rien de le compresser en .zip par exemple, vous ne

gagnerez rien ! Interessons-nous au **codage de Huffman**.

C'est un algorithme de compression qui fut mis au point en 1952 par l'américain David Albert Huffman (un pionnier dans le domaine de l'informatique). C'est une compression de type statistique qui grâce à une méthode d'arbre que nous allons détailler plus loin permet de coder les octets revenant le plus fréquemment avec une séquence de bits beaucoup plus courte que d'ordinaire. Cet algorithme offre des taux de compression démontrés les meilleurs possibles pour un codage par symbole (on code des lettres ou des u_n dans le cas du son). Pour aller plus loin, il faut passer par des méthodes plus complexes réalisant une modélisation probabiliste de la source et tirant profit de cette redondance supplémentaire.

3.2.1 Principe

Le principe du codage de Huffman repose sur la création d'un arbre composé de noeuds. Supposons que la phrase à coder soit « montipeestcool ». On recherche tout d'abord le nombre d'occurrences de chaque caractère (ici les caractères 'm', 'n', 't', 'i' et 'p', 's', 't', 'c', 'l' sont représentés chacun une fois, le caractère 'e' deux fois et le 'o' 3 fois). Chaque caractère constitue une des feuilles de l'arbre à laquelle on associe un poids valant son nombre d'occurrences (le nombre de fois qu'il apparaît). Puis l'arbre est créé suivant un principe simple : on associe à chaque fois les deux noeuds de plus faibles poids pour donner un noeud dont le poids équivaut à la somme des poids de ses fils jusqu'à n'en avoir plus qu'un, la racine. On associe ensuite par exemple le code 0 à la branche de gauche et le code 1 à la branche de droite.

Pour obtenir le code binaire de chaque caractère, on remonte l'arbre à partir de la racine jusqu'aux feuilles en rajoutant à chaque fois au code un 0 ou un 1 selon la branche suivie. Il est en effet nécessaire de partir de la racine pour obtenir les codes binaires car lors de la décompression, partir des feuilles entraînerait une confusion lors du décodage. Ici, pour coder 'montipeestcool', nous obtenons le code :

((m (0 0 0)) (i (0 0 1 0)) (n (0 0 1 1)) (o (0 1)) (t (1 0 0)) (e (1 0 1)) (s (1 1 0 0)) (p (1 1 0 1)) (l (1 1 1 0)) (c (1 1 1 1)))

ce qui nous donne la phrase codée.

(0 0 0 0 1 0 0 1 1 1 0 0 0 0 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 0 1 0 0 1 1 1 1 0 1 0 1 1 1 1 0)

Dans ce petit exemple, on voit que la phrase codée prend beaucoup moins de place que si elle avait été écrite en caractère ASCII. Cependant, l'arbre de codage prend beaucoup plus de place (et oui, il faut bien se souvenir du code de chaque lettre...). Pour être gagnant, il faut des phrases beaucoup plus longues.

Il existe trois variantes de l'algorithme de Huffman, chacune d'elle définissant une méthode pour la création de l'arbre :

- **statique** : chaque octet a un code prédéfini par le logiciel. L'arbre n'a pas besoin d'être transmis, mais la compression ne peut s'effectuer que sur un seul type de fichier (exemple : un texte en français, où les fréquences d'apparition du 'e' sont énormes ; celui-ci aura donc un code très court, rappelant l'alphabet morse).
- **semi-adaptatif** : le fichier est d'abord lu, de manière à calculer les occurrences de chaque octet, puis l'arbre est construit à partir des poids de chaque octet. Cet arbre restera le même jusqu'à la fin de la compression. Il sera nécessaire pour la décompression de transmettre l'arbre (c'est le cas de notre exemple, et celui que l'on verra plus loin)
- **adaptatif** : c'est la méthode qui offre à priori les meilleurs taux de compression, car l'arbre est construit de manière dynamique au fur et à mesure de la compression du flux. Cette méthode représente cependant le gros désavantage de devoir modifier souvent l'arbre, ce qui implique un temps d'exécution plus long. Par contre la compression est toujours optimale et le fichier ne doit pas être connu avant de compresser. Il ne faut donc pas transmettre ou stocker la table des fréquences des symboles (c'est la cas le plus intéressant, mais le plus dur ; on l'utilise pour compresser la musique).

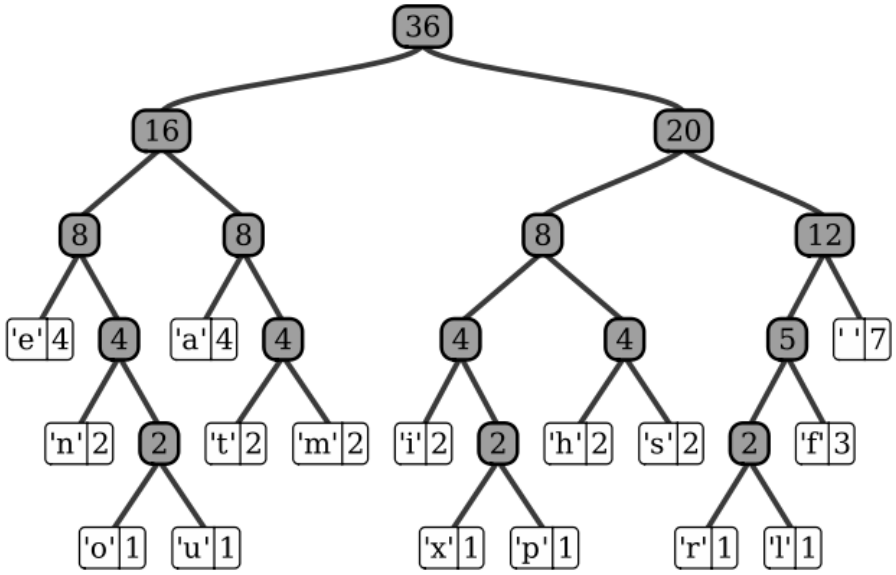

FIG. 3.3 – Un arbre d'exemple avec la phrase *this is an example of a huffman tree*

3.2.2 Algorithmes

Cette section pourrait être remplie de codes (en Scheme) qui permettent de réaliser le codage du Huffman semi-adaptatif d'une phrase. Au lieu de remplir 4 pages de codes que les lecteurs ne pourront sans doute pas comprendre (le Scheme, c'est quand même pas le langage le plus répandu), je vais mettre quelques exemples, pour que vous vous fassiez une idée (si vous tenez absolument à tester le code, envoyez-moi un message) !

phrase :
(a b r a c a d a b r a)
liste des frequences
((a 5) (r 2) (b 2) (d 1) (c 1))
arbre :
(11 (a 5) (6 (b 2) (4 (2 (d 1) (c 1)) (r 2))))
code :
((a (0)) (b (1 0)) (d (1 1 0 0)) (c (1 1 0 1)) (r (1 1 1)))
phrase codee :
(0 1 0 1 1 1 0 1 1 0 1 0 1 1 0 0 0 1 0 1 1 1 0)
decodage :
(a b r a c a d a b r a)

phrase :
(l a n a l y s e n u m e r i q u e c e s t b i e n)
liste des frequences
((n 3) (e 5) (i 2) (b 1) (t 1) (s 2) (c 1) (u 2) (q 1) (r 1) (m 1) (y 1) (l 2) (a 2))
arbre :
(25 (10 (5 (a 2) (3 (y 1) (2 (r 1) (m 1)))) (e 5)) (15 (7 (n 3) (4 (u 2) (l 2))) (8 (4 (i 2) (s 2)) (4 (2 (c 1) (q 1)) (2 (b 1) (t 1)))))))
code :
((a (0 0 0)) (y (0 0 1 0)) (r (0 0 1 1 0)) (m (0 0 1 1 1)) (e (0 1)) (n (1 0 0)) (u (1 0 1 0)) (l (1 0 1 1)) (i (1 1 0 0)) (s (1 1 0 1)) (c (1 1 1 0 0)) (q (1 1 1 0 1)) (b (1 1 1 1 0)) (t (1 1 1 1 1)))

phrase codee :
(1 0 1 1 0 0 0 1 0 0 0 0 0 1 0 1 1 0 0 1 0 1 1 0 1 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 1 0 0 1 1 0 1 1 0 0
1 1 1 0 1 1 0 1 0 0 1 1 1 1 0 0 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 0 0 0 1 1 0 0)
decodage :
(l a n a l y s e n u m e r i q u e c e s t b i e n)

3.2.3 Application à l'audio

Le format d'application du codage entropique à l'audio le plus connu est le format **FLAC** (Free Lossless Audio Codec). Il se démarque de ZIP et gzip car il a été créé spécifiquement pour compresser des données audio. La méthode ZIP réduit la taille d'un fichier audio de qualité CD de 20 à 40 %, alors que FLAC obtient des taux de 30 à 70 %. Par ailleurs, un fichier FLAC est composé de blocs successifs d'environ 100 ms de données, et peut être décompressé à la volée durant la lecture (quel que soit le niveau de compression), y compris par un système autonome équipé de peu de mémoire (platine CD FLAC, lecteur portable à disque dur ou mémoire Flash, lecteur autonome à disque dur pour chaîne Hi-Fi ou voiture...).

Ce sont bien sûr les coefficients de la transformée de Fourier qui vont être compressés. On commence par exemple par mettre pu_n, quand un coefficient est répété p fois de suite. Ensuite, on compte les occurences de la portion que l'on compresse (en général 512 coefficients), et nous procédons à un changement de dictionnaire (comme vu dans les parties précédentes). Si u_m apparaît le plus de fois, alors il aura le code binaire le plus petit en nombre de bits, et ainsi de suite...

Chapitre 4

Transformée de Fourier Rapide

On rentre dans la partie la plus intéressante et la plus compliquée de ce TIPE. Dites-vous bien que tout ce que vous avez fait avant ne servirait à rien si la Transformée de Fourier Rapide (FFT) n'existait pas. En effet, quand vous écoutez de la musique ou que vous regardez la HDTV (entre autres), comme nous l'avons vu dans les deux chapitres précédents, il y a Transformation de Fourier Discrète puis son inverse. Le problème est que cette dernière est de complexité $O(n^2)$ (elle demanderait $O(n^2)$ multiplications complexes pour le calcul de n coefficients complexes). James William Cooley et John Wilder Tukey, deux mathématiciens américains, ont trouvé un remarquable algorithme que certains définiront comme *l'invention en mathématiques appliquées du siècle* en 1965, même s'il semble qu'il avait été découvert par Danielson et Lanczos en 1942 et que Gauss le connaissait. Dans le cas le plus simple où n est une puissance de 2, l'algorithme de FFT requiert $O(N \log_2(N))$ opérations réelles (multiplications ou additions) pour aboutir au résultat.

Pour ceux qui ne font pas de l'informatique et qui ne sont pas habitués à manipuler des complexités, prenez 1000 et faites le calcul, la différence sera énorme. Prenez maintenant 1000000... Dans un cas, cela donne environ 14 millions, soit 4,6 millièmes de seconde de calculs sur un ordinateur standard. En revanche, dans la version naïve, cela donne 10^{12} opérations, soit plus de 5 minutes de calculs... Et je ne vous parle même pas du cas 1 000 000 000, où il faudrait une décennie de calculs...

Au risque d'insister, n'oubliez pas que c'est CET algorithme qui fait tout ! Il est à la base de nombreuses technologies dans notre société (on va en parler dans la partie d'introduction qui va suivre). Je suis allé mener une enquète au batiment des mathématiques de l'Université Lyon 1 pour voir si les mathématiciens connaissaient Cooley-Tuckey ou la Transformée de Fourier Rapide. La plupart connaisse le nom de FFT, mais ne saurait l'expliquer. Il a fallu frapper à une vingtaine de bureaux avant de trouver un chercheur connaissant les noms de Cooley et de Tuckey.

Même si certains enseignants mépriseraient presque les mathématiques (certes pas compliquées, puisque je les comprends) derrière cet algorithme, d'autres sont en admiration devant l'ingéniosité, l'utilité et la simplicité d'une telle invention ! Je me permets la comparaison avec un morceau de musique (cela tombe bien, c'est le sujet du TIPE) : ne vaut-il pas mieux trouver le bon tube à 4 accords que tout le monde adore (on peut par exemple penser aux Beatles, oui quand je dis tube, je parle pas des daubes qui passent maintenant sur nos ondes-radio) plutôt qu'une chanson bien complexe, mais qui sera que très peu écoutée ?

4.1 Introduction : où rencontrons-nous la FFT ?

L'algorithme FFT convient mieux à l'analyse des enregistrements audio numériques qu'au filtrage ou à la synthèse sonore. Elle permet par exemple d'obtenir l'équivalent logiciel d'un analyseur de spectre, que les ingénieurs utilisent pour tracer le graphe des fréquences contenues dans un signal électrique. Vous pouvez utiliser la FFT pour déterminer la fréquence d'une note dans une

musique enregistrée, pour essayer d'identifier des oiseaux ou des insectes... La FFT s'utilise aussi dans des domaines qui n'ont rien à voir avec le son, tel que le traitement d'image (avec une version bi-dimensionnelle de la FFT). La FFT a aussi des applications scientifiques ou statistiques, par exemple pour essayer de détecter des fluctuations périodiques dans les prix du marché, les populations animales... La FFT s'applique aussi à l'analyse des informations sismographiques, qui permettent de prendre des *sonogrammes* de l'intérieur de la Terre. Même l'analyse des séquences d'ADN utilise la transformée de Fourier ! Allez, je vous en donne une dernière : quand vous appuyez sur votre clé de voiture, ce sont des séries de Fourier qui sortent, et on utilise la FFT (à vérifier mais je suis à peu près sûr de ce que je dis !). J'espère qu'après tous les efforts que j'ai faits, vous êtes convaincus de l'importance de la FFT !

4.2 Retour sur la Transformation de Fourier discrète

Nous avions parlé dans le premier chapitre de TFD. Remettons une couche qui nous permettra de parler de la TFR.

Soit f une fonction continue, et périodique de période 1. La discrétisation par formule des rectangles à points équidistants s'écrit

$$U_k = \frac{1}{N} \sum_{j=0}^{N-1} f(\frac{j}{N}) \cdot e^{-2i\pi kj/N}$$

Remarquons que cette formule produit au plus N nombres complexes distincts ; en effet,

$$\begin{aligned} U_{k+N} &= \frac{1}{N} \sum_{j=0}^{N-1} f(\frac{j}{N}) \cdot e^{-2i\pi (k+N)j/N} \\ &= \frac{1}{N} \sum_{j=0}^{N-1} f(\frac{j}{N}) \cdot e^{-2i\pi kj/N} \\ &= U_k \end{aligned}$$

En posant $u_j = f(\frac{j}{N})\ \forall 0 \leqslant j \leqslant N-1$, on définit donc la transformation de Fourier discrète F_N comme l'opérateur linéaire qui associe à une suite finie $(u_j)_{0 \leqslant j \leqslant N-1}$ de nombres complexes la suite des U_k définie par

$$U_k = \frac{1}{N} \sum_{j=0}^{N-1} u_j \cdot e^{-2i\pi kj/N}$$

Parlons de la matrice de F_N. On la définit avec $(F_N)_{kj} = e^{-2i\pi kj/N}$. De cette définition, on tire que la matrice est symétrique (en effet $(F_N)_{jk} = (F_N)_{kj}$)

L'opérateur linéaire conjugué est noté $\overline{F}_N$ (on peut le voir comme la transformation inverse exposée dans le chapitre 1), et on a le résultat élémentaire et essentiel suivant, qui implique qu'à un facteur constant près, F_N est unitaire :

Lemme . *On a pour tout N les identités suivantes :*

$$F_N \circ \overline{F}_N = NI = \overline{F}_N \circ F_N$$

Démonstration. Soit

$$u_j = \sum_{k=0}^{N-1} U_k \cdot e^{2i\pi jk/N}$$

Calculons

$$\sum_{j=0}^{N-1} u_j \cdot e^{-2i\pi jl/N}$$

On a

$$\sum_{j=0}^{N-1} u_j \cdot e^{-2i\pi jl/N} = \sum_{j=0}^{N-1} e^{-2i\pi jl/N} \sum_{k=0}^{N-1} U_k \cdot e^{2i\pi jk/N} = \sum_{k=0}^{N-1} \sum_{j=0}^{N-1} e^{2i\pi j(k-l)/N} U_k$$

Or,

$$\sum_{j=0}^{N-1} e^{2i\pi j(k-l)/N} = \begin{cases} N, & si\ N\ divise\ k-l, \\ 0, & sinon. \end{cases}$$

Comme k et l varient entre 0 et $N-1$, N ne peut diviser $k-l$ que si $k=l$. Par conséquent,

$$\sum_{j=0}^{N-1} u_j \cdot e^{-2i\pi jl/N} = Nu_l$$

En conjugant la première égalité, on obtient la seconde égalité. □

Comme F_N est un opérateur linéaire de $\mathbb{C}^N$ dans lui-même, il faudrait à priori N^2 multiplications complexes pour calculer les U_k en fonction des u_j ; nous allons voir qu'il n'en est rien !

4.3 Principe de l'algorithme de transformation de Fourier rapide

Ecrivons explicitement la matrice de F_2 et celle de F_4.

$$F_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

$$F_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}$$

Calculons $(U_j)_{0 \leqslant j \leqslant 3}$

$$\begin{aligned} U_0 &= u_0 + u_1 + u_2 + u_3 \\ U_1 &= u_0 - iu_1 - u_2 + iu_3 \\ U_2 &= u_0 - u_1 + u_2 - u_3 \\ U_3 &= u_0 + iu_1 - u_2 - iu_3 \end{aligned}$$

Changeons l'ordre des u_j :

$$\begin{aligned} U_0 &= u_0 + u_2 + u_1 + u_3 \\ U_1 &= u_0 - u_2 - iu_1 + iu_3 \\ U_2 &= u_0 + u_2 - u_1 - u_3 \\ U_3 &= u_0 - u_2 + iu_1 - iu_3 \end{aligned}$$

On peut donc écrire :

$$\begin{aligned} \begin{pmatrix} U_0 \\ U_1 \end{pmatrix} &= F_2 \begin{pmatrix} U_0 \\ U_2 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix} F_2 \begin{pmatrix} U_1 \\ U_3 \end{pmatrix} \\ \begin{pmatrix} U_2 \\ U_3 \end{pmatrix} &= F_2 \begin{pmatrix} U_0 \\ U_2 \end{pmatrix} - \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix} F_2 \begin{pmatrix} U_1 \\ U_3 \end{pmatrix} \end{aligned}$$

Il est clair que le calcul de F_4 demande le calcul de deux transformations F_2 et d'une multiplication par une matrice diagonale de taille 2*2.

On peut également réécrire les équations en échangeant l'ordre des lignes :

$$\begin{aligned} U_0 &= u_0 + u_1 + u_2 + u_3 \\ U_2 &= u_0 - u_1 + u_2 - u_3 \\ U_1 &= u_0 - iu_1 - u_2 + iu_3 \\ U_3 &= u_0 + iu_1 - u_2 - iu_3 \end{aligned}$$

Dans ce cas, on a :

$$\begin{aligned} \begin{pmatrix} U_0 \\ U_2 \end{pmatrix} &= F_2 \begin{pmatrix} u_0 + u_2 \\ u_1 + u_3 \end{pmatrix} \\ \begin{pmatrix} U_1 \\ U_3 \end{pmatrix} &= F_2 \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \begin{pmatrix} u_0 - u_2 \\ u_1 - u_3 \end{pmatrix} \end{aligned}$$

Dans ce cas, on fait d'abord la multiplication par la matrice diagonale 2*2, et ensuite les deux transformations F_2.

Les deux idées ci-dessus se généralisent au cas $N = 2^n$. Dans le premier cas, on dit qu'on a un algotithme avec entrelacement temporel (et oui n'oublions pas que les u_j sont nos échantillons de sons), et dans le deuxième cas, entrelacement fréquentiel (pour rappel la transformée de Fourier fait passer à la *fréquence*).

4.4 Algorithme de TFR : entrelacement fréquentiel

Nous allons nous intéresser à l'algorithme de transformation de Fourier dans le deuxième cas, c'est-à-dire l'entrelacement fréquentiel (le cas de l'entrelacement temporel étant tout à fait analogue).

4.4.1 Séparation entre les modes pairs et les modes impairs

On réécrit F_N en regroupant d'abord les modes pairs, puis tout les modes impairs ; on pose $M = \frac{N}{2}$, on a donc :

$$U_{2k} = \sum_{j=0}^{N-1} u_j \cdot e^{-4i\pi kj/N} = \sum_{j=0}^{M-1} u_j \cdot e^{-2i\pi kj/M} + \sum_{j=M}^{N-1} u_j \cdot e^{-2i\pi kj/M}$$

En effet, on a simplement séparé la somme en deux, et remplacé N par $2M$.
On remarque que,

$$\forall j \in \{M, ..., N-1\}, e^{-2i\pi kj/M} = e^{-2i\pi k(j-M)/M} \tag{4.1}$$

On a donc :

$$\begin{pmatrix} U_0 \\ U_2 \\ \vdots \\ U_{N-2} \end{pmatrix} = F_M \begin{pmatrix} u_0 + u_M \\ u_1 + u_{M+1} \\ \vdots \\ u_{M-1} + u_{N-1} \end{pmatrix}$$

De la même façon pour les impairs, on a :

$$\begin{aligned} U_{2k+1} &= \sum_{j=0}^{N-1} u_j \cdot e^{-2i\pi(2k+1)j/N} \\ &= \sum_{j=0}^{M-1} u_j \cdot e^{-2i\pi kj/M} \cdot e^{-2i\pi j/N} + \sum_{j=M}^{N-1} u_j \cdot e^{-2i\pi kj/M} \cdot e^{-2i\pi j/N} \\ &= \sum_{j=0}^{M-1} u_j \cdot e^{-2i\pi kj/M} \cdot e^{-i\pi j/M} + \sum_{j=M}^{N-1} u_j \cdot e^{-2i\pi kj/M} \cdot e^{-i\pi j/M} \end{aligned}$$

En utilisant 4.4.1 et la relation :

$$e^{-i\pi j/M} = -e^{-i\pi(j-M)/M}$$

on obtient :

$$\begin{pmatrix} U_1 \\ U_3 \\ \vdots \\ U_{N-1} \end{pmatrix} = F_M \begin{pmatrix} u_0 - u_M \\ e^{-i\pi/M} \cdot (u_1 - u_{M+1}) \\ \vdots \\ e^{-i\pi(M-1)/M} \cdot (u_{M-1} - u_{N-1}) \end{pmatrix}$$

Notons

$$u_I = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{M-1} \end{pmatrix}, u_{II} = \begin{pmatrix} u_M \\ u_{M+1} \\ \vdots \\ u_{N-1} \end{pmatrix}$$

et

$$U_{pair} = \begin{pmatrix} U_0 \\ U_2 \\ \vdots \\ U_{N-2} \end{pmatrix}, U_{impair} = \begin{pmatrix} U_1 \\ U_3 \\ \vdots \\ U_{N-1} \end{pmatrix}$$

On définit P_M la matrice diagonale avec :

$$(P_M)_{jj} = e^{-i\pi j/M}$$

On a donc la factorisation par blocs :

$$\begin{pmatrix} U_{pair} \\ U_{impair} \end{pmatrix} = \begin{pmatrix} F_M & 0 \\ 0 & F_M \end{pmatrix} \begin{pmatrix} I_M & 0 \\ 0 & P_M \end{pmatrix} \begin{pmatrix} u_I + u_{II} \\ u_I - u_{II} \end{pmatrix} \tag{4.2}$$

La figure 4.1 résume ce qu'il se passe pour $N = 2^3 = 8$. Les croisillons, ou *papillon* réprésentent l'endomorphisme de $\mathbb{C}^2$:

$$\begin{pmatrix} u \\ v \end{pmatrix} \mapsto \begin{pmatrix} u + v \\ u - v \end{pmatrix}$$

La flèche horizontale simple symbolise un transfert de données, et la flèche surmontée d'un nombre complexe symbolise la multiplication par ce nombre complexe.

4.4.2 Permutations

Comme on le voit sur la figure 4.1, les U_k n'arrivent pas dans le bon ordre. Il va falloir les permuter. A chaque étape (a chaque transformée de Fourier récursive), il faut faire la transformation :

$$\begin{pmatrix} U_{pair} \\ U_{impair} \end{pmatrix} \mapsto \begin{pmatrix} U_I \\ U_{II} \end{pmatrix}$$

Soit $U_j,\ 0 \leqslant j \leqslant 2^n - 1$ un mode. Essayons de retrouver sa place (autrement dit j). Notons σ la permutation à réaliser. On asocie à j sa représentation en numération binaire

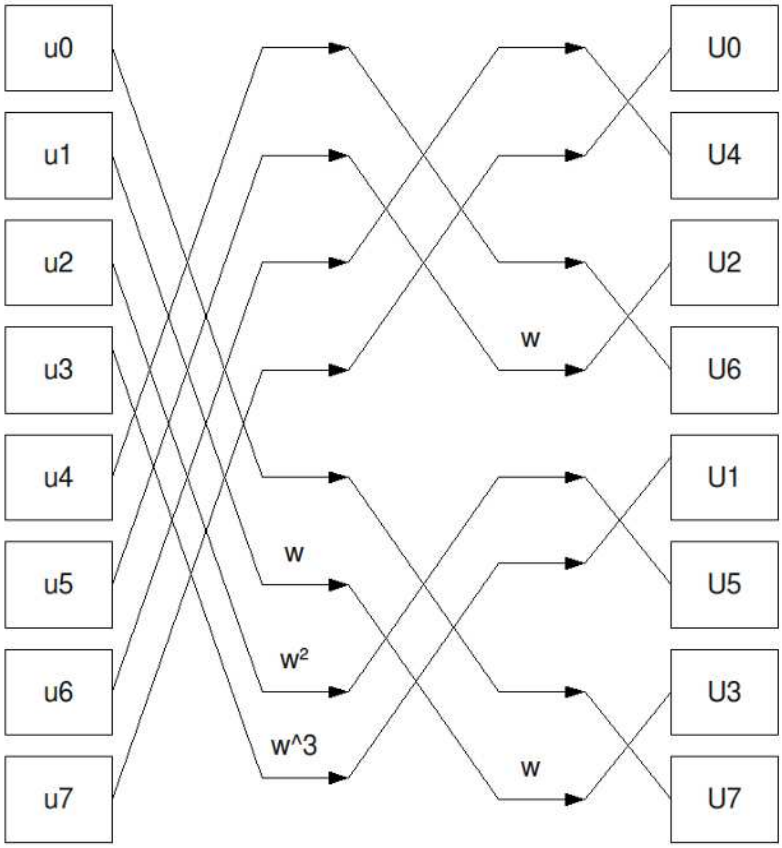

FIG. 4.1 – Les papillons de la TFR

$$j = \sum_{k=0}^{n-1} 2^k d_k = \overline{d_{n-1} d_{n-2} \dots d_1 d_0}$$

.

On vérifie alors que

$$\sigma_n(\overline{d_{n-1} d_{n-2} \dots d_1 d_0}) = \overline{d_{n-2} d_{n-3} \dots d_0 d_{n-1}}$$

En effet, voyons comment on passe de la suite

$$0, 1, \dots, 2^n - 1$$

à la suite

$$0, 2, \dots, 2p, \dots, 2^n - 2, 1, 3, \dots, 2p + 1, \dots, 2^n - 1.$$

Autrement dit, on va chercher σ_n^{-1} (une permutation est toujours bijective donc nous pouvons le faire).

Nous allons traiter 2 cas.

Si j est pair, on a

$$j = \sum_{k=0}^{n-1} 2^k d_k$$

avec $d_0 = 0$. Associons-lui $j/2$.

$$\frac{j}{2} = \sum_{k=1}^{n-1} 2^{k-1} d_k = \sum_{k=0}^{n-2} 2^k d_{k+1} = \overline{d_0 d_{n-1} \dots d_1}$$

Si j est impair, on a

$$j = \sum_{k=0}^{n-1} 2^k d_k$$

avec $d_0 = 1$. Associons-lui $2^{n-1} + \frac{j-1}{2}$. On a

$$\frac{j-1}{2} = \sum_{k=0}^{n-2} 2^k d_{k+1}$$

,

d'où

$$\frac{j-1}{2} + 2^{n-1} = 2^{n-1} + \sum_{k=0}^{n-2} d_{k+1} = \overline{d_0 d_{n-1} \dots d_1}$$

On a bien

$$\sigma_n^{-1}(\overline{d_{n-1} d_{n-2} \dots d_1 d_0}) = \overline{d_0 d_{n-1} \dots d_1} \Leftrightarrow \sigma_n(\overline{d_{n-1} d_{n-2} \dots d_1 d_0}) = \overline{d_{n-2} d_{n-3} \dots d_0 d_{n-1}}$$

Reprenons la figure 4.1, pour appliquer ce procédé à ce qu'on avait à la fin de la partie précédente. Il suffit d'appliquer $N - 1$ fois, donc dans notre cas 2 fois (la première fois sur les 8, la deuxième fois sur $\frac{8}{2} = 4$. On pourrait l'appliquer une troisième fois sur deux éléments mais on voit bien que cela reviendrait à appliquer l'identité. La figure 4.2 présente les permutations à faire (on peut réaliser d'abord la deuxième permutation puis la première, car elles sont commutatives).

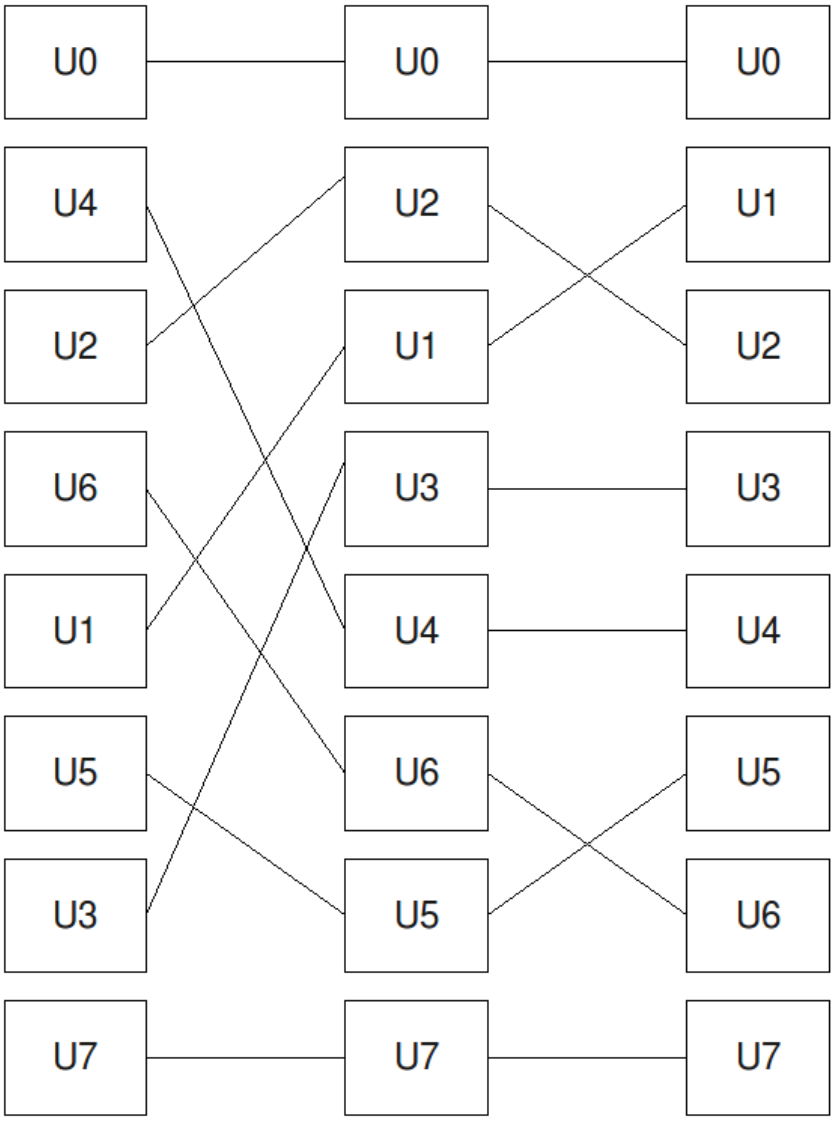

FIG. 4.2 – Les permutations de la TFR

4.4.3 Nombre d'opérations

Compte d'opérations . *Soit $N = 2^n$. Une TFR sur n nombres complexes demande un nombre d'opérations réelles (additions ou multiplications) au plus égal à $5N \log_2 N$*

Démonstration. Nous utilisons la convention consistant à compter les multiplications et les additions réelles. L'addition de deux nombres complexes demande 2 additions réelles, et la multiplication de deux nombres complexes demande deux additions réelles et quatre multiplications réelles. Soit a_n le nombre d'opérations réelles nécessaires pour un algorithme de TFR sur $N = 2^n$ modes ; on pose $M = N/2$.

D'après 4.4.1, pour obtenir $u_I + u_{II}$ et $u_I - u_{II}$, il nous faut $2M$ additions complexes, soit $2N$ opérations réelles.

La multiplication de $u_I - u_{II}$ par P_M demande M multiplications complexes, soit $6M = 3N$ opérations réelles.

Enfin, pour faire les deux transformations de Fourier rapide de dimension M, il nous faut $2a_{n-1}$ opérations réelles. On a donc l'inégalité

$$a_n \leqslant 2a_{n-1} + 5 \times 2^n$$

Par récurrence, et en observant que F_1 ne demande aucune opération, on a

$$a_n \leqslant 5n2^n$$

que nous réécrivons en

$$a_n \leqslant 5N \log_2 n$$

□

4.5 Un exemple d'utilisation

4.5.1 Approche mathématique

Prenons un exemple standard qui est celui des polynômes. On vous laisse le suspense sur ce que l'on va faire pour l'instant, mais c'est assez puissant !

Soit $P \in \mathbb{R}_n[X]$, il existe plusieurs façons de le représenter.

- On peut le représenter par la liste ordonnée de ses coefficients.
- On peut le représenter par sa valeur en n points distincts

Pour fixer les esprits, on se donne le polynôme $P(X) = 3 + X + 4X^2 + X^3$.

Dans le premier cas, cela donne $(3, 1, 4, 1)$. Pour le second cas, faisons le choix $(P(0),\ P(1),\ P(2),\ P(3))$, on a alors $(3, 9, 29, 69)$. A noter que le polynôme est bien identifié uniquement dans le second cas (il faut bien que les points soient distincts et d'un nombre égal à son degré !). La démonstration ne sera pas énoncée ici car un peu hors-sujet (je l'ai eu en kholle d'Algèbre IV pour la suite énoncée !).

Bien sûr, vous vous doutez bien que les ordinateurs préfèrent travailler avec les racines n-ièmes de l'unité (ce serait trop beau autrement).

Définition . *La transformée de Fourier du polynôme de degré n décrit par la liste ordonnée de ses coefficients, est le même polynôme décrit par la liste de ses valeurs pour les n racines nièmes de 1.*

Ecrivons explicitement cette transformée. Soit $N = 2^n$, $n \in \mathbb{N}$, $\forall 0 \leqslant j \leqslant N-1$, on a les racines n-ièmes de l'unité

$$\alpha_j = e^{2\pi ij/N}$$

La transformation de Fourier du polynôme $P = (x_0, x_1, \ldots, x_N)$ est donnée par la liste $(P(\alpha_0), P(\alpha_1), \ldots, P(\alpha_{N-1})$, telle que :

$$P(\alpha_j) = \sum_{k=0}^{N-1} x_k(\alpha_j)^k = \sum_{k=0}^{N-1} e^{2i\pi jk/N}$$

qui ressemble fortement à notre transformée de Fourier rapide de la précédente partie !

Ce n'est pas tout à fait pareil mais l'esprit est le même. Sans le démontrer (c'est analogue à ce que nous avons déjà fait), nous avons cette fois-ci une autre relation. Soit P_N la transformation de Fourier de N éléments, et posons $M = N/2$. Soit

$$x_{pair} = \begin{pmatrix} x_0 \\ x_2 \\ \vdots \\ x_{N-2} \end{pmatrix}.$$

De même, on a x_{impair}. Enfin, on a α la matrice diagonale des α_j.

$$P_N = \begin{pmatrix} P_{N/2} & 0 \\ 0 & P_{N/2} \end{pmatrix} \begin{pmatrix} x_{pair} \\ x_{pair} \end{pmatrix} + \alpha \begin{pmatrix} P_{N/2} & 0 \\ 0 & P_{N/2} \end{pmatrix} \begin{pmatrix} x_{impair} \\ x_{impair} \end{pmatrix}$$

4.5.2 Algorithmes

Regardons ce que cela donne en C++. On suppose que l'on a défini le module *complex* (la structure *complex* est formée de deux long double, la partie réelle et la partie imaginaire). On utilise les bibliothèques nécessaires *iostream.h* et *math.h*. La meilleure façon serait de la programmer en version itérative, car la version récursive est très couteuse en place. Néammoins, nous ne ferons que la version récursive...

```
void fft ( int n, complex * depart, complex * resultat)
// n est une  puissance de 2, depart et resultat deux tableaux de complex de taille n
{       complex * pair, *impair, *resultat_fft_pair, *resultat_fft_impair;
    int i, i_modulo;
    long double alpha_i, alpha;
    complex tau;
    initialisation_complex(tau)
    if(n == 1)
    {
        resultat[0] = depart[0]; // on suppose que cela marche entre deux structures
    }
    else
    {
        pair = new complex[n/2];
        impair = new complex[n/2];
                resultat_fft_pair = new complex[n/2];
                resultat_fft_impair = new complex[n/2];
                for(i = 0; i<n/2; i++)
                {
                        pair[i] = depart[2*i];
                        impair[i] = depart[2*i+1];
                }
                fft( n/2, pair, resultat_fft_pair);
                fft( n/2, impair, resultat_fft_impair);
                alpha = 2*pi/n;
                for(i= 0; i <= n-1; i++)
```

```
                {
                        alpha_i = alpha*i;
                        i_modulo = i % ( n/2 );
                        rentrer_donnees_complex (tau, cos(alpha_i), sin(alpha_i));
                        resultat[i] = somme_complex(resultat_fft_pair[i_modulo]
                        + produit_complex(tau * resultat_fft_impair[i_modulo]));
                }
                delete pair;
                delete impair;
                delete resultat_fft_pair;
                delete resultat_fft_impair;
        }
}
```

L'algorithme de calcul de la transformée de Fourier inverse se déduit de l'algorithme de calcul de la transformée de Fourier en changeant uniquement le signe de calcul du sinus. De plus il faut prendre soin, lorsque le calcul est terminé, de diviser tous les éléments du tableau resultat par n (nous avions vu cela dans un lemme précédent).

```
void fft_inverse ( int    n , complex * depart, complex * resultat)
// n est une   puissance de 2
{
   complex * pair, *impair, *resultat_fft_pair, *resultat_fft_impair;
        int i, i1;
        int i, i_modulo;
    long double alpha_i, alpha;
    complex tau;
    initialisation_complex(tau);
        if (n == 1)
    {
        resultat[0] = depart[0]; // on suppose que cela marche entre deux structures
    }
        else
        {
                pair = new complex[n/2];
        impair = new complex[n/2];
                resultat_fft_pair = new complex[n/2];
                resultat_fft_impair = new complex[n/2];
                for(i = 0; i<n/2; i++)
                {
                        pair[i] = depart[2*i];
                        impair[i] = depart[2*i+1];
                }
                fft_inverse ( n/2, pair, resultat_fft_pair);
                fft_inverse ( n/2, impair, resultat_fft_impair);
                alpha = 2*pi/n;
                for(i= 0; i <= n-1; i++)
                {
                        alpha_i = alpha*i;
                        i_modulo = i % ( n/2 );
                        rentrer_donnees_complex (tau, cos(alpha_i), - sin(alpha_i));
                        resultat[i] = somme_complex(resultat_fft_pair[i_modulo]
                        + produit_complex(tau * resultat_fft_impair[i_modulo]));
                }
                delete pair;
```

```
                delete impair;
                delete resultat_fft_pair;
                delete resultat_fft_impair;
        }
}

void fft_inverse ( int    n , complex * depart, complex * resultat)
{
                int i;
                fft_inverse (n, depart, resultat);
                for(i=0;i<n;++i)
                {
            resultat[i] = resultat[i]/n;
                }
}
```

4.5.3 Applications

Je vous vois venir. Vous allez me dire, *ben oui c'est beau votre truc, mais on fait quoi avec ça.* Je vous parlais plus haut de suspense. Et bien, par exemple, on peut multiplier des polynômes entre eux. En effet $\forall i\ P(i)Q(i)$ ne nécessite qu'une opération. Ainsi, on fait la transformée sur les deux polynômes que l'on veut multiplier, on multiplie les modes entre eux, puis on fait une transformée inverse. La complexité de la multiplication des modes est négligeable par rapport aux deux transformées, et on a donc une complexité en $n \log n$ au lieu du bon vieux n^2...

On peut également appliquer le processus aux multiplications de nombres réels.

Conclusion

Nous avons pu voir sans trop rentrer dans les détails les méthodes actuelles qui permettent la compression du son au format numérique. Nous avons également vu comment à partir de celle-ci, il est possible de réécouter la musique. Même s'il est probable que les prochaines découvertes nous permettront d'améliorer et d'optimiser ce processus, nous pouvons considérer que nous avons déjà réalisé beaucoup de choses dans ce domaine.

Cependant, un domaine de recherche en plein essor est celui de l'indexation des bases de données musicales. Pour naviguer dans une énorme base de données de ce type, nous pouvons en effet utiliser d'autres paramètres que les informations textuelles classiques sur un fichier sonore (nom de l'album, de l'artiste, de la piste...) : navigation par sonorité (type d'instruments), par rythme (le métal n'a pas le même rythme que le classique par exemple), génération automatique de liste de lecture... De nombreux algorithmes sont apparus récemment pour extraire automatiquement ce type d'informations avec une précision souvent étonnante. Ils effectuent des analyses parfois d'une grande complexité, qui combinent des critères temporels pour l'analyse du rythme et des critères fréquentiels pour l'analyse du timbre et de l'harmonie.

L'extraction de ces informations pour une très grande base de données est souvent très coûteuse en puissance de calcul. D'où l'idée de Laurent DAUDET, maître de conférences à l'Université de Pierre et Marie Curie, et de son équipe LAM de l'Institut Jean le Rond d'Alembert : s'ils disposent d'une base de données musicales sous forme compressée à l'aide de leur algorithme parcimonieux (nous avons survolé ces idées dans le Chapitre 3, il s'agit en gros de bien choisir les intervalles de travail du morceau), pourquoi décompresser les sons pour faire ces analyses ? Ne pourraient-ils pas exploiter le fait que les représentations parcimonieuses fournissent directement un jeu compact pour extraire cette information ? Ils ont récemment montré que c'est possible. Ainsi, à très faible coût calculatoire, nous pouvons obtenir les informations de rythme et d'harmonie avec une précision comparable à celle des algorithmes dédiés. Ce type d'étude ouvre l'ère de la *compression intelligente* des sons.

Les représentations numériques des sons musicaux devront dans l'avenir être capables non seulement de stocker et transmettre les sons de manière efficace, mais aussi de rendre possibles l'analyse et la manipulation de leurs structures constitutives. A cet égard, les représentations parcimonieuses offrent une approche crédible, car leur complexité calculatiore, bien qu'élevée, est maintenant à la portée de la plupart des ordinateurs.