**Mémoire de l'ENS Louis-Lumière**

**Section Son**

# La sommation numérique


## Bastien Prevosto


| | | |
|---|---|---|
| Directeur de Mémoire Externe | : | Gaël Martinet |
| Directeur de Mémoire Interne | : | Laurent Millot |
| Rapporteur | : | Jean Rouchouse |

Session de juin 2012

Je tiens à remercier chaleureusement mes directeurs de mémoire, Laurent Millot et Gaël Martinet pour leur aide précieuse et leur disponibilité.

Je tiens aussi à remercier tout ceux qui m'ont soutenu durant ce travail.

Enfin, je tiens à remercier particulièrement mes parents pour leur écoute et leurs conseils.

# Table des matières

# Chapitre 1

# Introduction Générale

Le numérique est aujourd'hui présent partout dans le monde de l'audio. De nombreux systèmes permettent d'enregistrer et de traiter des signaux sonores numérisés. Lors d'une numérisation, le signal électrique provenant du microphone est converti en informations numériques, c'est-à-dire échantillonné et quantifié. La forme d'onde du signal sonore est alors représentée par une liste de valeurs, codées en représentation binaire [1]. Ces valeurs peuvent être stockées dans un fichier audio.

Le signal ainsi numérisé est généralement traité numériquement, par des filtres, des compresseurs, des réverbérations, etc. . . ; puis *mélangé* à d'autres signaux numériques. Le résultat de ce mélange est stocké dans un fichier audio [2]. Cette étape de mélange est appelée *sommation*.

Une console de mixage analogique effectue la sommation des signaux entrants, en additionnant les signaux électriques analogiques de chacune de ses *tranches*. Une console de mixage numérique effectue cette opération en additionnant les signaux numériques de chacune de ses tranches.

Mais, contrairement à une console analogique, on ne peut pas *voir* comment fonctionne en interne une console numérique et nous n'avons pas, ou peu, de détails sur le fonctionnement interne des programmes traitant le signal audio, mis à part quelques spécifications souvent peu détaillées et servant souvent d'arguments commerciaux. Nous souhaitons donc en savoir plus sur le fonctionnement d'un algorithme de sommation.

1. Les signaux audio-numériques peuvent donc être vus comme une liste de nombres qui décrivent la forme d'onde du signal sonore.

2. ou diffusé directement.

Ainsi, de même que différentes consoles analogiques *sonnent* légèrement différemment, les consoles de mixage numérique [3] *sonnent*-elles différemment ?

Mais, comme la question est large, nous nous intéressons dans ce mémoire, uniquement au cas des consoles numériques des logiciels de mixage audio-numérique.

Aujourd'hui, de nombreux logiciels de mixage audio-numérique sont disponibles dans le commerce. Ce sont des programmes informatiques qui fonctionnent sur des ordinateurs.

Afin de bien comprendre comment est codé le signal audio-numérique, nous nous intéressons, dans le premier chapitre, aux différentes représentations binaires dont les deux grands types sont les représentations en virgule fixe et virgule flottante. Comme les ordinateurs calculent tous aujourd'hui très facilement dans la représentation binaire en virgule flottante, les logiciels de mixage audio-numériques utilisent tous maintenant cette représentation pour effectuer le traitement des signaux audio-numériques. Aussi, nous focalisons notre attention sur la représentation binaire en virgule flottante et sur les erreurs d'arrondi commises lors du codage de valeurs dans cette représentation.

Nous avons programmé un plug-in avec l'aide précieuse de mon directeur de mémoire externe Gaël Martinet, fondateur et développeur des plug-in Flux : :. Ce plug-in permet d'écouter en *temps réel* les résultats de quelques opérations de traitement de signal relativement simples, réalisées dans plusieurs formats de la représentation binaire en virgule flottante. Dans le deuxième chapitre, nous nous intéressons donc particulièrement, en nous basant sur l'utilisation de ce plug-in, aux formats 32 et 64 bits de la représentation afin d'étudier et/ou illustrer les problèmes de troncature lors de changements de formats, les erreurs d'arrondi lors de l'application d'un simple gain à un flux audio ou celles engendrées par l'application d'un filtrage relativement simple.

Enfin, dans le troisième chapitre, nous détaillons les protocoles des tests objectifs que nous avons réalisés afin de comparer plusieurs logiciels de mixage audio-numérique, avec du bruit rose puis de comparer un même mixage *musical* réalisé avec les plug-in effectuant leurs calculs avec l'un des formats de la représentation virgule flottante ou bien un autre format de représentation des données.

3. au sens large, englobant les logiciels.

Nous ne réalisons que des mesures *objectives* de ces différences, si elles existent, et non des tests d'écoute afin de déterminer s'il y a, *objectivement*, des différences entre un même mixage réalisé par plusieurs logiciels.

# Chapitre 2

# Les nombres binaires

Dans ce chapitre, nous commencerons par expliquer le codage des nombres entiers positifs ainsi que les règles de l'addition, de la multiplication et de la soustraction binaire, pour nous intéresser ensuite aux entiers relatifs. Nous présenterons les deux représentations binaires des nombres réels : représentation en virgule fixe et représentation en virgule flottante. Et nous étudierons les erreurs pouvant être commises lors du codage de certaines valeurs dans les deux représentation binaire, en insistant sur la représentation en virgule flottante. Enfin nous expliquerons rapidement les processus de conversions d'une représentations à une autre.

Un nombre est la représentation d'une valeur qui s'exprime au moyen de chiffres [1], écrits de façon ordonnée.

Ainsi, un nombre binaire est une valeur numérique représentée en binaire. Un nombre binaire est composée de *bits* [2], et s'exprime en base 2, qui n'utilise exclusivement que deux symboles : 0 et 1.

1. eux-même représentant une valeur
2. contraction de *binary digits*

## 2.1 Codage des nombres entiers positifs

Cette représentation est appelée *binaire naturel*. Un *mot* binaire composé de N bits s'écrit :

$$\underbrace{d_{N-1}d_{N-2}\ldots d_1 d_0}_{N}.$$

Chaque bit a un poids qui dépend de sa place à l'intérieur du mot. La pondération, définissant les poids associés à chaque bit, varie de puissance de 2 en puissance de 2 suivant le schéma suivant :

$$d_{N-1}\cdot 2^{N-1} + d_{N-2}\cdot 2^{N-2}\ldots + d_1\cdot 2^1 + d_0\cdot 2^0$$

où :

- le bit le plus à droite dans le nombre est le bit de poids le plus faible $2^0$, il est appelé en anglais *Least Significant Bit* ou *LSB* ;
- le bit le plus à gauche dans le nombre est le bit de poids le plus fort $2^{N-1}$, il est appelé en anglais *Most Significant Bit* ou *MSB*.

En base 10, que nous utilisons couramment, la pondération, définissant les poids associés à chaque symbole, varie de puissance de 10 en puissance de 10. Pour ne pas prêter à confusion, nous préciserons en indice la base associée au nombre affiché. Ainsi le nombre $1000_2$ représente un nombre binaire contrairement à $1000_{10}$.

La valeur en base 10 d'un mot binaire composé de N bits, interprété comme un entier non signé, est donnée par la formule suivante :

$$x_{10} = \sum_{n=0}^{n=N-1} 2^n \cdot d_n \qquad \text{où } d_n \text{ est le bit } n \text{ de } x_2. \tag{2.1}$$

Avec ce choix de codage, on peut exprimer les nombres entiers de l'intervalle $[0, 2^N - 1]$, soient $2^N$ valeurs différentes.

Exemple : Le nombre binaire 1100 interprété comme un réel non signé représente la valeur :

$$1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 0\cdot 2^0 = 12.$$

## 2.2 Arithmétique binaire

Il nous faut présenter les règles de l'addition, de la multiplication et de la soustraction binaire. Nous ne donnerons pas de détails sur la division binaire.

### 2.2.1 Addition binaire

Les règles de l'addition de deux bits sont les suivantes :

$$
\begin{aligned}
&0_2 \oplus 0_2 = 0_2,\\
&0_2 \oplus 1_2 = 1_2,\\
&1_2 \oplus 0_2 = 1_2,\\
&1_2 \oplus 1_2 = 0_2 \qquad \text{avec report de } 1_2 \text{ sur le bit suivant.}
\end{aligned}
$$

On remarque que, pour exprimer la valeur résultant de l'addition de deux bits $1_2 + 1_2$, il faut incrémenter de $1_2$ le bit associé au poids suivant [3] et remettre à 0 le bit concerné.

Comme en base 10, par exemple, lors de l'addition $5 + 9 = 14$ où le chiffre associé aux dizaines est incrémenté de $1_{10}$.

Le nombre résultant de l'addition de deux nombres entiers positifs $x$ et $y$, est représentable avec un mot de N bits si $x + y \leq 2^N - 1$ comme dans l'exemple suivant avec $N = 4$ bits :

$$
\begin{aligned}
&1100_2 && x = 12_{10}\\
\oplus\ &0001_2 && y = 1_{10}\\
=\ &1101_2 && x + y = 13_{10}.
\end{aligned}
$$

Mais, le nombre résultant de l'addition de deux nombres entiers positifs $x$ et $y$ n'est pas représentable avec un mot de N bits si $x + y > 2^N - 1$ comme dans l'exemple suivant :

$$
\begin{aligned}
&1100_2 && x = 12_{10}\\
\oplus\ &0100_2 && y = 4_{10}\\
=\ &10000_2 && x + y = 16_{10}.
\end{aligned}
$$

3. de la pondération en puissance de 2

Le résultat n'est représentable qu'avec N=5 bits ou plus. On appelle cela *overflow* ou dépassement de capacité. En pratique, le nombre binaire reste à la valeur $2^N - 1$ représentée avec $N = 4$ bits par le nombre binaire $1111_2$. En audio, cela conduit à une distorsion du signal très importante appelée *clipping*.

### 2.2.2 Multiplication binaire

Les règles de la multiplication de deux bits sont les suivantes :

$$\begin{aligned} 0_2 \otimes 0_2 &= 0_2, \\ 0_2 \otimes 1_2 &= 0_2, \\ 1_2 \otimes 0_2 &= 0_2, \\ 1_2 \otimes 1_2 &= 1_2. \end{aligned}$$

Ainsi, le résultat de la multiplication de deux nombres binaires $x$ et $y$ interprétés comme des nombres entiers est exprimable avec un mot binaire composé de N bits si $x \times y \leq 2^N - 1$ comme dans l'exemple suivant pour $N = 4$ bits :

| | | |
|---|---|---|
| | $0101_2$ | $x = 5$ |
| $\otimes$ | $0010_2$ | $y = 2$ |
| $=$ | $0000_2$ | produits partiels |
| $\oplus$ | $0101_2$ | $--$ |
| $\oplus$ | $0000_2$ | $--$ |
| $=$ | $1010_2$ | $x \times y = 10_{10}$ |

Mais, le résultat de la multiplication de deux nombres binaires $x$ et $y$ interprétés comme des nombres entiers, n'est pas exprimable avec un mot binaire composé de N bits si $x \times y > 2^N - 1$ comme dans l'exemple suivant pour $N = 4$ bits :

| | | |
|---|---|---|
| | $0101_2$ | $x = 5$ |
| $\otimes$ | $0100_2$ | $y = 4$ |
| $=$ | $0000_2$ | produits partiels |
| $\oplus$ | $0000_2$ | $--$ |
| $\oplus$ | $0101_2$ | $--$ |
| $=$ | $10100_2$ | $x \times y = 20_{10}$ |

On assiste alors aussi à un dépassement de capacité ou *overflow*.

## 2.3 Soustraction binaire

Les règles de la soustraction de deux bits sont les suivantes :

$$\begin{aligned} &0_2 \ominus 0_2 = 0_2, \\ &0_2 \ominus 1_2 = 1_2 \qquad \text{avec emprunt de } 1_2 \text{ au bit suivant,} \\ &1_2 \ominus 0_2 = 1_2, \\ &1_2 \ominus 1_2 = 0_2. \end{aligned}$$

Ainsi, on peut exprimer le nombre résultant d'une soustraction de 2 nombres entiers positifs $x$ et $y$, avec $x \geq y$ comme dans cet exemple :

$$\begin{array}{rll} & 0101_2 & x = 5_{10} \\ \ominus & 0010_2 & y = 2_{10} \\ = & 0011_2 & x - y = 3_{10}. \end{array}$$

Mais, dans le cas d'une soustraction de 2 nombres, $x - y$ avec $x < y$, il nous faut pouvoir exprimer les nombres négatifs.

## 2.4 Codage des nombres entiers relatifs

### 2.4.1 Représentation Signe - Valeur absolue

La représentation binaire des nombres entiers relatifs la plus simple est appelée *Signe-Valeur absolue*. Le MSB est dédié au codage du signe : "0" pour les entiers positifs et "1" pour les entiers négatifs. Les autres bits forment un mot qui est interprété comme un nombre entier non signé, ou positif.

Par exemple, les nombres entiers relatifs 3 et -3 sont codés, en représentation Signe-Valeur absolue, par les nombres binaires suivants :

$$0011_2 = 3 \text{ et } 1011_2 = -3.$$

Mais, cette représentation présente deux inconvénients :
- il y a deux nombres binaires pour représenter le 0 : $0000_2 = 1000_2$ ;
- l'addition et la soustraction doivent être deux opérations distinctes.

On utilise donc les représentations par complémentation pour coder les nombres réels relatifs.

### 2.4.2 Compléments

Le complément d'un nombre est le nombre qu'il faut ajouter au premier pour en obtenir un troisième, désigné par avance et servant en quelque sorte de référence [4].

L'idée est de se servir du complément d'un nombre pour coder sa valeur opposée. Nous allons donc expliquer brièvement la représentation par complément à 1, puis définir le complément à 2 qui est utilisé pour représenter les valeurs négatives d'un signal audio numérique lors de sa conversion analogique-numérique.

#### Complément à 1

On appelle $\bar{b}$ le complément à 1 d'un bit b, tel que $b + \bar{b} = 1_2$, où $\bar{b}$ et b sont donc toujours différents.

Pour un nombre binaire $x$ composé de n bits, on trouve $\bar{x}$ en changeant, pour chaque bit de $x$, les 0 par des 1, et les 1 par des 0.

Exemple de complémentation à 1 :

$$\begin{aligned} x &= 0011_2 \\ \bar{x} &= 1100_2 \\ x \oplus \bar{x} &= 1111_2. \end{aligned}$$

#### Complément à 2

Le complément à 2 d'un nombre binaire x, appelé $\hat{x}$, est défini par la relation : $\hat{x} = \bar{x} + 1_2$.

Exemple de complémentation à 2 :

$$\begin{aligned} x &= 0011_2 & x &= 3_{10} \\ \bar{x} &= 1100_2 & \bar{x} &= -4_{10} \\ \hat{x} &= \bar{x} + 1_2 & & \\ \hat{x} &= 1100_2 + 0001_2 & & \\ \hat{x} &= 1101_2 & \hat{x} &= -3_{10}. \end{aligned}$$

4. Daniel ROBERT. URL : `http://daniel.robert9.pagesperso-orange.fr/Digit/Digit_7TS.html`

L'intérêt de cette représentation est que la soustraction de 2 nombres signés est vue comme l'addition du premier nombre avec le complément à 2 du second. Il n'y a donc qu'un seul et même circuit qui réalise l'addition et la soustraction.

La valeur en base 10 d'un mot binaire composé de N bits, interprété comme un entier signé complément à 2, est donnée par la formule suivante :

$$x_{10} = -2^{N-1} \cdot d_{N-1} + \sum_{n=0}^{n=N-2} 2^n \cdot d_n \qquad \text{où } d_n \text{ est le bit } n \text{ de } x_2. \tag{2.2}$$

Avec ce choix de codage, on peut exprimer les nombres entiers de l'intervalle $[-2^{N-1}, 2^{N-1} - 1]$ , soient $2^N$ valeurs différentes.

Exemple : un mot binaire composé de 16 bits, interprété comme un entier signé complément à 2, peut coder un nombre entier de l'intervalle $[-32\,768, 32\,767]$, soient $2^{16}$ valeurs possibles.

### 2.4.3 Représentation binaire décalé

L'idée de la représentation en binaire décalé est que l'on peut représenter des nombres entiers négatifs, dans un certain intervalle, uniquement à l'aide de nombres entiers positifs auxquels on retranche une valeur constante prédéfinie.

Ainsi, pour coder $x$ un nombre entier relatif, en représentation binaire décalé, on définit $\check{x}$, la valeur *biaisée* de $x$, comme un entier non signé, par la relation suivante :

$$x = \check{x} - biais$$

avec $biais = 2^{N-1}$ ou $biais = 2^{N-1} - 1$ et N, le nombre de bits sur lequel est codé $\check{x}$.

L'avantage de la représentation binaire décalé pour représenter les nombres entiers relatifs est que toutes les opérations sont effectuées avec des nombres interprétés comme des entiers non signés. Cela rend, par exemple, l'opération de comparaison de deux nombres plus simple.

La valeur en base 10 d'un mot binaire composé de N bits, interprété comme un entier relatif en représentation binaire décalé, est donnée par la formule suivante :

$$x_{10} = -biais + \sum_{n=0}^{n=N-1} 2^n \cdot d_n \qquad \text{où } d_n \text{ est le bit } n \text{ de } x_2. \tag{2.3}$$

Exemple de représentation biaisée, $\check{x}$, d'un nombre entier négatif, $x$, avec $N = 4$ bits et $biais = 2^{N-1}$ :

$$\begin{aligned} \check{x} &= 0101_2 & \check{x} &= 5 \text{ et } N = 4 \\ biais &= 1000_2 & biais &= 8 \\ x &= \check{x} - biais & x &= 5 - 8 \\ x &= 1101_2 & x &= -3. \end{aligned}$$

### 2.4.4 Rappel des différentes représentations binaires des nombres entiers relatifs

Afin de résumer et de rassembler toutes les informations nécessaires à une bonne compréhension du codage binaire des nombres entiers relatifs, nous donnons le tableau 2.1 qui regroupe les différentes représentations binaires des nombres entiers relatifs avec des mots binaires composés de 4 bits.

| Base 10 | Codage par signe-valeur absolue | Codage par complément à 1 | Codage par complément à 2 | Codage par représentation décalée (biais= 8) |
|---|---|---|---|---|
| +7 | 0111 | 0111 | 0111 | 1111 |
| +6 | 0110 | 0110 | 0110 | 1110 |
| +5 | 0101 | 0101 | 0101 | 1101 |
| +4 | 0100 | 0100 | 0100 | 1100 |
| +3 | 0011 | 0011 | 0011 | 1011 |
| +2 | 0010 | 0010 | 0010 | 1010 |
| +1 | 0001 | 0001 | 0001 | 1001 |
| +0 | 0000 | 0000 | 0000 | 1000 |
| -0 | 1000 | 1111 | - | - |
| -1 | 1001 | 1110 | 1111 | 0111 |
| -2 | 1010 | 1101 | 1110 | 0110 |
| -3 | 1011 | 1100 | 1101 | 0101 |
| -4 | 1100 | 1011 | 1100 | 0100 |
| -5 | 1101 | 1010 | 1011 | 0011 |
| -6 | 1110 | 1001 | 1010 | 0010 |
| -7 | 1111 | 1000 | 1001 | 0001 |
| -8 | - | - | 1000 | 0000 |

TABLE 2.1 – Les différentes représentations binaires des entiers relatifs avec des mots binaires composés de 4 bits.

## 2.5 Codage des nombres réels

Pour coder les nombres réels en binaire, nous disposons d'une représentation en *virgule fixe* et d'une autre en *virgule flottante*. Par analogie, en base 10, nous pouvons afficher un même nombre sous les deux formes suivantes :

$$x = 0,00245 \qquad\qquad x = 2,45 \times 10^{-3}.$$

### 2.5.1 Représentation en virgule fixe

La représentation des nombres binaires en virgule fixe implique une position immuable et tacite de la virgule [5].

On définit les représentations $U(a, b)$ et $A(a, b)$ comme les représentations des nombres réels, respectivement non signés et signés, par un mot binaire interprété avec $a$ bits avant la virgule et $b$ bits après la virgule.

#### Les nombres réels non-signés

La valeur en base 10 d'un nombre binaire $x$ dans la représentation $U(a, b)$, est donnée par la formule suivante :

$$x_{10} = \frac{1}{2^b} \cdot \Big[ \sum_{n=0}^{n=N-1} 2^n \cdot d_n \Big] \qquad \text{avec : } a = N - b. \tag{2.4}$$

On peut ainsi exprimer, $2^N$ valeurs dans l'intervalle $[0, 2^a - 2^{-b}]$.

Par exemple, avec un mot binaire composé de 8 bits interprété comme un nombre binaire en virgule flottante dans la représentation $U(5, 3)$, la pondération, définissant les poids associés à chaque bit, varie suivant le schéma suivant :

$$x_7 \cdot 2^4 + x_6 \cdot 2^3 + x_5 \cdot 2^2 + x_4 \cdot 2^1 + x_3 \cdot 2^0 + x_2 \cdot 2^{-1} + x_1 \cdot 2^{-2} + x_0 \cdot 2^{-3}.$$

Par ailleurs, on remarque que quand $b = 0$, la virgule est donc interprétée comme étant placée après le LSB et l'équation (2.4) devient :

$$x_{10} = \sum_{n=0}^{n=N-1} 2^n \cdot d_n \qquad \text{avec } a = N.$$

La représentation en binaire naturel, définie à la section 2.1, est bien un cas particulier de la représentation des réels en virgule fixe.

Exemple : Avec un mot binaire de 16 bits, interprété comme un réel non signé, dans la représentation $A(16, 0)$, on peut prendre des valeurs dans l'intervalle $[0, 65\,535]$, soient $2^{16}$ valeurs différentes.

5. Randy Yates. *Fixed-Point Arithmetic: An Introduction.* Rap. tech. Digital Signal Labs, 2009

**Les nombres réels signés complément à 2**

La valeur d'un nombre binaire $x$ dans la représentation $A(a, b)$ est donnée par la formule suivante :

$$x = \frac{1}{2^b} \cdot \Big[ -2^{N-1} \cdot x_{N-1} + \sum_{n=0}^{n=N-2} 2^n \cdot x_n \Big] \qquad \text{avec } a = N - b - 1. \tag{2.5}$$

On peut ainsi exprimer $2^N$ valeurs dans l'intervalle $[-2^a, 2^a - 2^{-b}]$.

Exemple : Avec un mot binaire de 16 bits interprété comme un réel signé et codé en complément à 2, dans la représentation $A(15, 0)$, on peut prendre des valeurs dans l'intervalle $[-32\,768, 32\,767]$, soient $2^{16}$ valeurs différentes.

### 2.5.2 Représentation en virgule flottante

Un nombre en virgule flottante a une représentation de la forme :

$$(-1)^s \cdot m \cdot \beta^e$$

où $s$ est la valeur du bit de signe, $m$ le signifiant, $\beta$ la base[6] et e l'exposant.

Les nombres à virgule flottante sont utilisés afin de représenter de très grandes valeurs comme de très petites valeurs, avec relativement peu de *digits*.

Exemple : En base 10, le nombre réel $x = 3,141 \times 10^3$ est $10\,000_{10}$ fois plus grand que le nombre $y = 3,141 \times 10^{-1}$.

#### Le standard IEEE754

Un standard a été édicté par l'IEEE (Institute of Electrical and Electronics Engineers), la norme 754, afin d'uniformiser l'arithmétique binaire en virgule flottante. Cette dernière impose trois formats : *single*, *double* et *double-extended* exposés dans la table 2.2[7] :

| Format | type C++ | Taille(bits) | p (bits) | K (bits) |
|---|---|---|---|---|
| Single | float | 32 | 24 | 8 |
| Double | double | 64 | 53 | 11 |
| Double-Extended | long double | $\geq 79$ | $\geq 64$ | $\geq 15$ |

TABLE 2.2 – Les formats du standard IEEE 754, avec $p - 1$ le nombre de bits alloués au signifiant et $K$ le nombre de bits alloués au codage de l'exposant.

**Normalisation du signifiant :** Comme le montre l'exemple suivant en base 10, un nombre réel peut s'écrire de différentes façons avec la représentation en virgule flottante :

$$x = 3,141 \times 10^0$$
$$x = 31,41 \times 10^{-1}$$
$$x = 3141,0 \times 10^{-4}$$

6. en général 2 ou 10

7. Paul Zimmermann VINCENT LEFEVRE. *Arithmetique flottante*. Institut National de Recherche en Informatique et en Automatique. 2004

Afin de n'avoir qu'un nombre binaire pour chaque valeur, l'exposant sera minimisé, le premier bit du signifiant sera toujours non-nul et la virgule placée juste derrière. On appelle cela la *normalisation*.

Une conséquence intéressante de la normalisation en base $\beta = 2$ est que le premier bit du signifiant est toujours $1_2$. On peut alors décider de ne pas le stocker physiquement en mémoire, on parle alors de bit de poids fort implicite [8].

Le signifiant normalisé prend alors pour forme :

$$m = (1, f)_2$$

avec $f$, appelée *fraction*, interprétée comme un nombre réel non signé.

Il y a $p-1$ bits alloués à la représentation du signifiant mais le premier bit n'est pas codé, si bien que $p-1$ bits sont alloués à la représentation de la fraction $f$.

On peut donc coder $f$ en utilisant la représentation $U(0, p)$ définie à la section 2.5.1, soit 0 bit avant la virgule et $p-1$ bits après la virgule.

La valeur associée à la fraction, $f$, est donc comprise toujours comprise entre :

$$\begin{aligned} 0 \leq & f \leq 2^0 - 2^{-(p-1)}, \\ 0 \leq & f \leq 1 - 2^{1-p}. \end{aligned}$$

Ainsi, le signifiant $m$, quand il est normalisé, est compris entre :

$$1 \leq m \leq 2 - 2^{1-p}.$$

Il faut donc comprendre que dès que la fraction est supérieure à $1 - 2^{1-p}$, l'exposant est incrémenté de 1 et la fraction remise à zéro.

Exemple avec $p = 3$ bits. Quand $m$ est normalisé, il prend successivement pour valeur :

| | | |
|---|---|---|
| $m = 1,00_2$ | représente | $1 \cdot 2^0 + \cdot 2^{-1} + 0 \cdot 2^{-2} = 1,00_{10}$ |
| $m = 1,01_2$ | représente | $1 \cdot 2^0 + \cdot 2^{-1} + 1 \cdot 2^{-2} = 1,25_{10}$ |
| $m = 1,10_2$ | représente | $1 \cdot 2^0 + \cdot 2^{-1} + 0 \cdot 2^{-2} = 1,50_{10}$ |
| $m = 1,11_2$ | représente | $1 \cdot 2^0 + \cdot 2^{-1} + 1 \cdot 2^{-2} = 1,75_{10}$ |

8. *implicit leading bit* en anglais.

**Exposant :** Le standard IEEE754 utilise la représentation en binaire décalé, définie à la section 2.4.3 pour le codage de l'exposant. Ce dernier est représenté par l'exposant *biaisé* avec $K$ bits, et la constante utilisée appelée *biais* est égale à $2^{(K-1)} - 1$.

On appellera donc $\breve{k}$, l'exposant *biaisé*, interprété comme un nombre entier non-signé. On peut donc borner $\breve{k}$ avec l'inéquation suivante :

$$0 \leq \breve{k} \leq 2^K - 1. \tag{2.6}$$

On appellera $e$ l'exposant non-biaisé, qui est donc un nombre entier relatif, et on obtient la valeur de $e$ en retranchant la constante *biais* à $\breve{k}$ :

$$e = \breve{k} - biais \tag{2.7}$$
$$e = \breve{k} - (2^{(K-1)} - 1). \tag{2.8}$$

Le standard IEEE754 définit deux nombres, $e_{min}$ et $e_{max}$, comme les exposants, non-biaisés, minimum et maximum pour coder les nombres réels normalisés en représentation virgule flottante (cf.Table 2.3). Ils sont définis par l'inégalité suivante :

$$e_{min} - 1 \leq e \leq e_{max} + 1$$

En retranchant *biais* à l'inéquation (2.6) on obtient :

$$2^{(K-1)} - 1 \leq e \leq 2^K - 2^{(K-1)}.$$

On peut donc exprimer $e_{min}$ et $e_{max}$ par les équations suivantes :

$$e_{min} = -2^{(K-1)} + 2 \qquad \text{et} \qquad e_{max} = 2^K - 2^{(K-1)} - 1.$$

Exemple en simple précision où $K = 8$ bits :

$$e_{min} = -2^{(8-1)} + 2 \qquad e_{max} = 2^8 - 2^{(8-1)} - 1$$
$$e_{min} = -126 \qquad e_{max} = 127.$$

Le fait que la valeur absolue de l'exposant minimum soit inférieure à l'exposant maximum, $|e_{min}| < |e_{min}|$ permet à l'inverse du plus petit nombre normalisé exprimable $2^{-e_{min}}$, d'être exprimable dans le format sans dépassement de capacité.

| Format | type C++ | K (bits) | $e_{min}$ | $e_{max}$ |
|---|---|---|---|---|
| Single | float | 8 | -126 | 127 |
| Double | double | 11 | -1022 | 1023 |
| Double-Extended | long double | $\geq 16$ | $\geq -16382$ | $\leq 163823$ |

TABLE 2.3 – Les formats du standard IEEE 754, l'exposant. Avec K, le nombre de bits alloués au codage de l'exposant et $e_{min}$ et $e_{max}$ les exposants non-biaisés minimum et maximum utilisés pour le codage des nombres réels en virgule flottante normalisés.

**Zéro et Valeurs spéciales :** Le standard définit aussi par convention la valeur 0, ainsi que les valeurs *spéciales* $\pm\infty$ et $NaN$, comme le précise la table 2.4.

| Exposant | Fraction | Valeur représentée |
|---|---|---|
| $e = e_{min} - 1$ | $f = 0$ | $\pm 0$ |
| $e = e_{min} - 1$ | $f \neq 0$ | $(0, f) \times 2^{e_{min}}$ |
| $e_{min} \leq e \leq e_{max}$ | - | $(1, f) \times 2^{e}$ |
| $e = e_{max} + 1$ | $f = 0$ | $\pm\infty$ |
| $e = e_{max} + 1$ | $f \neq 0$ | $NaN$ |

TABLE 2.4 – Valeurs représentées, par les nombres flottants définis par le standard IEEE 754, en fonction des valeurs de leurs exposants non-biaisés $e$ et de leurs fraction $f$. Les symboles $\pm\infty$ représentent les valeurs trop grandes pour être utilisées par la représentation et $Nan$ représente les nombres qui sont produits par des opérations impossibles à calculer.

Ainsi la valeur 0 est représentée par les nombres [9] dont l'exposant non-biaisé est égal à $e_{min} - 1$ et dont la fraction est nulle.

Les nombres codés par l'exposant $e_{max} + 1$, et dont la fraction est non-nulle sont appelés *nombres dénormalisés*. L'étude de ces nombres demanderait un travail particulier mais qui n'est pas, ici, l'objet du sujet. Je ne développerai donc pas cet aspect du codage binaire en virgule flottante.

Les valeurs $e_{min}$ et $e_{max}$ représentent les exposants minimum et maximum qui peuvent être utilisés pour coder les nombres en représentation binaire, virgule flottante, normalisés, comme expliqué au paragraphe précédent.

9. Il y a deux nombres pour représenter le 0 car le signe est indiqué par le bit de signe.

Les caractères $\pm\infty$ sont utilisés pour exprimer les nombres dont la valeur absolue est trop grande pour être exprimable dans le format utilisé, comme dans les cas d'overflow.

Le caractère $NaN$, contraction de l'expression anglaise "Not a Number", est produit lors de certaines opérations définies par la table 2.5.

| Opération | $NaN$ est produit par |
|---|---|
| $+$ | $\infty + (-\infty)$ |
| $\times$ | $0 \times \infty$ |
| $/$ | $0/0$,$\infty/\infty$ |
| $\sqrt{}$ | $\sqrt{x}$ (quand $x < 0$) |

TABLE 2.5 – Opérations qui produisent un $NaN$.

## 2.6 Erreur de troncature et d'arrondi

Maintenant que nous en savons plus sur les représentations binaires en virgule fixe et flottante, il nous faut étudier les erreurs générées par la représentation des nombres réels avec des mots binaires composés d'un nombre de bit fini.

Le fait que les nombres réels soient codés en binaire avec un nombre défini de bits, implique que certains nombres ne peuvent pas être représentés par manque de bits pour les exprimer exactement. Par analogie, en base 10 lorsque nous utilisons un nombre défini de décimales, nous ne pouvons représenter certaines valeurs que par des représentations approchées. Par exemple, le nombre réel $5,2538$ ne peut pas être exprimé exactement si l'on utilise une représentation avec moins de 4 décimales.

De plus, certains nombres réels nécessitent une infinité de décimales pour être représentés exactement. Par exemple $\frac{1}{3}$ ne peut pas être exactement représenté [10] avec un nombre fini de décimales. On peut adopter, avec une précision de 5 décimales, une valeur approchée de $\frac{1}{3} = 0,33333$.

Ainsi tous les nombres réels ne pouvant pas être exprimés exactement, dans la représentation utilisée [11], seront approchés en utilisant la *troncature* ou *l'arrondi*.

10. sauf sous sa forme fractionnaire
11. hormis les cas de dépassement de capacité

Donnons un exemple avec l'approximation, par troncature ou par arrondi, d'un nombre réel $x$ exprimable exactement avec 4 décimales, dans un format qui utilise moins de 4 décimales :

$$\begin{aligned} x &= 5,2538; \\ trunc(x) &= 5,253 && \text{troncature de la valeur;} \\ round(x) &= 5,254 && \text{arrondi de la valeur.} \end{aligned}$$

Nous pouvons donc tronquer la valeur pour ne garder que les 3 premières décimales, ou bien arrondir la valeur suivant un mode d'arrondi prédéfini [12].

Mais ces approximations génèrent des erreurs, qui en audio, peuvent se traduire par des distorsions du signal. Il nous faudra donc mesurer ces erreurs.

Nous étudierons donc en premier lieu, la représentation de l'erreur générée par une approximation, puis nous nous intéresserons particulièrement à l'erreur d'arrondi dans la représentation en virgule flottante car c'est un point fondamental du sujet. Enfin, nous analyserons l'erreur générée lors de la troncature d'un nombre réel en un nombre entier.

### 2.6.1 Représentations de l'erreur

On utilise deux mesures de l'erreur :
- l'erreur absolue est donnée par la relation $\Delta_x = \diamond(x) - x$ ;
- l'erreur relative est donnée par la relation $\delta_x = \frac{\Delta_x}{x}$ ;

avec $\diamond(x)$, une approximation [13] de $x$ qui peut donc s'écrire sous la forme :

$$\diamond(x) = x + \Delta_x \qquad et \qquad \diamond(x) = x \cdot (1 + \delta_x).$$

Lorsqu'on effectue une opération sur plusieurs nombres, les erreurs s'additionnent ou se compensent. Ainsi, l'erreur relative lors d'une addition peut être estimée par l'équation :

$$\begin{aligned} z &= x + y \\ \diamond(z) &= \diamond[(\diamond(x) + \diamond(y))] \\ z &= [x \cdot (1 + \delta_x) + y \cdot (1 + \delta_y)] \cdot (1 + \delta_{x+y}) \\ z &= (x + y) + x(\delta_x + \delta_{x+y} + \delta_x \cdot \delta_{x+y}) + y(\delta_y + \delta_{x+y} + \delta_y \cdot \delta_{x+y}). \end{aligned}$$

12. ici au plus proche nombre représentable
13. troncature ou arrondi ;

### 2.6.2 Erreur d'arrondi dans la représentation en virgule flottante

Il est fréquent qu'un nombre réel ne puisse pas être représenté exactement dans la représentation car seul un ensemble fini de nombres, parmi l'ensemble infini des nombres réels, peuvent être exprimés exactement dans la représentation binaire en virgule flottante respectant le standard IEEE754.

On appellera donc *nombres flottants*, les nombres réels étant exprimables exactement par la représentation binaire en virgule flottante dans une précision donnée. Et l'on notera $float(x)$ et $double(x)$, les nombres flottants, respectivement simple et double précision, représentant une valeur arrondie de $x$.

Ainsi un nombre réel n'appartenant pas à l'ensemble des nombres flottants, mais étant compris dans l'intervalle prévu par le format pour représenter les nombres normalisés, sera arrondi à l'un des deux nombres flottants encadrant la valeur exacte de $x$ :

$$float_{\text{précédent}} < |x| < float_{suivant}$$

Le standard IEEE754 prévoit que le résultat d'une opération arithmétique soit exactement calculé puis arrondi dans la précision donnée, en utilisant l'une des quatre méthodes d'arrondi suivante :

- l'arrondi au nombre le plus proche (pair) ;
- l'arrondi au nombre le plus proche vers $+\infty$ ;
- l'arrondi au nombre le plus proche vers $-\infty$ ;
- l'arrondi au nombre le plus proche vers 0.

Par défaut, c'est l'arrondi vers le nombre le plus proche (pair) qui est utilisé. Cela signifie que le nombre flottant $float(x)$ est le nombre flottant le plus proche de $x$. Si x est exactement entre deux nombres flottants, $float(x)$ est le nombre pair.

Afin de bien visualiser comment sont répartis les nombres flottants dans l'intervalle permis par une représentation donnée, on peut regarder la table 2.6 listant les nombres flottants positifs avec $p = 3$, soit 2 $bits$ alloués au codage de la fraction car le premier bit, 1, est implicite, et, $K = 3$ bits alloués au codage de l'exposant. Nous n'avons pas représenté le bit de signe.

| Bit implicite | Fraction | Exposant | Valeur en base 10 | Valeur représentée |
|---|---|---|---|---|
| 0 | 00 | 000 | 0 | $0 \cdot 2^{-2}$ |
| 0 | 01 | 000 | 0,0625 | $2^{-2} \cdot 2^{-2}$ |
| 0 | 10 | 000 | 0,1250 | $2^{-1} \cdot 2^{-2}$ |
| 0 | 11 | 000 | 0,1875 | $(2^{-1} + 2^{-2}) \cdot 2^{-2}$ |
| 1 | 00 | 001 | 0,2500 | $1 \cdot 2^{-2}$ |
| 1 | 01 | 001 | 0,3175 | $(1 + 2^{-2}) \cdot 2^{-2}$ |
| 1 | 10 | 001 | 0,3750 | $(1 + 2^{-1}) \cdot 2^{-2}$ |
| 1 | 11 | 001 | 0,4375 | $(1 + 2^{-1} + 2^{-2}) \cdot 2^{-2}$ |
| 1 | 00 | 010 | 0,500 | $1 \cdot 2^{-1}$ |
| 1 | 01 | 010 | 0,625 | $(1 + 2^{-2}) \cdot 2^{-1}$ |
| 1 | 10 | 010 | 0,750 | $(1 + 2^{-1}) \cdot 2^{-1}$ |
| 1 | 11 | 010 | 0,875 | $(1 + 2^{-1} + 2^{-2}) \cdot 2^{-1}$ |
| 1 | 00 | 011 | 1,00 | $1 \cdot 2^{0}$ |
| 1 | 01 | 011 | 1,25 | $(1 + 2^{-2}) \cdot 2^{0}$ |
| 1 | 10 | 011 | 1,50 | $(1 + 2^{-1}) \cdot 2^{0}$ |
| 1 | 11 | 011 | 1,75 | $(1 + 2^{-1} + 2^{-2}) \cdot 2^{0}$ |
| 1 | 00 | 100 | 2,00 | $1 \cdot 2^{1}$ |
| 1 | 01 | 100 | 2,5 | $(1 + 2^{-2}) \cdot 2^{1}$ |
| 1 | 10 | 100 | 3,0 | $(1 + 2^{-1}) \cdot 2^{1}$ |
| 1 | 11 | 100 | 3,5 | $(1 + 2^{-1} + 2^{-2}) \cdot 2^{1}$ |
| 1 | 00 | 101 | 4 | $1 \cdot 2^{2}$ |
| 1 | 01 | 101 | 5 | $(1 + 2^{-2}) \cdot 2^{2}$ |
| 1 | 10 | 101 | 6 | $(1 + 2^{-1}) \cdot 2^{2}$ |
| 1 | 11 | 101 | 7 | $(1 + 2^{-1} + 2^{-2}) \cdot 2^{2}$ |
| 1 | 00 | 110 | 08 | $1 \cdot 2^{3}$ |
| 1 | 01 | 110 | 10 | $(1 + 2^{-2}) \cdot 2^{3}$ |
| 1 | 10 | 110 | 12 | $(1 + 2^{-1}) \cdot 2^{3}$ |
| 1 | 11 | 110 | 14 | $(1 + 2^{-1} + 2^{-2}) \cdot 2^{3}$ |
| 1 | 00 | 111 | $+\infty$ | $+\infty$ |
| 1 | 01 | 111 | $+\infty$ | $+\infty$ |
| 1 | 10 | 111 | $+\infty$ | $+\infty$ |
| 1 | 11 | 111 | $+\infty$ | $+\infty$ |

TABLE 2.6 – Liste des nombres flottants positifs, et des valeurs en base 10 qu'ils représentent, avec $p = 3$ soient 2 bits alloués au codage de la fraction, car le premier bit est implicite, et, 3 bits pour l'exposant.

En utilisant l'arrondi au nombre le plus proche (pair), l'erreur absolue, $\Delta_x$, commise lors du codage d'un nombre réel $x$ est au maximum de la moitié de l'intervalle entre les deux nombres les plus proches de $x$, exprimables exactement dans la représentation utilisée. La figure 2.1 représente l'intervalle entre les deux nombres flottants encadrant la valeur exacte de $x$ et montre l'erreur absolue maximum pouvant être commise.

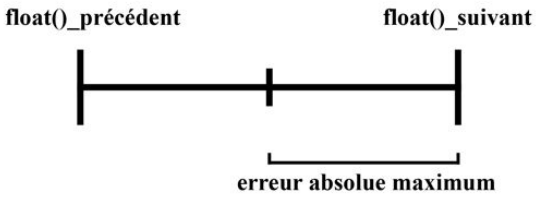


FIGURE 2.1 – Schéma représentant les deux nombres flottants encadrant la valeur exacte d'un nombre $x$ et montrant l'erreur absolue maximum pouvant être commise lors de l'arrondi, de la valeur $x$, prévu par le standard IEEE754.

Or on sait que le signifiant normalisé $m$, défini à la section 2.5.2 peut prendre $2^{(p-1)}$ valeurs, régulièrement réparties dans l'intervalle $[0, 2-2^{1-p}]$, l'intervalle entre deux de ces valeurs étant égal à $2^{1-p}$.

On sait aussi que les nombres flottants sont de la forme : $float(x) = \pm m \cdot 2^e$. Donc le signifiant normalisé est multiplié par $2^e$ avec $e$, l'exposant non-biaisé. Ainsi l'intervalle entre deux nombres flottants, est le même pour tous les nombres flottants ayant le même exposant, mais il est multiplié par deux à chaque fois que l'exposant est incrémenté, et il est divisé par deux lorsque l'exposant est décrémenté de 1. La figure 2.2 et le tableau 2.6 peuvent aider à se représenter cet intervalle.

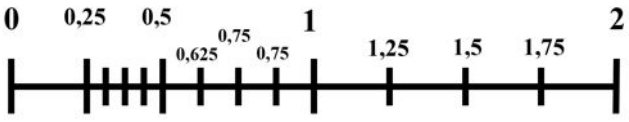


FIGURE 2.2 – Représentation sur un axe orthonormé, des premiers nombres flottants, normalisés positifs, de la représentation $p = 3$, $K = 3$.

Ainsi, pour résumer, plus le nombre à coder $x$ est grand, plus l'exposant $e$ est grand et donc plus l'intervalle entre les deux nombres flottants les plus proches de $x$ est grand. Cela signifie que l'erreur absolue, $\Delta_x$, commise lors de l'arrondi de $x$ par le nombre flottant $float(x)$, augmente et diminue suivant la valeur de $x$.

En prenant compte des remarques précédentes, l'erreur relative maximum, $\delta_{max}$, pouvant être commise lors de l'opération d'arrondi [14] d'un nombre dans une représentation virgule flottante normalisée avec une précision donnée, est définie par l'égalité suivante :

$$\delta_{max} = 2^{1-p}.$$

On peut donner la valeur $x_{dB}$ en dB FS d'une valeur $x$ avec le calcul suivant :

$$x_{dB} = 20 \cdot log_{10}(\frac{|x|}{1})$$

en prenant $1_{10}$ comme valeur *référence*, donc représentant 0 dB FS.

Ainsi, lors du codage d'un nombre réel en représentation à virgule flottante simple précision, $\delta_x$ sera au maximum égale à $\delta_{max\_dB} =\approx 6 \times 10^{-8}$ soient environ $\delta_{max\_dB} = -144\, dB\ FS$. Cela signifie aussi que $\Delta_{x\_dB} = 20 \cdot log_{10}(|x| \cdot |\delta max\_dB|)$ sera au maximum égale à $20 \cdot log_{10}(x) + 20 \cdot log_{10}(|\delta max\_dB|)$.

Donc, le niveau de l'erreur absolue en dB FS, lors de l'arrondi d'un nombre réel en virgule flottante simple précision par la méthode prévue par le standard IEEE754, est toujours inférieur d'au minimum 144 dB FS au niveau du signal entrant.

Plus la valeur que j'arrondis est grande, plus l'erreur absolue est plus grande. Plus la valeur que j'arrondis est petite, plus l'erreur sera petite. Mais l'erreur relative maximum reste donc constante.

Ainsi, à chaque opération d'arrondi, une erreur relative de $2^{(1-p)}$, au maximum, vient s'ajouter aux autres erreurs précédemment commises. Car, comme expliqué à la sous-section 2.6.1, les erreurs peuvent s'ajouter, ou se compenser, lors de chaque opération arithmétique utilisée lors des traitements.

14. au plus proche (pair).

**Epsilon machine :** Le standard IEEE754 définit donc $\epsilon_{machine}$ comme le plus petit nombre flottant pouvant être additionné à 1,0 pour donner un nombre flottant différent de 1,0 dans un format donné. (cf. Table2.7).

$$\epsilon_{machine} = 2^{1-p}$$

Cette dernière correspond donc, comme expliqué aux paragraphes précédents, à l'erreur relative maximum pouvant être commise lors du codage d'une valeur en un nombre flottant. $\epsilon_{machine}$ dépend uniquement de $p$, le nombre de bits alloués au codage du signifiant. Cela décrit donc l'exactitude mathématique du format [15].

En simple précision $\epsilon_{mach} \approx 6 \times 10^{-8}$, ce qui signifie que tous les nombres de plus de 7 chiffres après la virgule seront arrondis. En double précision, $\epsilon_{mach} \approx 10^{-16}$ ce qui nous amène à considérer environ 16 chiffres après la virgule.

| Format | type C++ | $\epsilon_{mach} = 2^{1-p}$ | $\epsilon_{mach}(dBFS)$ |
|---|---|---|---|
| Single | float | $\approx 6 \times 10^{-8}$ | -144,5 |
| Double | double | $\approx 10^{-16}$ | -319,1 |
| Double-Extended | long double | $\approx 5 \times 10^{-19}$ | -385,3 |

TABLE 2.7 – Valeurs de $\epsilon_{mach}$ dans les différents formats du standard IEEE 754.

**Dynamique de codage :** En se plaçant dans le cadre des conventions précédentes, le plus petit nombre positif normalisé représentable en virgule flottante est :

$$float(x)_{min} = 2^{e_{min}},$$

soit en simple précision :

$$\begin{aligned} float(x)_{min} &= 2^{-126} \\ float(x)_{min} &\simeq 1,2 \times 10^{-38}. \end{aligned}$$

Et, le plus grand nombre positif représentable en virgule flottante est :

$$\begin{aligned} float(x)_{max} &= m_{max} \times 2^{e_{max}+1} \\ float(x)_{max} &\simeq (2 - 2^{-23}) \times 2^{127} \\ float(x)_{max} &\simeq 3,4 \times 10^{38}, \end{aligned}$$

15. Attention celle-ci est différente du plus petit nombre représentable par le format.

ce qui correspond à une dynamique de codage en dB de :

$$\begin{aligned} 20 \cdot \log_{10}(float(x)_{max}) - 20 \cdot \log_{10}(float(x)_{min}) &\simeq 771 - (-758) \\ &\simeq 1530 \text{ dB.} \end{aligned}$$

Le fait qu'en représentation en virgule flottante le nombre représentant le 0 dB FS ne soit pas la plus grande valeur codable implique que la situation d'*overflow*[16] n'est plus vraiment un problème lorsqu'on utilise cette représentation en audio numérique. Cela constitue l'un des avantages du codage en virgule flottante pour l'audio-numérique.

### 2.6.3 Petits programmes illustrant l'erreur d'arrondi

Afin d'illustrer l'erreur d'arrondi en précision simple, j'ai mis en place quelques programmes en C++.

Le premier programme calcule les erreurs commises par les arrondis de deux nombres réels lors du codage de leurs valeurs en représentation binaire virgule flottante simple précision par rapport à celles commises par une représentation double précision.

Le second programme calcule la soustraction de ces deux nombres et calcule les erreurs d'arrondis commises lors du codage du résultat de cette dernière.

Un troisième programme effectuent $n$ fois un calcul simple mais utilisant la valeur du résultat précédent pour calculer la valeur du résultat actuel. Les erreurs d'arrondi commises par le calcul en simple précision sont calculées par rapport au valeurs calculées en double précision.

Enfin un dernier programme réalise les sommations de tableaux de valeurs comprises entre -1 et 1 et générées aléatoirement, en utilisant les deux précisions du codage en virgule flottante. Les différence entre les deux versions des sommations seront analysées par le programme.

16. ou dépassement de capacité (Cf. sous-section 2.2

### Codage d'une valeur en virgule flottante

Soient les deux nombres

$$x = 0,993 \text{ et } y = 0,014 \ .$$

On définit deux variables en précision simple, $x\_f$ et $y\_f$, et deux autres variables en double précision, $x\_d$ et $y\_d$, dans lesquelles on stocke les nombres $x$ et $y$, par les lignes de code de la figure 2.3.

```
float x_f = 1.010;
float y_f = 0.993;
double x_d = 1.010;
double y_d = 0.993;
```

FIGURE 2.3 – Déclarations et définitions de quatre variables en C++, $x\_f$ et $y\_f$, en précision simple et $x\_d$ et $y\_d$ en double précision.

Les variables $x\_f$ et $y\_f$ contiennent donc les nombres flottants $float(x)$ et $float(y)$ tandis que les variables $x\_d$ et $y\_d$ contiennent les nombres flottants $double(x)$ et $double(y)$.

Donc, si l'on demande au programme d'afficher ces nombres avec une précision de 8 ou 16 chiffres significatifs après la virgule, le programme nous donne :

```
x_f = 0.99299997,
y_f = 0.01400000,
x_d = 0.9930000000000000,
y_d = 0.0140000000000000,
```

ainsi que leurs valeurs en dB FS :

```
x_d_dB = -0.06 dB FS,
y_d_dB = -37.1 dB FS.
```

Pour calculer la valeur en dB FS d'une variable, nous avons écrit la fonction $linear2dB$ (cf. Figure 2.4).

Afin d'illustrer l'erreur d'arrondi lors du codage d'une valeur en précision simple, on définit donc la différence (en double précision) entre la valeur de $x\_f$ convertie en double précision et la valeur $x_d$.[17] par les lignes de codes de la figure 2.5.

17. nous allons donc calculer double((float(x))-double(x))

```
double Linear2dB(double x)
{
   double res = log10(x);
   return 20.0*res;
}
```

FIGURE 2.4 – Déclaration et définition de la fonction *Linear*2*dB* servant à calculer la valeur en dBFS d'un nombre en double précision.

```
double erreur_x_float=(double)x_f-x_d;
double erreur_y_float=(double)y_f-y_d;
double erreur_x_float_rel=erreur_x_float/x_d;
double erreur_y_float_rel=erreur_y_float/y_d;
```

FIGURE 2.5 – Déclarations et définitions, en C++, des erreurs absolues et relatives commises par le codage des nombres réels $x$ et $y$, en nombres flottants en simple précision en prenant pour référence la valeur codée en double précision.

Le programme nous donne les valeurs suivantes, affichées avec 16 chiffres après la virgule :

```
erreur_x_float = -0.0000000290870666,
erreur_y_float =  0.0000000004321337,
erreur_x_float_dB = -150.7 dB FS,
erreur_y_float_dB = -187.3 dB FS,
erreur_x_float_rel_dB -150.7 dB FS,
erreur_y_float_rel_dB -150.2 dB FS.
```

Les erreurs relatives commises lors des arrondis des deux nombres $x$ et $y$ sont environ égales comme expliqué à la sous-section 2.6.2.

### Erreurs d'arrondis lors d'un calcul simple

Effectuons maintenant la soustraction de ces deux même nombres.

$$S = x - y$$

Le résultat exact est :

$$S = 0.979.$$

On définit ce calcul dont le résultat sera stocké dans une variable simple précision et dans une autre double précision, par les lignes de code de la figure 2.6.

```
float S_float = x_f - y_f;
double S_double = x_d - y_d;
```

FIGURE 2.6 – Déclarations et définitions, en C++, des variables stockant les résultats de soustractions réalisées en simple et double précisions.

Puis, on demande au programme d'afficher le résultat avec une précision de 8 ou 16 chiffres significatifs après la virgule. Il donne :

```
S_float = 0.978999972,
S_double = 0.97899999999999998.
```

On définit donc la différence (en double précision) entre la valeur de $S\_float$ et la valeur de $S\_double$ codée en double précision par les lignes de codes de la figure 2.7.

```
double erreur_S_float = (double)S_float - S_double;
double erreur_S_float_rel = erreur_S_float/S_double;
```

FIGURE 2.7 – Déclarations et définitions, en C++, des erreurs absolues et relatives commises par le codage en nombre flottant en simple précision en prenant pour référence la valeur codée en double précision.

Et, on affiche les résultats avec 17 chiffres significatifs après la virgule :

```
erreur_S_float = -0.00000002765655516,
erreur_S_float_rel = -0.00000002824980098,
erreur_S_float_dB -151.2 dB FS,
erreur_S_float_rel_dB -151.0 dB FS.
```

Nous retrouvons toujours notre erreur relative d'environ -150 dB FS.

### Propagation de l'erreur lors de calculs itératifs

Un autre programme permet de montrer que l'erreur absolue peut dépasser $\delta_{max}$ lors de calculs itératifs.

Soient 3 nombres :

$$x = 0.993,$$
$$y = 0.827,$$
$$fGain = 1.9952623.$$

Ces dernières seront définies en simple précision et en double précision afin d'effectuer $n$ fois le calcul suivant en simple et double précisions :

$$x_n = \left( \left( \frac{(x_{n-1} + y)}{\text{Gain}} \right) \times \text{Gain} \right) - y$$

.

Nous devons d'abord définir les erreurs absolue et relatives, $Erreur\_x\_float$ et $Erreur\_x\_float\_rel$ pour chaque itération $n$, comme la différence entre la valeur $x_n$ et la valeur $x$ lorsque le calcul est réalisé avec des variables en simple précision, et, $Erreur\_x\_double$ et $Erreur\_x\_double\_rel$ pour chaque itération $n$, comme la différence entre la valeur $x_n$ et la valeur $x$ lorsque le calcul est réalisé avec des variables en double précision.

*A priori*, on devrait retrouver, pour tout $n$, $x_n = x$. donc l'erreur devrait être nulle pour toutes les itérations.

Or, le programme affiche, pour la première itération du calcul, les valeurs suivantes :

```
Iteration 1
x_f = 0.99300003
Erreur_x_float = 0.0000000596046448
Erreur_x_float_dB -144.5 dB FS
Erreur_x_float_rel = 0.0000000600248167
Erreur_x_float_rel_dB -144.4 dB FS
x_d = 0.9929999999999999
Erreur_x_double = -0.0000000000000001
Erreur_x_double_dB -319.1 dB FS
Erreur_x_double_rel = -0.0000000000000001
Erreur_x_double_rel_dB -319.0 dB FS
```

Et, pour la 20ième itération les valeurs suivantes :

```
Iteration 20
x_f = 0.99300229
Erreur_x_float = 0.0000023245811462
Erreur_x_float_dB -112.7 dBFS
Erreur_x_float_rel = 0.0000023409625101
Erreur_x_float_rel_dB -112.6 dBFS
x_d = 0.9929999999999999
Erreur_x_double = -0.0000000000000001
Erreur_x_double_dB -319.1 dBFS
Erreur_x_double_rel = -0.0000000000000001
Erreur_x_double_dB -319.0 dBFS
```

On peut voir qu'en 20 itérations d'un calcul relativement simple l'erreur relative, commise à chaque itération du calcul en 32 bits flottants, a augmenté d'environ 32 dB. Alors que l'erreur relative, commise en 64 bits flottants n'a pas augmenté et elle est très petite (-319 dB FS).

### Sommation de valeurs en simple et double précision

Lors d'un mixage, pour chaque échantillon, les valeurs associées à chaque piste sont additionnées. Ainsi, les différentes erreurs d'arrondi réalisées à chaque addition peuvent se compenser, mais, elles peuvent aussi s'additionner.

Gaël Martinet a réalisé un programme afin de mettre en évidence les différences qu'il peuvent exister entre une sommation réalisée en simple précision et une autre réalisée en double précision. De plus, une des méthodes donnée par D. Goldberg pour minimiser les erreurs commises lors d'une somme de termes consiste à trier ces derniers de la plus petite à la plus grande valeur absolue avant de les additionner [18].

**Programation du sommateur :** Ce programme génère *nNumberOfIteration* tableaux composés de *nNumberOfValues* nombres aléatoires entre -1 et 1.

Pour chaque itération, les *nNumberOfValues* valeurs sont additionnées :
- en simple précision ;
- en double précision.

Puis, le programme trie les valeurs de la plus petite à la plus grande, et les additionne :
- en simple précision ;
- en double précision.

Dans le même temps, pour chaque itération, il calcule la différence entre :
- la sommation non triée en simple précision et la sommation triée en simple précision ;
- la sommation non triée en double précision et la sommation triée en double précision ;
- la sommation non triée en simple précision et la sommation non triée en double précision ;
- la sommation triée en simple précision et la sommation non triée en double sommation.

Enfin, il affiche la valeur maximum, minimum et moyenne de chaque différence.

18. David Goldberg. « What Every Computer Scientist Should Know About Floating-Point Arithmetic ». Dans : *ACM Computmg Surveys* 23.1 (1991), p. .34

**Analyse des résultats :** Le programme, compilé avec compilateur Intel, avec 128 pistes et calculé sur 48000 échantillons, affiche une différence moyenne entre la version triée et non triée du calcul en précision simple de -118 dB FS , ainsi qu'une valeur maximum de -97 dB FS.

Ce qui veut dire qu'en simple précision, en fonction de l'ordre dans lequel la sommation est réalisée, le résultat est différent. Lorsque le calcul est réalisé en précision double ce problème n'existe pas.

Avec 256 pistes, la différence moyenne entre la version triée et non-triée augmente d'environ 6 dB. Diverses options de compilations ainsi que le compilateur utilisé ne changent que légèrement les résultats (cf. Annexe B.2).

Le même programme, affiche aussi une différence moyenne d'environ -120 dB entre la version simple précision et la version double précision de la sommation, ainsi qu'une valeur maximum de -97 dB.

Cela signifie que lors de la sommation, dans la représentation en virgule flottante, en simple précision, d'un certain nombre de valeurs, des erreurs d'arrondis sont générées. Et, même si la valeur moyenne de ces erreurs absolues est relativement faible (-120 dB FS), certaines d'entre elles peuvent être bien plus importantes. En double précision, ces erreurs sont ramenées à un niveau très faible.

## 2.7 Conversions entre les représentations virgule fixe et virgule flottante

### 2.7.1 virgule fixe $\rightarrow$ virgule flottante

Lors d'un enregistrement audio sur un support numérique, le convertisseur analogique-numérique quantifie le signal sur N bits en virgule fixe, en général 24 bits, et complément à 2.

Ainsi les valeurs représentées peuvent prendre toutes les valeurs entières dans l'intervalle suivant :

$$[-2^{(N-1)}; 2^{(N-1)} - 1].$$

On utilise un *type* de variables dit *entières* nommées *int* en C++ [19] pour stocker les valeurs entières représentant le signal ainsi codé.

Or, les valeurs audio codées en virgule flottante appartiennent généralement à l'intervalle [20]

$$[-1;\ 1].$$

Ainsi pour réaliser la conversion, on divise donc la variable entière associée à chaque échantillon par $2^{(N-1)}$ et on la stocke dans une variable de *type* virgule flottante.

```
long nNumberOfSamples=48000;
int nBits=24;
int nQuantizeSteps=pow(2,nBits-1);
float nInvQuantizeSteps=(float)(1.f/nQuantizeSteps);
int iData[nNumberOfSamples];
float fData[nNumberOfSamples];
for (long j = 0; j < nNumberOfSamples; j++)
   {
      fData[j] = (float)(nInvQuantizeSteps * iData[j
         ]);
   }
```

FIGURE 2.8 – Lignes de code en C++ servant à convertir les valeurs entières du tableau $iData[]$ en des valeurs comprises entre $-1_{10}$ et $1_{10}$ et stockées dans le tableau $fData[]$.

La figure 2.8 montre les lignes de codes servant à convertir les valeurs d'un tableau composé de valeurs entières, $iData[]$, en un tableau, $fData[]$, composé de valeurs codées en virgule flottante.

On comprend bien que cette opération n'est pas "transparente" pour le signal car les valeurs en virgule flottante sont symétriquement réparties autour de $\pm 0$ contrairement aux valeurs entières codées en virgule fixe.

19. Nous étudierons plus en détail les types de données à la section 3.1.1 ainsi que les conversions de types à la section 3.1.2.

20. car le 0 dB FS est représenté par la valeur $\pm 1$

### 2.7.2 virgule flottante → virgule fixe

Le convertisseur numérique-analogique accepte des données quantifiées sur N bits en virgule fixe, en général 24 bits, et complément à 2.

Ainsi, on effectue l'opération inverse puisqu'on multiplie la variable virgule flottante par $2^{(N-1)}$ et qu'on la stocke dans une variable entière. La figure 2.9 montre les lignes de codes servant à convertir les valeurs d'un tableau, $fData[]$, composé de valeurs codées en virgule flottante, en un tableau, $iData[]$, composé de valeurs entières.

```
long nNumberOfSamples=48000;
int nBits=24;
int nQuantizeSteps=pow(2,nBits-1);
float nInvQuantizeSteps=(float)(1.f/nQuantizeSteps);
int iData[nNumberOfSamples];
float fData[nNumberOfSamples];
for (long j = 0; j < nNumberOfSamples; j++)
   {
      iData[j] = (int)(nQuantizeSteps * iData[j]);
   }
```

FIGURE 2.9 – Lignes de code en C++ servant à convertir les valeurs en virgule flottante du tableau $fData[]$ en des valeurs entières comprises entre $-2^{nBits}$ et $2^{nBits} - 1_{10}$ et stockées dans le tableau $iData[]$.

On appelle l'opération de conversion d'une variable représentant des nombres réels, en une variable représentant des nombres entiers, la *troncature* en entiers.

**Troncature en entier :** Comme on l'a vu à la section 2.4.2, avec un mot de N bits, interprété comme un nombre entier relatif complément à 2, on peut exprimer les nombres entiers dans l'intervalle $[-2^{N-1}, 2^{N-1} - 1]$.

Or, le résultat d'une division de deux nombres entiers, par exemple, n'est parfois représentable que par un nombre réel composé d'une partie entière et d'une partie décimale.

Par exemple, avec deux nombres entiers $x$ et $y$ :

$$\begin{aligned} x &= 0011 & x &= 3_{10} \\ y &= 0010 & y &= 2_{10} \\ x \oslash y &= 0001,1 & \frac{x}{y} &= 1,5_{10}. \end{aligned}$$

On ne peut donc pas coder exactement cette valeur avec la représentation binaire des nombres entiers. Ainsi, lors du codage d'un nombre réel composé d'une partie entière et d'une partie décimale par un nombre entier, on ne stocke que la partie entière du nombre réel. On appelle cela la *troncature en entier*.

## 2.8 Conclusion du chapitre

Dans ce chapitre, nous avons compris ce qu'est le codage binaire. Nous connaissons maintenant la différence entre la représentation en virgule fixe et en virgule entière. Nous avons aussi étudié, théoriquement, l'erreur commise par l'arrondi dans la représentation en virgule flottante simple et double précision.

On peut retenir que le codage en virgule flottante permet de représenter des nombres compris dans un intervalle très grand, qui implique que la situation d'*overflow*[21] n'est plus un problème comme cela est expliqué au paragraphe 2.6.2.

De plus, on peut retenir que le codage en virgule flottante permet de représenter des nombres avec une précision qui dépend de la valeur à coder. La valeur de l'erreur absolue est donc dépendante de la valeur codée. Une valeur élevée est codée moins précisément qu'une valeur plus faible. Cela signifie que le rapport signal/bruit est à peu près constant lors du codage de valeurs en virgule flottante et défini par la précision du format utilisé, comme expliqué à la sous-section 2.6.2.

Mais, il faut aussi retenir que les erreurs relatives s'ajoutent ou se compensent tout au long des calculs effectués. Ainsi, l'erreur relative totale peut devenir très importante au bout d'un certain nombre d'opérations et, par voie de conséquence, le rapport signal/bruit peut alors diminuer très sensiblement (cf. Sous-section 2.6.3).

21. correspondant à la situation de saturation numérique appelée aussi *clipping*

# Chapitre 3

# Le plug-in Quantum

Un plug-in est un module externe qui complète un logiciel hôte. Les plug-in destinés à l'audio sont des modules de traitement du signal qui s'insèrent dans un logiciel audio. Plusieurs "formats" de plug-in existent et chaque logiciel est compatible avec un ou plusieurs de ces formats. Le plug-in *Quantum* est disponible pour l'intant au format VST.

Afin d'écouter et de mesurer l'erreur de troncature et l'erreur d'arrondi, j'ai programmé en C++, avec l'aide du framework [1] Flux : :, un *plug-in* permettant d'écouter, en *temps réel*, le résultat de l'un des trois types de traitements programmés.

Le premier traitement est la conversion du signal entrant, codé en 32 bits flottants, en représentation entière [2]. Ce mode réalise donc la troncature en entiers, des échantillons entrants, avec ou sans dithering [3]. On peut alors écouter la version tronquée du signal entrant, le signal original ou bien encore la différence entre la version tronquée et la version originale.

Le deuxième traitement disponible est simplement un gain en dB, suivi d'une atténuation d'autant de dB que voulu ou bien l'inverse c'est-à-dire une amplification. On peut augmenter le nombre d'itérations, correspondant au nombre de fois, par échantillon, que le calcul est réalisé. On peut écouter le résultat de l'opération réalisée en 32 bits flottants, en 64 bits flottants ou bien la différence entre les deux versions du traitement.

1. traduit par *cadre de travail*, désignant un *kit* de composants logiciels servant pour le développement de programmes.
2. c'est-à-dire sans décimales
3. traduit par bruit de blanchissement, c'est un bruit rajouté à un niveau très faible, afin de décorreller l'erreur de troncature.

Le troisième traitement proposé par *Quantum* est un filtrage en cloche à $f$ Hz, avec un facteur de qualité $Q$ et un gain de $\pm g$ dB avec un filtre bi-quadratique. On peut écouter la version traitée en 32 bits flottants, la version traitée en 64 bits flottants, ou bien la différence entre la version traitée en 64 bits flottants et la version traitée en 32 bits flottants.

Ainsi dans ce chapitre, nous commencerons par donner quelques éléments pour comprendre le code C++, puis, pour chaque traitement, nous expliquerons comment il a été programmé, mais nous commenterons et analyserons aussi le résultat sonore.

## 3.1 Eléments de programmation en language C++

### 3.1.1 Le type des données

Le langage C++ utilise des variables, c'est-à-dire des zones de la mémoire, pour stocker les données manipulées. Toute variable utilisée par le code doit être *déclarée*. c'est-à-dire que l'on doit indiquer le nom de la variable et son *type*. Le type indique la nature des données contenues par une variable.

Il existe, en C++, plusieurs types prédéfinis pour les variables représentant des nombres. Ces types correspondent aux différentes représentations binaires des nombres vues au Chapitre 2. On déclare donc une variable avec l'un des types suivants [4] :

- *short*,*int* et *long* pour les nombres entiers signés correspondant respectivement aux codages sur 16, 32 et 64 bits ;
- *float*, *double* et *long double* pour les nombres réels, correspondant respectivement aux précisions simple, double et double étendue.

La figure 3.1 donne un exemple de déclaration de plusieurs variables en C++.

Mais, le programmeur peut définir ses propres types et, pour des raisons que je ne développerai pas ici, utiliser d'autres noms pour utiliser les types de données. Ainsi, le framework Flux : : utilise ses propres noms pour les types de données. Nous utiliserons,

- *Int* pour les entiers ;
- *f32* et *f64* pour les réels représentés en virgule flottante respectivement simple et double précision.

4. Il en existe d'autres mais nous ne les citerons pas ici car nous ne les avons pas utilisés.

```
int mode=0;
long nNumberOfIteration=48000;
float gain=2.5;
double res;
```

FIGURE 3.1 – Exemples de déclaration, en C++, d'une variable. Ici, les variables *mode* et *nNumberOfIteration* représentent des nombres entiers codés respectivement sur 32 et 64 bits, les variables *gain* et *res* représentent des nombres réels exprimés en virgule flottante respectivement en simple précision et double précision.

L'exemple donné par la figure 3.1 est donc équivalent à celui donné par la figure 3.2 qui sera compilé à l'aide du framework Flux : :.

```
Int nNumberOfIteration=48000;
f32 gain=2.5;
f64 res;
```

FIGURE 3.2 – Exemple de déclaration d'une variable, en C++, avec le framework Flux : :. Ici, la variable *nNumberOfIteration* représente un nombre entier, les variables *gain* et *res* représentent des nombres réels exprimés en virgule flottante respectivement en simple précision et double précision.

Des variables peuvent également être déclarées avec un type *SampleType*. Ces données seront alors interprétées selon un type ou un autre "à la demande". Mais, ici, nous utiliserons toujours le type *f64* pour l'ensemble des variables déclarées avec un type *SampleType*.

### 3.1.2 Les conversions de type

Les conversions de types sont appelées opérations de *casting*. Nous utiliserons souvent l'opération de casting la plus simple qui consiste à demander au programme d'interpréter une variable comme étant d'un autre type, et stocker la variable ainsi convertie dans une autre variable du type demandé. Ainsi, la variable initiale reste du type prévu par sa déclaration tandis que la nouvelle variable est égale à la valeur convertie de cette dernière dans un type demandé. D'autres opérations de casting sont possibles mais nous ne les détaillerons pas ici.

La figure 3.3 donne un exemple de la conversion d'une variable de type *f32* en une autre variable de type *int*.

```
//declarations des variables
f64 x_double;
f32 x_float;
Int x_int;
//Conversions
x_float=(f32)x_double;
x_int=(Int)x_double;
```

FIGURE 3.3 – Exemple de conversion de type d'une variable. Ici, la variable $x\_float$ représente la valeur de $x\_double$ qui est un nombre réel exprimé en virgule flottante double précision, mais convertie en virgule flottante simple précision. La variable $x\_int$ est égale à la valeur de $x\_double$ convertie en nombre entier.

### 3.1.3 Les buffers

Afin de réaliser les différents traitements, nous disposons de plusieurs *buffers*, que l'on peut traduire par *mémoires tampons*, qui sont des zones de mémoire permettant de stocker des données de façon temporaire. Ce sont les échantillons qui composent le signal audio-numérique qui sont stockés temporairement dans ces buffers. Quand le signal numérique est lu, les buffers se remplissent d'échantillons et se vident afin de laisser la place aux échantillons suivants.

Nous n'expliquerons pas comment déclarer un buffer car ce n'est pas l'objet de notre sujet. Néanmoins, le plug-in Quantum utilise plusieurs buffers, en premier lieu afin de recevoir les échantillons venant du logiciel hôte puis de renvoyer les échantillons vers le logiciel hôte, mais aussi afin de stocker les résultats de plusieurs

versions d'un même traitement, en parallèle[5]. En effet, il peut être intéressant de pouvoir comparer, en *temps réel*, un même traitement réalisé avec des variables de types f32 et f64, donc codées en 32 bits ou 64 bits virgule flottante.

Dans notre code, le buffer d'entrée est appelé $pfSource$, le buffer de sortie $pfTarget$, mais nous disposons de trois autres buffers : $pf32$ et $pf64$ sont alimentés par pfSource et servent à réaliser en parallèle les traitements en 32 bits flottants et en 64 bits flottants, $pf3264$ sert à stocker le résultat de la différence entre les deux versions du traitement.

La figure 3.4 est un schéma représentant les buffers d'entrée, de sortie, ainsi que les buffers de traitement avant et après que les lignes de codes correspondantes aux opérations de traitement du signal ne soient exécutées.

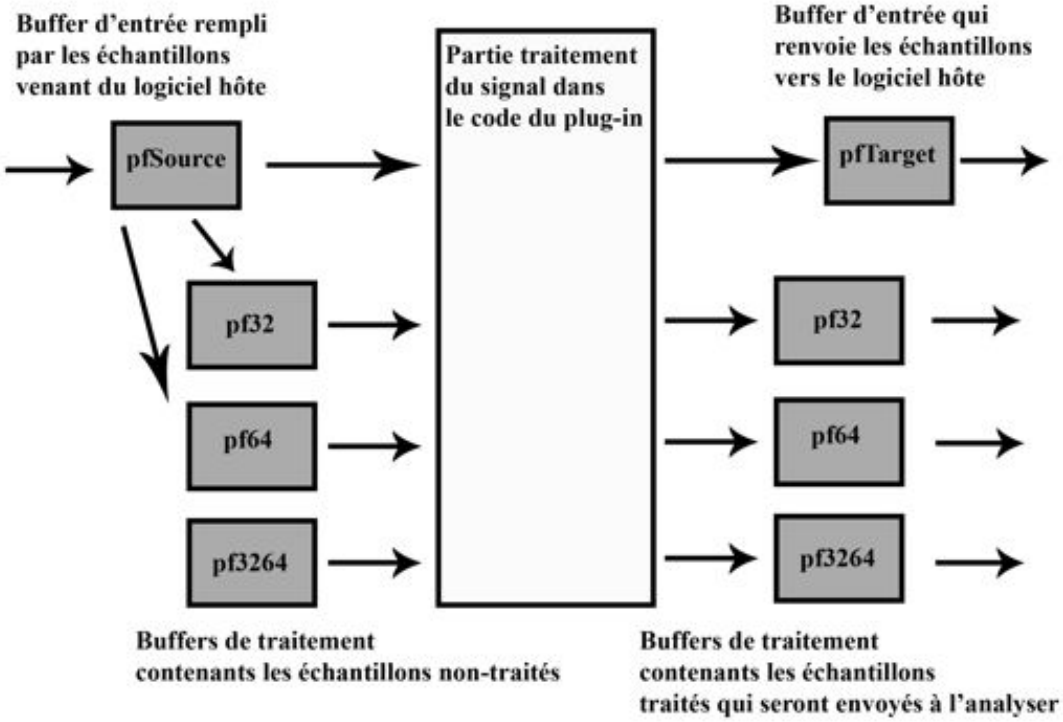


FIGURE 3.4 – Schéma représentant les buffers d'entrée, de sortie, ainsi que les buffers de traitement avant et après que les échantillons entrants ne soient traités par les lignes de codes correspondantes aux opérations de traitement du signal.

5. c'est-à-dire en même temps

### 3.1.4 Partie traitement du signal du programme

Le coeur du programme de traitement du signal se situe à l'intérieur de deux boucles *for(){}* imbriquées qui répètent les opérations contenues entre leurs accolades autant de fois qu'il y a d'échantillons à lire par canal puis pour le nombre de canaux à traiter.

La figure 3.5 montre les lignes du code qui correspondent aux deux boucles *for(){}* imbriquées. Ainsi les variables $pf32[lSample]$, $pf64[lSample]$, $pf3264[lSample]$ et $pfTarget[lSample]$ représentent les valeurs associées aux échantillons numéro $lSample$ du canal actuellement traité par le programme et placés dans les buffers $pf32$, $pf64$, $pf3264$ et $pfTarget$.

```
[...];
for (Int lChannel = 0; lChannel < lNumberOfChannels;
    lChannel++)
{
   for (Int lSample = 0; lSample <
      lNumberOfSamplesPerChannel; lSample++)
   {
      // Partie traitement du signal du programme
      pf32[lSample]=[...];
      pf64[lSample]=[...];
      [...];
      pf3264=[...];
      pfTarget=[...];
   }
}
[...];
```

FIGURE 3.5 – Résumé du code du plug-in Quantum mettant en évidence les deux boucles *for(){}* imbriquées, encadrant la partie traitement du signal, du code. $lChannel$ correspond au numéro du canal qui sera traité par la boucle, $NumberOfChannels$ correspond au nombre de canaux à traiter, $lSample$ correspond au numéro de l'échantillon du canal actuellement traité, qui sera traité par la boucle. $lNumberOfSamplesPerChannel$ correspond au nombre d'échantillons par canal. $lChannel++$ et $lSample++$ signifient que les nombres $lChannel$ et $lSample$ sont incrémentés de 1 à la fin de chaque itération de la boucle.

## 3.2 Troncature lors de la conversion en entiers

Dans ce mode de fonctionnement du plug-in *Quantum*, l'utilisateur dispose de 5 paramètres sur lesquels agir. Ce dernier pourra :

- choisir sur combien de bits tronquer le signal entrant ;
- choisir d'ajouter du dither ou pas ;
- définir l'amplitude du dither en multiple du LSB ;
- choisir d'écouter la version tronquée ou la différence entre la version tronquée et la version originale du signal entrant et de lui appliquer un gain ;
- choisir la valeur en dB du gain à appliquer à la différence entre version tronquée et non tronquée.

### 3.2.1 Programmation de l'opération à réaliser

#### Déclaration des variables définies par l'utilisateur

Dans notre code, les paramètres utilisateurs sont déclarés par les lignes de code de la figure 3.6.

La variable *nQuant* est associée à la valeur donnée par le bouton rotatif de l'interface graphique définissant le nombre de bits sur lequel on veut coder le signal. Elle prend pour valeur un nombre entier compris entre 8 et 32.

*Dither* est la variable associée au bouton nommé *Dither* de l'interface graphique. Elle prend pour valeur 1 ou 0, suivant que le bouton est ou non appuyé, suivant que l'on veut appliquer ou non du dither.

*nDither* est la variable associée à la valeur donnée par le bouton rotatif définissant l'amplitude du dither. Sa valeur est donnée en multiple du LSB.

$ModeSNR$ est la variable associée au bouton *ModeSNR*. Elle prend pour valeur 1 ou 0, suivant que le bouton est ou non enclenché, suivant que l'utilisateur veut écouter la différence entre la version tronquée et la version non tronquée.

$fGain\_dB$ est la variable associée à la valeur en dB du gain à appliquer à la différence. On définit cette dernière grâce au bouton rotatif situé le plus à droite de l'interface graphique.

```
Int nQuant=params.nQuantification;
Int Dither=params.Dither;
SampleType nDither=params.nDither;
Int ModeSNR = params.ModeSNR;
SampleType fGain_dB = params.Gain;
```

FIGURE 3.6 – Déclaration des variables, modifiables par l'utilisateur, utilisées lors de la troncature en entiers proposée par le plug-in Quantum. La déclaration *Int* définit une variable entière et *SampleType* représente, ici, un nombre réel codé en virgule flottante double précision.

### Déclaration des variables de calcul

Comme expliqué à la sous-section 2.7.2, nous devons convertir chaque échantillon entrant, qui représente un nombre réel, codé en virgule flottante, compris dans l'intervalle $[-1, 1]$ en un nombre entier, codé en virgule fixe, compris dans l'intervalle $[-2^{N-1}, 2^{N-1} - 1]$.

Nous devons donc définir deux variables, $f0dB$ et $fInv0dB$, par les égalités suivantes :

$$f0dB = 2^{(nQuant-1)}$$
$$fInv0dB = 2^{-(nQuant-1)}$$

avec $nQuant$, la variable, modifiable par l'utilisateur, correspondant au nombre de bits sur lequel on veut coder nos valeurs entières.

Nous devons aussi calculer le niveau du dither ou bruit de blanchissement en fonction des variables précédentes et des variables représentant les paramètres modifiables par l'utilisateur. Le générateur de bruit blanc qui servira pour ce dither simple requiert uniquement un niveau en dB. On définit donc la variable $fNiveauDither\_dB$, qui sert à configurer le générateur, par l'égalité suivante :

$$fNiveauDither\_dB = 20 \cdot \log_{10}[fInv0dB \cdot nDither].$$

La valeur de la variable gain $fgain\_dB$, applicable pour écouter la différence entre la version tronquée et non tronquée, est donnée par l'utilisateur en dB. Il faut donc la convertir afin de déterminer le facteur $fgain$ par lequel on multipliera les valeurs de nos échantillons. La formule suivante permet cela :

$$fGain = 10^{(fGain\_dB \cdot 0.05)}.$$

La figure 3.7 indique les lignes de code utilisées pour déclarer et définir les variables de calcul dont on vient de parler et qui nous serviront pour les opérations suivantes du code.

```
SampleType f0dB=(SampleType)pow(2.0,(f64)(nQuant -
    1));
SampleType fInv0dB=(SampleType)(1.0/f0dB);
SampleType fNiveauDither_dB=(SampleType)(20.0*log10(
    fInv0dB*nDither);
SampleType fGain = (SampleType)pow(10,fGain\_dB
    *0.05);
```

FIGURE 3.7 – Déclaration et définition des variables de calcul utilisées par la troncature en entier proposée par le plug-in Quantum. Avec $f0dB = 2^{nQuant-1}$, $fInv0dB = 2^{-(nQuant-1)}$, $fNiveauDither\_dB = 20,0 \cdot \log_{10}[fInv0dB * nDither]$ et $fGain = 10^{(fGain\_dB \cdot 0.05)}$.

### Partie traitement du signal

Les échantillons venant du logiciel hôte sont dans les buffers $pf32$ et $pf64$. Les échantillons générés pour le dither doivent y être ajoutés si le paramètre *dither* est à 1, et le résultat de cette addition doit être multiplié par $f0dB$ puis tronqué en valeur entière. Enfin, nous multiplions le résultat de ces opérations par $fInv0dB$ afin de revenir dans le format flottant pour comparer la version traitée et la version originale. La figure 3.8 montre les lignes de code correspondant à ces opérations.

Avant l'exécution des lignes de codes de la figure 3.8, les valeurs associées aux échantillons $pf32[lSample]$ et $pf64[lSample]$ étaient égales à celles des échantillons $pfSource[lSample]$. Mais, dans ces lignes de code, nous indiquons que $pf32[lSample]$ et $pf64[lSample]$ doivent être maintenant remplacés par les résultats des calculs décrits au paragraphe précédent.

On doit ensuite renvoyer les échantillons ainsi traités vers le logiciel hôte en passant par le buffer de sortie $pfTarget$. Mais, nous voulons pouvoir écouter et analyser la différence entre le signal traité et le signal original lorsque l'utilisateur appuiera sur le bouton $ModeSNR$. Nous avons donc utilisé les structures conditionnelles *if(){}else{}*. Ainsi les lignes reproduites à la figure 3.9, indiquent que si la variable $ModeSNR$, correspondant à l'état du bouton du même nom, est égale à 1, la valeur de l'échantillon actuellement traité $pfTarget[lSample]$ du buffer de

```
pf32[lSample]=(f32)((Int)((pf32[lSample]+Dither*
    pf32WhiteNoise[lSample])*(f32)f0db)*(f32)fInv0db);
pf64[lSample]=(f64)((Int)((pf64[lSample]+Dither*
    pf64WhiteNoise[lSample])*f0db)*fInv0db);
```

FIGURE 3.8 – Ligne de code du plug-in Quantum correspondant au calcul de la troncature, en entier codé sur $nQuant$ bits, du signal entrant. Avec $pf32[lSample]$ et $pf64[lSample]$ les échantillons des buffers de traitement en 32 et 64 bits virgule flottante.

sortie est égale à la différence des valeurs $pf(32)[lSample]$ et $pfSource[lSample]$, associées aux échantillons tronqués et non traités.

```
if (ModeSNR == 1)
{
    [...]
    pfTarget[lSample] = (SampleType)(((SampleType)
        pf32[lSample]-pfSource[lSample])*(1+fGain_dB));
    [...]
}
else
{
    [...]
    pfTarget[lSample] = (SampleType)pf32[lSample];
    [...]
}
```

FIGURE 3.9 – Lignes de code du plug-in Quantum correspondant à la sélection par l'utilisateur de ce qu'il souhaite écouter lorsqu'il est dans le mode troncature en entier. Par exemple, si le bouton $ModeSNR$ est appuyé, les échantillons, $pfTarget[lSample]$, du buffer de sortie sont égaux à $pf32[lSample]$-$pfSource[lSample]$, c'est-à-dire les échantillons du buffer de traitement $pf32$, moins les échantillons non traités. Mais, si la variable $ModeSNR$ est différente de 1, alors les échantillons $pfTarget[lSample]$, du buffer de sortie sont les échantillons du buffer $pf32$.

### 3.2.2 Analyse et écoute du résultat du traitement

Ensuite on lance le plug-in en mode troncature, et on affiche les données dans l'analyseur Flux : :.

La figure 3.10 montre le résultat de la troncature en entiers sur 16 bits d'une sinusoïdale sans dithering. On y voit les harmoniques générées par l'opération de troncature. La figure suivante 3.11 montre la même troncature mais cette fois avec dithering. Cette fois, le bruit blanc masque presque toutes les harmoniques générées par la troncature en entier.

Lorsqu'on appuie sur le bouton $ModeSNR$ du plug-in, et que l'on écoute donc la différence entre la version tronquée et la version non tronquée des valeurs associées aux échantillons du signal entrant sans dithering, on se rend bien compte que l'erreur générée lors de l'opération de troncature est corrélée au signal entrant : on entend des composantes sonores du signal entrant. Cette corrélation implique que ce bruit peut être appelé distorsion.

Lorsque qu'on ajoute du dither, l'erreur de troncature n'est plus corrélée au signal entrant, du moins nettement moins. En effet, on entend bien le bruit de blanchissement lorsqu'on écoute la différence entre la version tronquée et non tronquée avec dithering, mais beaucoup moins, voire plus du tout, les composantes du signal entrant.

Si j'essaye de donner mon ressenti personnel, je dirais que, même en ayant écouté la distorsion apportée par la troncature en entier sur 16 bits d'un signal sonore, il est assez difficile d'entendre une différence en passant de la version tronquée avec dithering à celle sans dithering.

Le bruit généré lors de la troncature en entier est un sujet maintes fois débattu. Mais je trouvais intéressant d'écouter ce dernier justement car il est *a priori* mieux connu que les bruit générés par l'arrondi lors du calcul en virgule flottante.

Le mode suivant proposé par le plug-in Quantum traite justement cette question avec l'implémentation d'un simple gain et d'une atténuation équivalente.

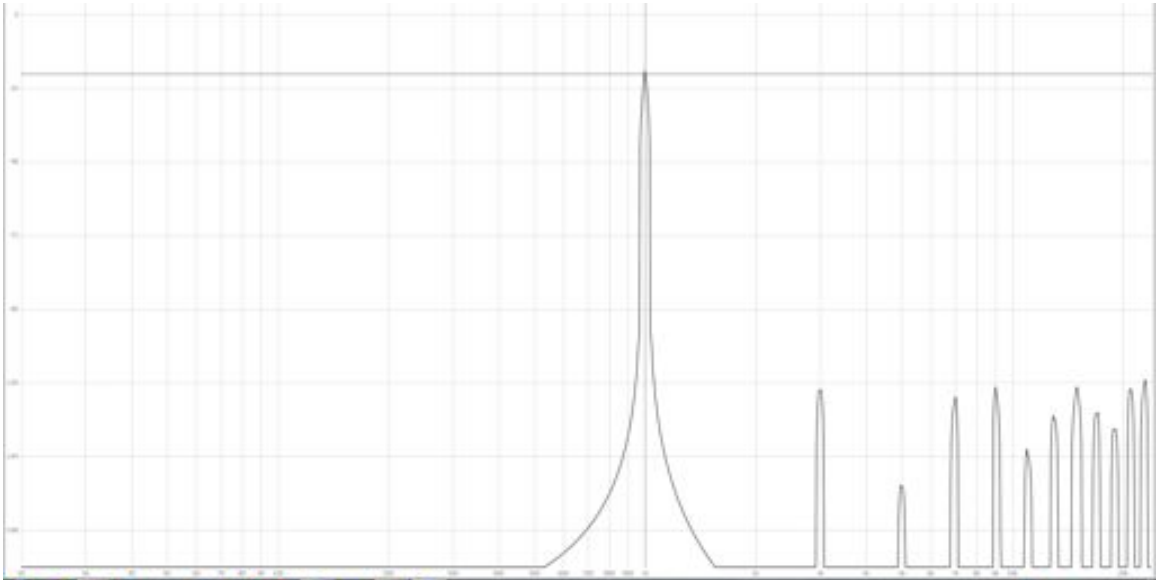

FIGURE 3.10 – Troncature en valeurs entières codées sur 16 bits, sans dithering, d'une sinusoïdale de fréquence 1 kHz.

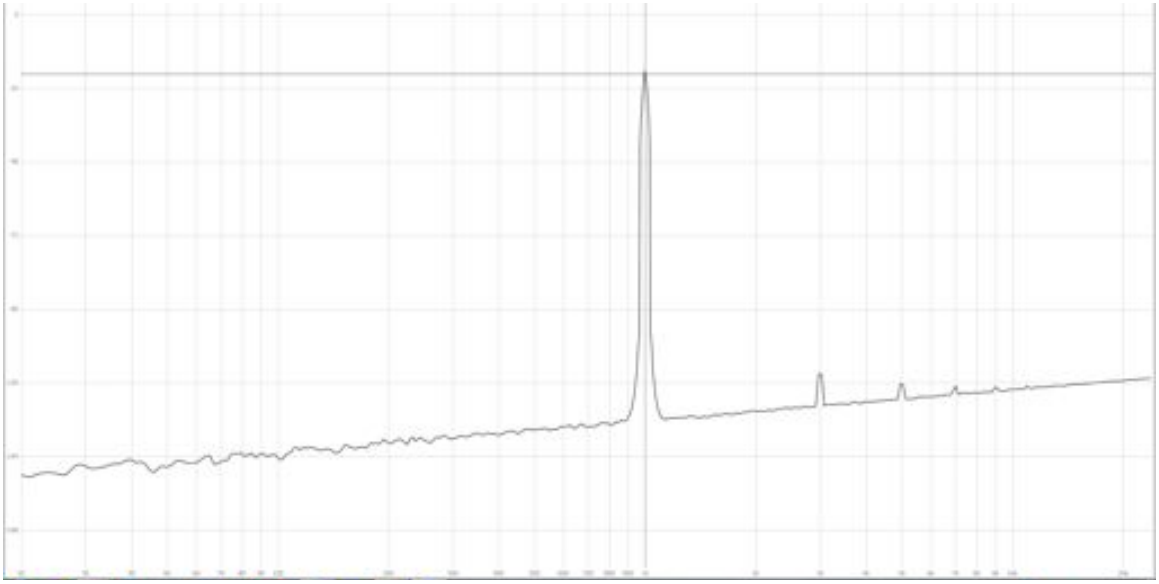

FIGURE 3.11 – Troncature en valeurs entières codées sur 16 bits, avec dithering, d'une sinusoïdale de fréquence 1 kHz.

## 3.3 Erreurs d'arrondi lors d'un gain

Lorsqu'on passe dans le mode *itération*, le programme réalise $n$ fois le calcul suivant :

$$x_n = \frac{x_{n-1} \cdot Gain}{Gain} \tag{3.1}$$

avec $x_n$ la valeur de la variable $x$, la $n$ième fois que le calcul est réalisé, et, $x_{n-1}$, la valeur de la variable $x$, lors de la $(n-1)$ième fois que le calcul est réalisé.

L'utilisateur peut donc agir sur 5 paramètres. Il peut :
- régler la valeur du gain ;
- régler le nombre d'itérations ou le nombre de fois que le calcul de l'équation (3.1) est réalisé pour chaque échantillon ;
- écouter la version 32 bits flottants ou 64 bits flottants du traitement ;
- écouter la différence entre la version du traitement réalisée en simple précision et celle réalisée en double précision ;
- appliquer un gain pour l'écoute de la différence entre les deux versions du traitement.

### 3.3.1 Programmation de l'opération à réaliser

#### Déclaration des variables définies par l'utilisateur

Les variables modifiables par l'utilisateur du plug-in dans ce mode sont déclarées par les lignes de codes de la figure 3.12.

La variable $fGain\_iter\_dB$ est associée à la valeur donnée par le bouton rotatif de l'interface graphique définissant le gain en dB. Elle prend pour valeur un nombre entier compris entre -24 et 24 dB.

$nNumberOfIteration$ est la variable associée au bouton nommé *Itérations* de l'interface graphique. Elle prend des valeurs entre 1 et 80.

$Mode64$ est la variable associée à la valeur donnée par le bouton *Mode32/64* de l'interface graphique. Elle prend la valeur 1 ou 0 suivant que le bouton est appuyé ou non, suivant que l'utilisateur veut ou non écouter la version 64 bits flottants du traitement.

$Diffe$ est la variable associée au bouton *Diff* de l'interface graphique. Elle prend pour valeur 1 ou 0, suivant que le bouton est enclenché ou non, suivant que l'utilisateur veut ou non écouter la différence entre la version 32 bits flottants et la version 64 bits flottants du traitement.

$fGain\_dB$ est la variable associée à la valeur en dB du gain à appliquer à la différence. On définit cette dernière grâce au bouton rotatif situé le plus à droite de l'interface graphique. On l'a déjà déclarée à la section 3.2.

```
Int nNumberOfIteration = params.nIterate;
SampleType fGain_iter_dB = params.Gain_iter;
Int Mode64 = params.Mode64;
Int Diffe = params.Diff;
SampleType fGain = params.Gain;
```

FIGURE 3.12 – Déclaration des variables, modifiables par l'utilisateur, utilisées lors du calcul d'un gain en dB puis d'une atténuation d'autant. La déclaration *Int* définit une variable entière et *SampleType* représente, ici, un nombre réel codé en virgule flottante double précision.

### Déclaration des variables de calcul

La valeur de la variable $fgain\_iter\_dB$, est donnée par l'utilisateur en dB. Il faut donc la convertir afin de déterminer le facteur $fgain\_iter$ par lequel on multipliera les valeurs de nos échantillons. La formule suivante permet cela :

$$fgain\_iter = 10^{(fgain\_iter\_dB \cdot 0.05)}.$$

Plutôt que de diviser par la valeur du facteur $fgain\_iter$, on préfère calculer l'inverse de cette valeur nommée $fInvGain\_iterdB$ qui sera multiplié à la valeur des échantillons, ceci pour des raisons d'économie de ressources de calcul puisque la division est beaucoup plus gourmande en ressources que la multiplication par un nombre entre 0 et 1.

Ainsi, l'équation (3.1) devient :

$$x_n = (x_{n-1} \cdot fGain\_iter) \cdot fInvGain\_iter \tag{3.2}$$

avec $fInvGain\_iter = {}^{1}/_{fGain\_iter}$.

Nous réutiliserons la variable $fGain$ définie à la section 3.2, mais, elle servira cette fois à appliquer un gain sur la différence entre la version du traitement en 32 bits et en 64 bits virgule flottante.

La figure 3.13 indique les lignes de code utilisées pour déclarer et définir les variables de calcul dont on vient de parler et qui nous serviront pour les opérations suivantes du programme.

```
SampleType fGain_iter = (SampleType)pow(10,
    fGain_iter_dB*0.05);
SampleType fInvGain_iter= (SampleType)1.0/fGain_iter
    ;
SampleType fGain = (SampleType)pow(10,fGain_dB*0.05)
    ;
```

FIGURE 3.13 – Déclaration et définition des variables de calcul utilisées par le traitement proposé par le plug-in Quantum en mode *itération*. $fGain\_iter$ est la valeur du facteur qui servira à multiplier les valeurs des échantillons et $fInvGain\_iter$ est la valeur du facteur inverse qui servira pour le traitement du signal en mode *itération*.

### Partie traitement du signal

Les échantillons venant du logiciel hôte sont dans les buffers $pf32$ et $pf64$. Ils doivent être multipliés par la valeur de la variable $fGain\_iter$ puis être multipliés par la valeur de la variable $fInvGain\_iter$. Ce cycle d'opérations devra être répété $nNumberOfIteration$ fois. Pour programmer la répétition d'opération, on utilise une boucle *for(){}*. Enfin, nous stockerons dans le buffer $pf3264$, la différence entre le traitement réalisé en virgule flottante simple et double précision, soient 32 et 64 bits. La figure 3.14 montre les lignes de code correspondant à ces opérations.

Avant l'exécution des lignes de codes la figure 3.14, les valeurs associées aux échantillons $pf32[lSample]$ et $pf64[lSample]$ étaient égales à celles des échantillons $pfSource[lSample]$. Mais, dans ces lignes de code, nous indiquons que $pf32[lSample]$ et $pf64[lSample]$ doivent être maintenant remplacés par les résultats des calculs décrits au paragraphe précédant.

On doit ensuite renvoyer les échantillons ainsi traités vers le logiciel hôte en passant par le buffer de sortie $pfTarget$. Mais, nous voulons pouvoir écouter et

```
for (Int j = 0; j < nNumberOfIteration; j++)
{
    pf32[lSample] *= (f32)fGain_iterdB;
    pf32[lSample] *= (f32)fInvGain_iterdB;

    pf64[lSample] *= (f64)fGain_iterdB;
    pf64[lSample] *= (f64)fInvGain_iterdB;
}
pf3264[lSample] =  (pf64[lSample] - (f64)(pf32[
    lSample]));
```

FIGURE 3.14 – Lignes de code du plug-in Quantum correspondant à l'application d'un gain sur le signal entrant suivi d'une atténuation de même valeur, calculée 32 et 64 bits virgule flottante. Avec $pf32[lSample]$ et $pf64[lSample]$ les valeurs des échantillons des buffers de traitement en 32 et 64 bits virgule flottante, et $pf3264[lSample]$, la valeur de l'échantillon actuellement traité du buffer $pf3264$.

analyser la différence entre le signal traité en 32 bit virgule flottante, ou simple précision, et le signal traité en 64 bit virgule flottante, ou double précision, lorsque l'utilisateur appuiera sur le bouton $Diff$. Nous utilisons donc les structures conditionnelles *if(){}else{}*. Ainsi, les lignes reproduites à la figure 3.15, indiquent que si, par exemple, la variable $Diffe$ est égale à 1, alors la valeur de l'échantillon actuellement traité $pfTarget[lSample]$ du buffer de sortie est égale à la valeur $pf3264[lSample]$. En revanche, si la variable $Diffe$ est égale à 0 et que la variable $Mode64$ est égale à 0, alors la valeur associée à l'échantillon actuellement traité $pfTarget[lSample]$ du buffer de sortie est égale à $pf32[lSample]$.

### 3.3.2 Analyse et écoute du résultat du traitement

Lorsqu'on appuie sur le bouton $Diff$ du plug-in, et que l'on écoute donc la différence entre les versions traitées en virgule flottante simple et double précisions du signal entrant, il nous faut augmenter de beaucoup le gain d'écoute situé le plus à droite de l'interface graphique et manipuler le bouton correspondant à la variable $fGain_iter\_dB$. On entend alors un bruit qui ressemble la plupart du temps à un *bruit blanc*, d'un niveau très faible, mais, pour certaines valeurs de $fGain\_iter$ multiple de 6 dB, le niveau de la différence entre les deux traitements augmente et l'on se met à entendre des composantes sonores du signal entrant. Cette corrélation implique que ce bruit peut parfois être appelé *distorsion*.

```
if (Diffe == 1)
{
    pfTarget[lSample] = (SampleType)(pf3264[lSample]
        * (1.0 + fGain));
}
else
{
    if (Mode64 == 0)
    {
        pfTarget[lSample] = (SampleType)pf32[lSample]
    }
    else
    {
        pfTarget[lSample] = (SampleType)pf64[lSample];
    }
}
```

FIGURE 3.15 – Lignes de code du plug-in Quantum correspondant à la sélection par l'utilisateur de ce qu'il souhaite écouter lorsqu'il est dans le mode *iteration*. Avec $pfTarget[lSample]$, la valeur de l'échantillon du buffer de sortie actuellement traité, $pf32[lSample]$, $pf64[lSample]$ et$pf3264[lSample]$ les valeurs associées aux échantillons, actuellement traités, des buffers de traitement.

Lorsqu'on augmente $nNumberOfIteration$, le nombre d'itérations du calcul, le niveau du bruit augmente. Cela signifie qu'au bout d'un grand nombre d'opérations, le niveau de l'erreur générée par l'arrondi en 32 bits flottants des valeurs utilisées augmente et est susceptible d'atteindre un niveau suffisant pour être potentiellement *audible*. Mais, il faudrait un grand nombre d'opérations, or, certains traitements, comme la convolution par exemple, impliquent beaucoup d'opérations sur les valeurs associées aux échantillons. On peut donc supposer qu'un moteur de convolution, effectuant ses opérations avec des variables codées en 64 bits flottants, générera moins d'erreurs d'arrondis qu'un autre qui travaille avec des variables codées en 32 bits flottants.

## 3.4 Erreurs d'arrondi lors d'un filtrage

On passe dans le mode *filtrage*, en appuyant sur le bouton *filter* de l'interface graphique du plug-in.

Ce dernier réalise alors le filtrage du signal d'entrée à l'aide d'un numérique filtre bi-quadratique simple dont la fréquence, le facteur de qualité et le gain, sont réglables par l'utilisateur.

Un filtre bi-quadratique est un filtre RII, soit à réponse impulsionnelle infinie, dont l'implémentation peut prendre la forme montrée par la figure 3.16 et par l'équation suivante :

$$s[n] = b_0 \cdot e[n] + b_1 \cdot e[n-1] + b_2 \cdot e[n-2] - a_0 \cdot s[n-1] + a_1 \cdot s[n-2]. \quad (3.3)$$

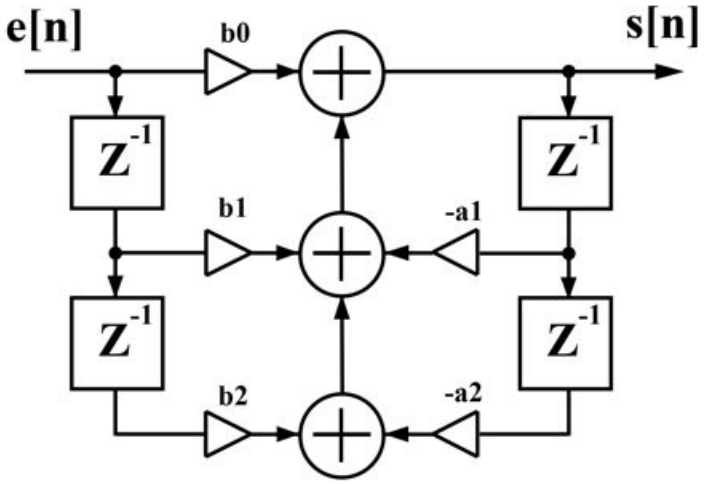


FIGURE 3.16 – Schéma d'une des formes d'implémentation que peut prendre un filtre bi-quadratique. $e[n]$ représente les échantillons entrants du filtre, $s[n]$ représente les échantillons sortants du filtre, les rectangles $Z^{-1}$ représentent les cellules de retard du filtre, et les triangles blancs représentent les multiplications par les coefficients du filtre.

On comprend bien que les valeurs des échantillons de sortie $s[n]$ dépendent des valeurs des échantillons de sortie précédents. Il est donc possible que les erreurs commises pour chaque échantillon de sortie s'amplifient comme expliqué à la sous-section 3.3.

Nous souhaitons donc pouvoir comparer les deux versions d'un même filtre biquadratique, l'une calculée avec des variables codées en 32 bits virgule flottante, l'autre en 64 bits virgule flottante.

Dans ce mode de fonctionnement du plug-in *Quantum*, l'utilisateur dispose de 7 paramètres sur lesquels agir. Ce dernier pourra :

- définir la fréquence du filtre ;
- définir le facteur de qualité du filtre ;
- définir le gain du filtre ;
- définir combien de fois le filtrage sera appliqué sur les échantillons entrants ;
- choisir d'écouter la version traitée en 32 ou 64 bits flottants ;
- choisir d'écouter la différence entre les deux versions du filtrage ;
- choisir la valeur en dB du gain à appliquer à la différence entre les deux versions du filtrage pour mieux l'écouter.

### 3.4.1 Programmation de l'opération à réaliser

#### Déclaration des variables définies par l'utilisateur

Les variables modifiables par l'utilisateur du plug-in dans ce mode sont déclarées par les lignes de codes de la figure 3.17. Un certain nombre de ces variables sont aussi utilisées dans le mode *itération* du plug-in. Nous les avons donc déjà définies à la sous-section 3.3.1. Cependant, deux variables modifiables par l'utilisateur ne sont utilisées que dans le mode *flitrage*.

La figure 3.17 rassemble les lignes de codes déclarant les variables modifiables par l'utilisateur.

La variable $fFreq$ est associée à la valeur donnée par le bouton rotatif de l'interface graphique définissant la fréquence du filtrage en Hz. Elle prend des valeurs comprises entre -5 et 20000 Hz.

```
SampleType fFreq = params.Freq;
SampleType fGain_iter_dB = params.Gain_iter;
SampleType Q_Iter = params.Q_Iter;
Int nNumberOfIteration = params.nIterate;
Int Mode64 = params.Mode64;
Int Diffe = params.Diff;
SampleType fGain = params.Gain;
```

FIGURE 3.17 – Déclaration des variables, modifiables par l'utilisateur, utilisées dans le mode filtrage. La déclaration *Int* définit une variable entière et *SampleType* représente, ici, une variable réelle codée en virgule flottante double précision.

La variable $Q\_iter$ est associée à la valeur donnée par le bouton rotatif de l'interface graphique définissant le facteur de qualité du filtre. Elle prend des valeurs comprises entre 1 et 100.

**Partie traitement du signal**

Les fonctions de filtrage ayant déjà été programmées par Gaël Martinet nous ne les détaillerons pas ici. Nous avons dû néanmoins les initialiser et instancier autant de bandes de filtres que nécessaires.
Les lignes de codes de la figure 3.18 explicitent cette opération.

Les échantillons venant du logiciel hôte sont dans les buffers $pf32$ et $pf64$. Ils doivent être envoyés dans les fonctions de filtrage correspondantes, $m_f32EqFilter$ et $m_f32EqFilter$, par les lignes de codes décrites à la figure 3.19.
Les résultats des filtrages sont stockés dans ces même buffers. Enfin, nous stockerons dans le buffer $pf3264$, la différence entre le traitement réalisé en virgule flottante simple et double précision, soient 32 et 64 bits, de la même façon qu'à la section 3.3.

Il faut maintenant renvoyer les résultats des traitements vers le buffer de sortie $pfTarget$ afin de les écouter. Pour cela, nous allons réutiliser les lignes de codes de la figure 3.15 donnée à la section 3.3 .

Ainsi, quand le bouton $Diff$ est appuyé, l'utilisateur écoute la différence entre les deux versions du traitement. Quand il est relâché et que le bouton $Mode64$ est appuyé, l'utilisateur écoute la version 64 bits du traitement. Lorsque aucun des boutons précédents n'est appuyé, l'utilisateur écoute la version 32 bits du traitement.

```
for (Int i = 0; i < lNumberOfChannels; i++)
{
    m_f32EqFilter.SetNumberOfBands(i,
        nNumberOfIteration);
    m_f64EqFilter.SetNumberOfBands(i,
        nNumberOfIteration);

    for (Int j = 0; j < nNumberOfIteration; j ++)
    {
        m_f32EqFilter.Set(i, j, EqBT_PeakFilter,
            fSamplingRate, fFreq, Q_Iter, fGain_iter);
        m_f64EqFilter.Set(i, j, EqBT_PeakFilter,
            fSamplingRate, fFreq, Q_Iter, fGain_iter);
    }
}
```

FIGURE 3.18 – Lignes de code correspondant à l'initialisation de $nNumberOfIteration$ bandes de filtres codées en 32 et 64 bits virgule flottante, dans le mode*filtrage* du plug-in Quantum. La fréquence, le facteur de qualité et le gain des filtres sont définis par les variables, $fFreq$, $Q\_iter$ et $fGain\_iter$, paramétrables par l'utilisateur.

```
m_f32EqFilter.Process(ppf32Source, lNumberOfChannels
    , lNumberOfSamplesPerChannel, ppf32Source,
    lNumberOfChannels,lNumberOfSamplesPerChannel,
    false);
m_f64EqFilter.Process(ppf64Source, lNumberOfChannels
    , lNumberOfSamplesPerChannel, ppf64Source,
    lNumberOfChannels, lNumberOfSamplesPerChannel,
    false);
```

FIGURE 3.19 – Lignes de code du plug-in Quantum correspondant à l'appel des fonctions de filtrage afin qu'elles traitent les échantillons entrants, contenus dans les buffers $pf32$ et $pf64$.

### Analyse et écoute du résultat du traitement

Grâce à la fonction *sample grabber* du framework Flux : :, nous pouvons réaliser, en *temps réel* sur l'analyseur Flux : :, une analyse fréquentielle des trois buffers de traitement, $pf32$, $pf64$ et $pf3264$, et les afficher en même temps sur un même graphique. Nous avons donc réalisé des captures d'écran pour montrer quelques courbes relativement intéressantes.

La figure 3.20 représente le spectre fréquentiel d'un bruit rose filtré en 32 bits flottants ainsi que celui de la différence entre les résultats du filtrage réalisé en 32 bits et les échantillons du filtrage réalisé en 64 bits flottants, soient les données contenues dans le buffer $pf3264$. On comprend que les erreurs absolues commises lors d'un filtrage, relativement simple, réalisé en 32 bits flottants, d'un signal audio, sont assez importantes.

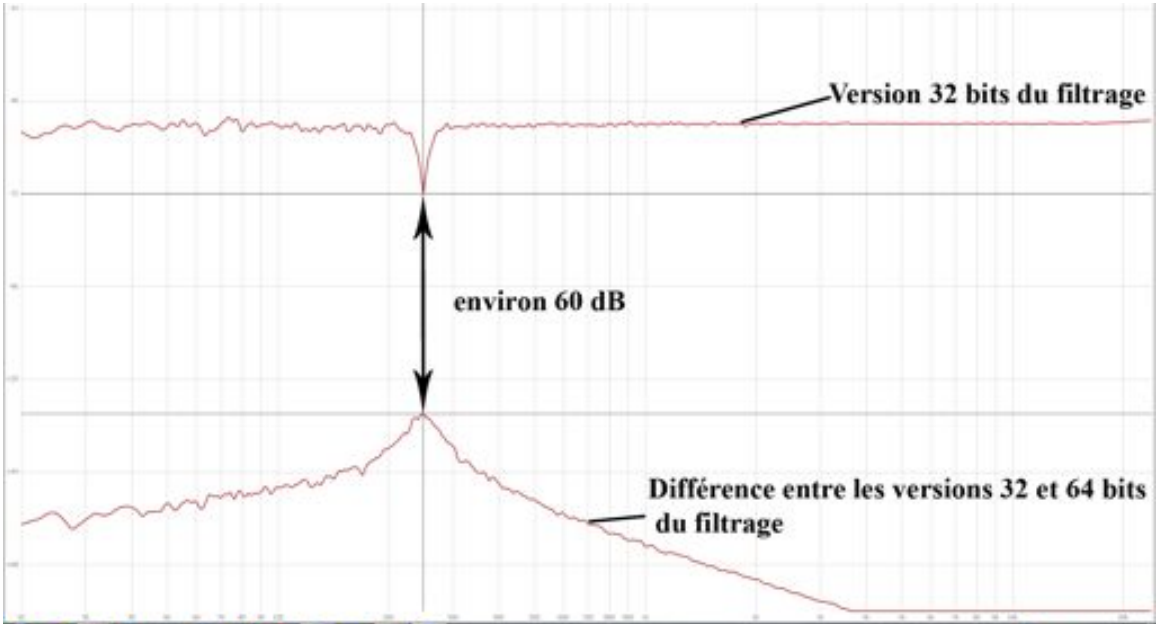


FIGURE 3.20 – Analyse spectrale du filtrage d'un bruit rose par un filtre biquadratique. La fréquence du filtrage à été réglée à 250 Hz, le facteur de qualité est de 100, et le gain du filtre à été fixé à 24 dB. La fréquence d'échantillonnage est de 48 kHz.

Et, plus on diminue la fréquence du filtre, plus l'erreur est importante. La figure 3.21 montre la même analyse mais réalisée avec la fréquence des filtres réglée sur 30 Hz.

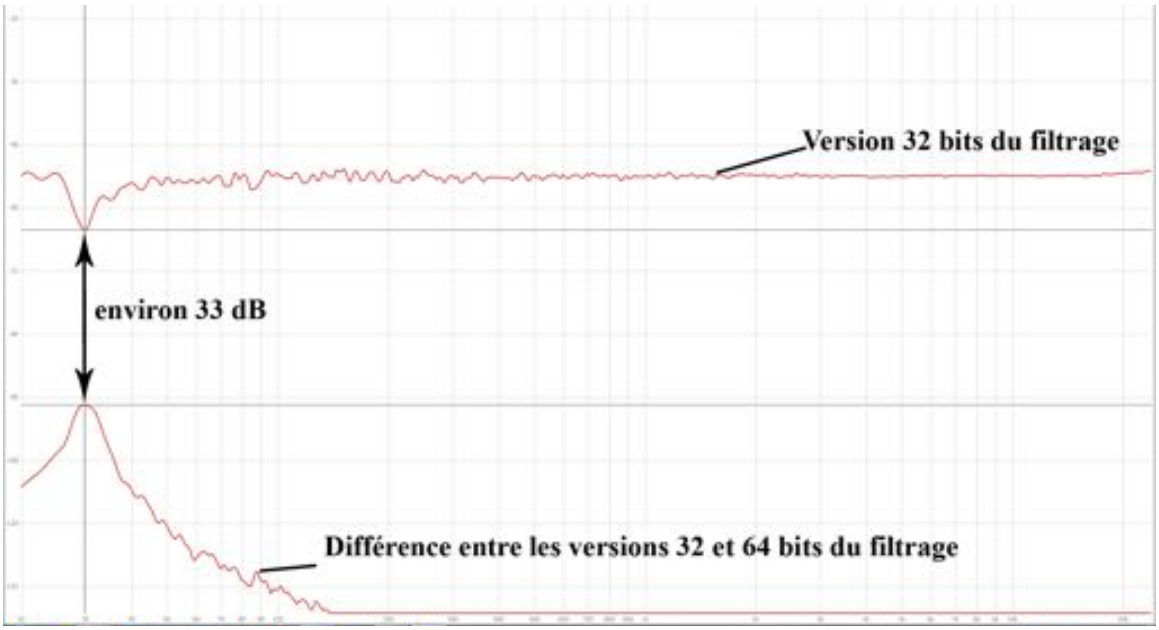


FIGURE 3.21 – Analyse spectrale du filtrage d'un bruit rose par un filtre biquadratique. La fréquence du filtrage à été réglée à 30 Hz, le facteur de qualité est de 100, et le gain du filtre à été fixé à 24 dB. La fréquence d'échantillonnage est de 48 kHz.

De même, le fait d'augmenter la fréquence d'échantillonnage rend le filtre encore plus *instable*. La figure 3.22 montre le même filtrage que la figure 3.21 mais avec une fréquence d'échantillonnage de 96 kHz.

D'autres captures d'écran sont visibles sur le cd de données accompagnant le mémoire.

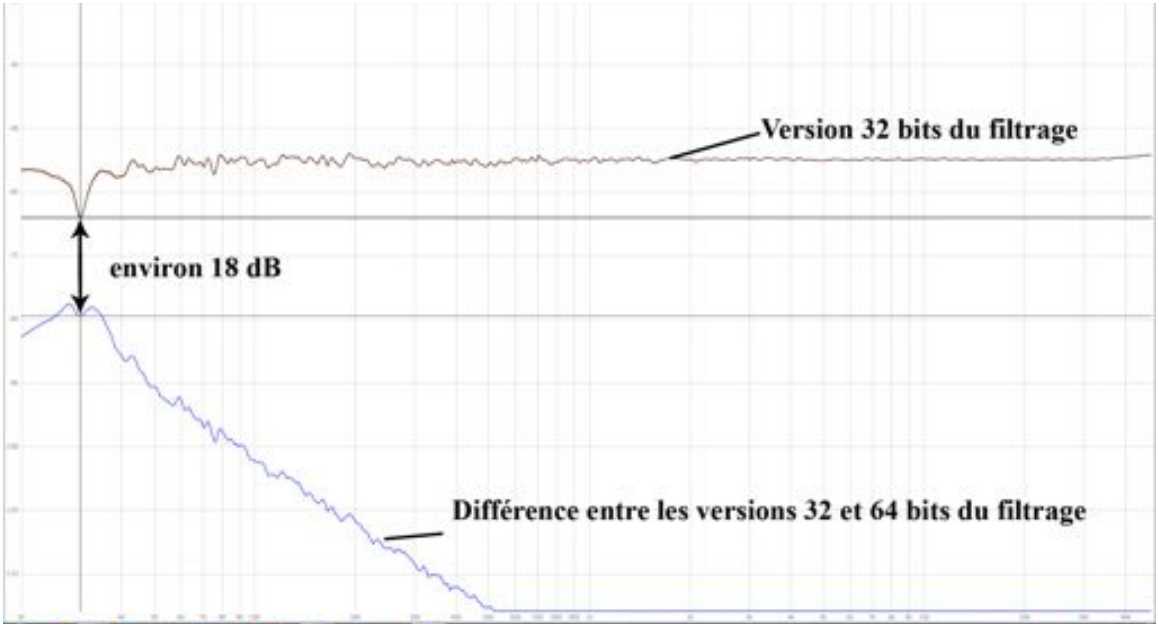


FIGURE 3.22 – Analyse spectrale du filtrage d'un bruit rose par un filtre bi-quadratique. La fréquence du filtrage à été régléé à 30Hz, le facteur de qualité est de 100, et le gain du filtre à été fixé à 24 dB. La fréquence d'échantillonnage est de 96 kHz.

Si l'on écoute maintenant, avec de la musique, ces erreurs d'arrondi[6], on se rend compte qu'elles sont corrélées au signal entrant. On comprend aussi que, dans notre cas, comme les figures 3.21 et 3.20 le suggèrent, les erreurs sont beaucoup plus importantes dans les basses fréquences que dans les hautes fréquences. Cela est dû à la structure même du filtre bi-quadratique, mais nous ne rentrerons pas dans les détails concernant cette question. Néanmoins, on peut dire qu'il existe d'autres filtres qui sont moins *instables* dans les basses fréquences.

Le fait que, lors d'un filtrage, les échantillons déjà traités soient *ré-injectés* dans le calcul, implique que les erreurs commises lors des opérations arithmétiques, utilisées par le filtre, peuvent s'amplifier et ainsi faire diminuer le rapport signal/bruit.

6. en appuyant sur bouton $Diff$ de l'interface graphique du plug-in,

Le filtrage proposé par le plug-in Quantum est simple, néanmoins on peut imaginer que des filtrages plus compliqués, ou des traitements qui utilisent un grand nombre d'opérations arithmétiques, ainsi que des boucles de rétroaction, comme une réverbération par exemple, génèrent des erreurs d'arrondis qui peuvent être relativement importantes[7] en 32 bits flottants. Et, par suite, ces erreurs seront beaucoup moins importantes si ces traitements sont réalisés en utilisant des traitements codés en 64 bits flottants.

La question est de savoir si cette diminution est significative et si les différences, entre un traitement en 32 bits flottants et le même traitement en 64 bits flottants, s'entendent. Nous ne répondrons pas de manière scientifique à cette question dans ce mémoire mais la question mériterait probablement la mise en place de tests subjectifs.

## 3.5 Conclusion du chapitre

Grâce au plug-in Quantum, nous avons pu[8] écouter et analyser les différents bruits générés par les erreurs commises par la troncature ou l'arrondi du signal entrant lors : de la conversion virgule flottante vers virgule fixe ; d'un gain suivi d'une atténuation codés en virgule flottante ; ou d'un filtrage à l'aide d'un filtre bi-quadratique. Nous en savons donc un peu plus sur les bruits susceptibles d'être générés lors d'un mixage.

Ce plug-in pourrait être amélioré dans le futur, déjà grâce à une meilleure interface graphique, mais aussi par l'amélioration de l'algorithme de dithering, ainsi qu'une optimisation du code.

7. par rapport au signal traité.
8. et nous pouvons encore,

# Chapitre 4

# Tests comparatifs de sommateurs numériques et de traitements du signal audio

Dans ce chapitre, nous nous intéressons aux tests que nous avons réalisés. Dans un premier temps, nous avons effectué un mixage test, composé de 128 pistes de bruit rose, sur plusieurs logiciels de mixage audio-numérique afin les comparer. La première section de ce chapitre est dédiée au détail du protocole de test utilisé, puis aux analyses des fichiers créés par chacun des logiciels testés.

Dans un deuxième temps, nous détaillons le protocole, et les analyses d'un autre test, qui compare cette fois l'utilisation de la simple ou double précision, de la représentation binaire en virgule flottante, pour le traitement de signaux audio-numériques : filtres, compresseurs, réverbération, etc. . . ; lors d'un même mixage musical.

## 4.1 Comparaison de sommateurs numériques

Dans cette section, nous définissons un protocole, que nous avons appliqué, pour répondre à la question suivante : "Existe-t-il des différences entre la sommation de $n$ fichiers audio-numériques réalisée par un logiciel $A$ et celle réalisée par un logiciel $B$ ?". Nous répondrons de manière partielle car nous nous limiterons dans ce test au cas simple où aucun (ou presque) traitement n'est réalisé sur les signaux à sommer : pas de gain, tous les *panoramiques* au centre [1].

Nous commencerons donc par faire la liste des logiciels testés, et nous définirons un logiciel référence pour nos tests. Puis nous détaillerons le protocole utilisé. Enfin, nous analyserons ensuite les résultats, en montrant les courbes des analyses spectrales de l'analyseur Flux : :, mais, aussi en considérant les résultats (bruts) de l'analyse IDS proposée par Laurent Millot.

### 4.1.1 Choix des logiciels à tester

Nous avons choisi 9 logiciels différents pour ce test :
- Cubase 6 ;
- Nuendo 5.5 ;
- Pro-tools 10 natif [2] ;
- Pyramix 7 ;
- Samplitude 11 ;
- Sequoia 11 ;
- Sonar X1 ;
- Reaper 4 ;
- Digital Performer 6.

Ces logiciels utilisent, *a priori*, des *moteurs* audio, c'est-à-dire des programmes dédiés à l'audio, qui réalisent l'opération de sommation avec des variables codées en virgule flottante, simple ou double précision. En effet, aujourd'hui les ordinateurs utilisent des processeurs qui calculent très facilement avec des variables en virgule flottante.

1. La loi de panoramique que nous utiliserons implique pourtant une atténuation de 3 dB d'un signal placé au centre car il est donc placé dans les deux canaux, gauche et droite à égale intensité (cf. Annexe C).

2. c'est-à-dire n'utilisant le processeur de l'unité centrale pour faire les calculs de traitement du signal et non un DSP, soit une puce dédiée

Pro-tools utilise depuis peu un moteur audio réalisant les traitements dont la sommation avec des variables codées en 32 bits flottants. Sonar utilise depuis longtemps la double précision de la représentation en virgule flottante. Reaper permet de réaliser les sommations dans les deux précisions.

**Logiciel référence :** D'après les éléments contenus dans les chapitres précédents, et compte tenu de la remarque précédente, nous avons besoin d'un programme dit *référence* qui puisse réaliser l'opération de sommation dans les deux précisions, simple et double, du codage en virgule flottante.

Nous avons donc choisi pour référence un programme de Gaël Martinet qui se rapproche beaucoup du programme de sommation expliqué à la sous-section 2.6.3. Le code, légèrement modifié pour traiter des fichiers audio, de cette référence est disponible sur le cd accompagnant le mémoire.

Nous disposons donc d'un logiciel référence pour notre test. Nous comparerons les autres logiciels à cette référence car nous savons exactement comment notre programme de sommation est codé, contrairement à tous les autres logiciels testés.

### 4.1.2 Protocole du test

#### Génération des fichiers de test

Nous allons réaliser ce test avec 128 fichiers audio de bruit rose échantillonnés à 48 kHz. Ces bruits sont tous décorrélés les uns des autres.

Un programme réalise *l'encapsulage* de 10 secondes de bruit rose, 128 fois de suite, dans des fichiers "wav", mono, codés en 32 bits virgule flottante[3]. Nous avons ensuite converti, en 24 bits fixes, ces fichiers afin de réaliser les tests avec les sources sonores codées dans ces deux formats.

Nous disposons donc de :
- 128 fichiers wav d'une durée de 10 secondes, échantillonnés en 48 kHz et codés en 32 bits flottant ;
- 128 fichiers wav d'une durée de 10 secondes, échantillonnés en 48 kHz et codés en 24 bits fixe.

3. Le code de cette fonction est donné sur le cd accompagnant le mémoire.

### Réalisation du test

**Logiciel référence** Notre logiciel référence Flux : : n'a pas d'interface graphique, on place donc les fichiers à sommer dans le dossier prévu par le programme puis on configure la précision avec laquelle l'opération de sommation sera réalisée, puis on compile le programme [4] et on le lance. On récupère notre mixage dans un fichier échantillonné à 48 kHz et codé dans la représentation donnée par le paramétrage du programme.

Le logiciel référence Flux : : peut donc réaliser les sommations de 128 pistes de bruits rose avec : un moteur de sommation en 32 bits et en 64 bits virgule flottante ; ainsi qu'avec un moteur de sommation optimisée, en 32 bits flottants qui trie les valeurs associées aux échantillons afin de minimiser l'erreur (cf. sous-section 2.6.3). Les résultats peuvent être enregistrés dans des fichiers wav codés en 32 ou en 64 bits virgule flottante.

**Logiciels à tester** Pour chaque logiciel, on doit en premier lieu créer un projet, ou une session, avec une fréquence d'échantillonnage de 48 kHz.

Puis, on doit s'assurer de la précision avec laquelle le moteur audio va réaliser ses opérations si plusieurs d'entre elles sont disponibles. Nous trouvons, souvent dans les options de la session, le paramètre qui permet de choisir entre 32 bits et 64 bits lorsqu'il existe.

Nous devons alors importer les 128 fichiers audio sur 128 pistes différentes du logiciel testé, en prenant garde à ce que les données ne soient pas converties [5]. Il faut aussi faire attention à ce que tous les fichiers audio soient bien alignés les uns par rapport aux autres, le moindre décalage faussant les résultats.

Puis, il nous faut vérifier que tous les *faders* de toutes les pistes de la console de mixage sont bien alignés à 0 dB et que les *panoramiques* sont bien tous au centre.

Il nous faut aussi vérifier que la loi de panoramique choisie est bien la loi *sinusoïdale -3 dB*. Car c'est la loi que nous avons utilisée lors de la programmation de notre logiciel référence.

Nous baisserons le master fader de 12 dB, afin que le résultat de la sommation ne dépasse pas le niveau 0 dB FS.

4. ici, opération qui consistera à créer un fichier exécutable
5. du moins de façon explicite

Enfin, il faut sélectionner la zone temporelle qui sera *bouncée* ou exportée, donc correspondant à la durée des fichiers audio de bruit rose, soit 10 secondes, et, *bouncer* ou exporter le mixage dans un fichier audio wav, échantillonné à 48 kHz, codé en 32, et en 64 bits flottants lorsque c'est proposé par le logiciel.

Si le logiciel testé propose de paramétrer la précision du moteur de sommation, il faut réaliser le test dans les deux précisions proposées, souvent 32 bits et 64 bits de la représentation en virgule flottante.

### 4.1.3 Analyses des résultats

Pour analyser les fichiers obtenus par le protocole décrit à la section précédente, nous avons utilisé la méthode par opposition de phase, afin d'étudier uniquement les différences entre les différents mixages. Nous avons utilisé un logiciel permettant de traiter des données audio codées en 32 et 64 bits flottants, et de les envoyer vers l'analyseur Flux : :, par l'intermédiaire du plug-in Flux : : nommé *sample grabber*[6]. Nous avons choisi Reaper car nous n'avions que des versions beta de Sonar[7], qui réagissaient de façon étrange avec les plug-in Flux : :.

Nous avons donc importé sur la *time-line* de Reaper, tous les fichiers wav de toutes les sommations réalisées, par tous les logiciels testés, et, nous avons analysé les différences entre toutes ces sommations. Pour cela, nous avons classé les analyses en 2 parties :

- analyse des différences entre les sommations réalisées en 32 bits flottants ;
- analyse des différences entre les sommations 32 bits et 64 bits,

en prenant pour référence notre sommateur programmé Flux : :.

#### Sommations en 32 bits flottants

Nous avons tout d'abord visualisé sur l'analyseur Flux : : les différences qui existaient entre notre logiciel référence Flux : :, paramétré pour réaliser une sommation avec des variables de calculs en 32 bits flottants, avec tous les logiciels traitants les données audio avec la même précision, en 32 bits flottants. Nous avons donc réalisé des captures d'écran pour illustrer les résultats.

D'après les résultats, nous pouvons regrouper les logiciels testés dans deux groupes. En effet, ces deux groupes de logiciels donnent des résultats légèrement différents.

6. qui lui aussi traite des données en 32 ou 64 bits virgule flottante.
7. l'autre logiciel à notre disposition permettant cela

Le premier groupe de logiciel est composé des logiciels suivants :
- Cubase 6 et Nuendo 5.5, regroupés sous le nom *Steinberg*[8] car ils donnent des résultats similaires ;
- Sonar X1, configuré pour réaliser une sommation sur 32 bits flottants ;
- Pro-tools 10 version native ;
- Pyramix 7.

La figure 4.1 représente l'analyse spectrale des différences existant entre le logiciel référence Flux : : et une première partie des logiciels testés.

Ainsi, les logiciels Cubase 6, Nuendo 5.5, Pro-tools 10 version native, Pyramix 7 ainsi que notre logiciel référence Flux : :, affichent des différences relativement faibles (autour de -140 dB) et relativement proches en terme spectral.

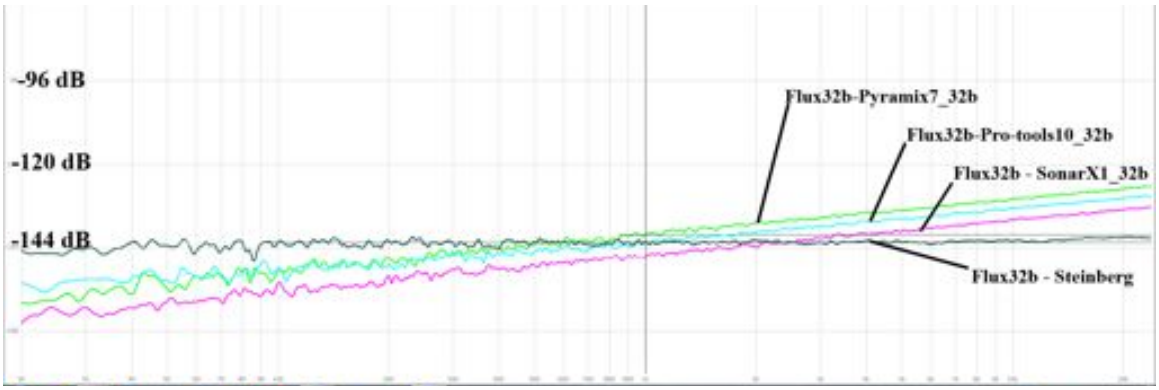


FIGURE 4.1 – Analyse spectrale des différences entre la sommation de 128 pistes de bruit rose réalisée par le logiciel référence Flux : :, configuré pour effectuer la sommation en 32 bits flottants, et un premier groupe de logiciels testés, présentant des résultats relativement similaires. Ce premier groupe de logiciels est composé : des logiciels Steinberg (Cubase 6 et Nuendo 5.5), de Sonar X1 configuré pour réaliser la sommation en 32 bits flottants, de Pro-tools 10 version native et de Pyramix 7.

Le deuxième groupe est composé des logiciels suivants :
- Samplitude 11 ;
- Reaper 4 configuré pour réaliser la sommation en 32 bits flottants ;
- Sequoia 11 ;
- Digital Performer 6 ;

8. du nom de l'entreprise qui développe ces logiciels

La figure 4.2 montre les analyses spectrales des différences existant entre notre logiciel référence Flux : :, et ce deuxième groupe de logiciels.

On peut constater que toutes les sommations réalisées par le second groupe de logiciels affichent des différences avec la référence Flux : :, d'un niveau bien plus important (de l'ordre de -70 dB), mais aussi que ces dernières sont très similaires les unes aux autres [9].

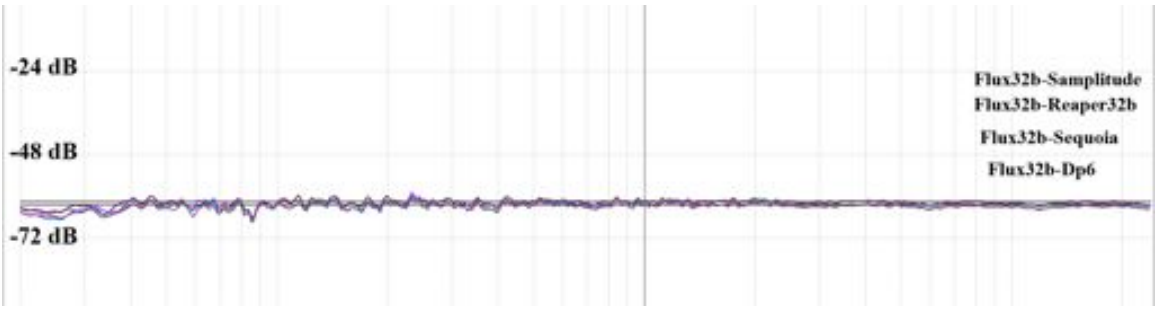


FIGURE 4.2 – Analyse spectrale des différences entre la sommation de 128 pistes de bruit rose réalisée par le logiciel référence Flux : :, configuré pour effectuer la sommation en 32 bits flottants, et le second groupe de logiciels testés, présentant des résultats assez différents de la référence. Ce second groupe de logiciels est composé : de Samplitude 11, de Reaper 4 configuré pour réaliser la sommation en 32 bits flottants, Sequoia 11 et de Digital Performer 6.

On peut supposer, vu leur profil spectral (exactement similaire à un bruit rose, soient les signaux sommés), qu'il s'agit de ce que l'on appellera un *offset de gain*. c'est-à-dire qu'il existe une différence entre deux signaux sonores mais que cette dernière n'est qu'une différence de niveau. Les signaux sonores des sommations réalisées par le second groupe de logiciels, seraient en fait relativement similaires à la sommation réalisée par la référence Flux : :, mis à part qu'ils diffèrent d'un niveau de $x$ dB.

Dans notre cas, les sommations réalisées par les logiciels du second groupe, sont moins fortes en niveau, d'environ $0,01$ dB, que celles réalisées par les logiciels du premier groupe. Laurent Millot a réalisé des analyses IDS des sommations réalisées par les logiciels testés, et les mesures indiquent que les niveaux moyens des sommations réalisées par les logiciels de ce que nous avons appelé le second groupe, sont inférieurs de 0,01 dB à celles réalisées par le premier groupe de logiciel.

9. spectralement parlant mais aussi en terme de niveau.

On peut interpréter ce constat en supposant que ces différences viennent du fait que certains logiciels utilisent une méthode légèrement différente pour coder la même loi de panoramique théorique. Comme on peut le voir à l'annexe C sur la loi de panoramique utilisée lors de la programmation de notre logiciel référence, cette dernière utilise les fonctions trigonométriques, sinus et cosinus. Ceci alors peut conduire à se poser plusieurs questions :

- Comment ces fonctions sont-elles programmées ?
- Utilisent-elles une table de valeurs ainsi qu'une interpolation pour donner leurs résultats ?

Néanmoins, mis à part ces *offset de gain*, on peut dire que les différences entre les sommations, en 32 bits flottants, de 128 pistes de bruit rose, par les logiciels testés sont relativement faibles.

### Différences entre les sommations en 32 et en 64 bits flottants

Deux des logiciels testés, ainsi que notre référence Flux : :, ont réalisé des sommations en 32 bits flottants et en 64 bits flottants. Nous avons donc décidé d'analyser la différence entre les deux versions des sommations d'un même logiciel. La figure 4.3 représente l'analyse spectrale des différences entre : les sommations réalisées en 64 bits et en 32 bits par le logiciel référence Flux : : ; les versions 64 et 32 bits du logiciel Sonar X1 et les versions 64 et 32 bits du logiciel Reaper 4.

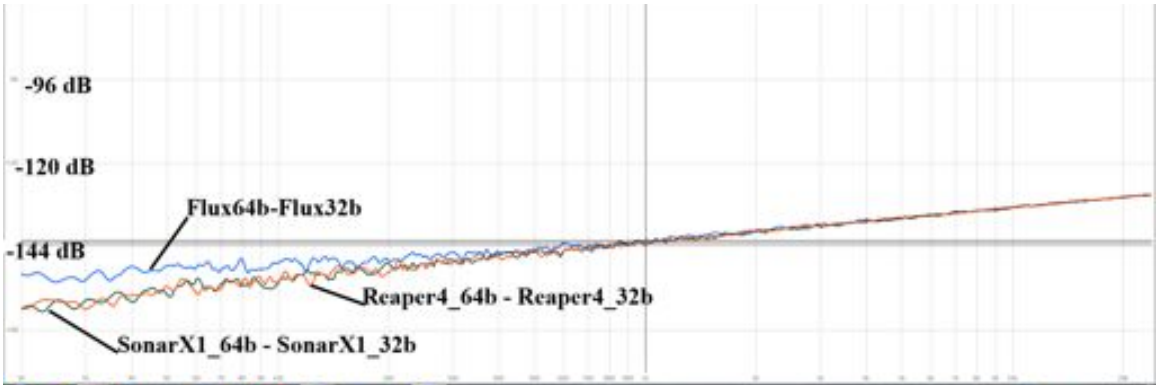


FIGURE 4.3 – Analyses spectrales des différences entre les deux versions, 64 bits et 32 bits, réalisées par Reaper 4, Sonar X1 et notre référence Flux : :, des sommations de 128 pistes de bruit rose.

Nous avons aussi analysé les différences qui existent entre la sommation en 64 bits, réalisée par notre logiciel référence Flux : :, et les sommations 32 bits réalisées par les logiciels testés.

La figure 4.4 montre le spectre fréquentiel de ces différences. Nous ne montrerons sur ce graphique, que les analyses des logiciels du premier groupe, défini à la sous-section 4.1.3.

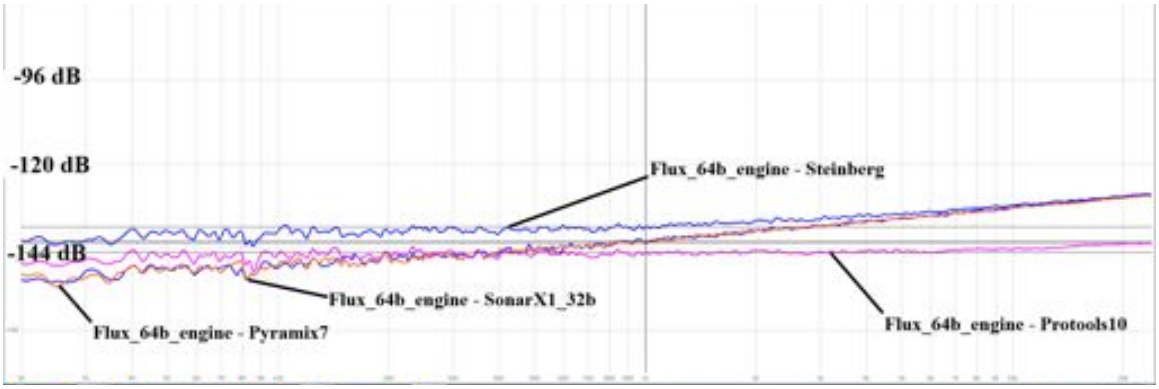


FIGURE 4.4 – Analyse spectrale des différences entre la sommation de 128 pistes de bruit rose réalisée par le logiciel référence Flux : :, configuré pour effectuer la sommation en 64 bits flottants, et le premier groupe de logiciels testés. Ce premier groupe de logiciel est composé : des logiciels Steinberg (Cubase 6 et Nuendo 5.5), de SonarX1 configuré pour réaliser la sommation en 32 bits flottants, de Pro-tools 10 version native et de Pyramix 7.

Ces valeurs sont toutes d'un niveau moyen faible. Les analyses IDS de Laurent Millot vont aussi dans ce sens en ne mesurant pas de différences de niveaux dans les sous-bandes fréquentielles de son analyseur [10].

10. en donnant les niveaux en dB relatifs des poids relatifs des sous-bandes avec une précision de deux décimales.

### 4.1.4 Conclusion du test

Compte tenu des mesures réalisées lors de ce test, nous pouvons dire que l'opération de sommation, lorsqu'elle est réalisée en 32 bits virgule flottante, génère des erreurs d'arrondi d'un niveau moyen faible. [11]

De plus, on a pu remarquer lors de ces analyses, la relative similitude des profils spectraux de chacune des sommations réalisée.

Il est aussi nécessaire de rappeler que certains logiciels donnent des résultats légèrement différents mais qu'il s'agit sûrement d'un offset de gain dû au codage de la loi de panoramique.

On peut donc conclure en disant que les opérations de sommations ne sont pas fondamentalement différentes d'un logiciel à un autre, mais qu'il existe néanmoins des différences. On peut alors se poser la question : "Peut-on entendre ces différences ?". Et même si ce n'est pas l'objet de ce mémoire, on peut douter de la possibilité de percevoir des différences d'un tel niveau.

Il pourrait être intéressant de réaliser un test avec le même protocole, mais en utilisant cette fois des fichiers de test ayant un contenu sonore *musical*[12], afin d'étudier la corrélation entre les erreurs d'arrondis et ces signaux sonores codés dans les fichiers de test.

Le sujet mériterait d'être continué, afin de déterminer l'impact des *sous-groupes* ou *stèmes* des erreurs d'arrondis. En effet, certains mixages peuvent comporter [13] des *bus* servant à mélanger certains signaux avant la sommation *master*. Certains logiciels utilisent sûrement des *bus* dans lesquels les données sont codées en 32 bits flottants, générant ainsi des erreurs d'arrondis pour chaque *sous-groupe* ainsi que lors de la sommation finale. Le niveau d'erreur au final est susceptible d'être plus important que lorsqu'on ne réalise qu'un seul mélange, la sommation *Master*.

Par ailleurs, nous avons réalisé, *presque*, tous nos tests en *bounce offline*[14], c'est-à-dire que le calcul de sommation n'est pas réalisé en *temps réel*. Pourtant lorsqu'on

11. On peut dire néanmoins, avec certitude, d'après les éléments de la sous-section 2.6.2, que les erreurs d'arrondi sont d'un niveau moyen bien plus faible lorsque l'opération de sommation est réalisée en 64 bits virgule flottante.
12. même si le bruit rose peut sûrement être considéré par certains comme musical
13. c'est souvent le cas.
14. sauf Protools qui ne fait que du *online*

mixe un morceau ou un film, on écoute le *playback* de la sommation réalisée *online*, c'est-à-dire en temps réel. Cela conduit, notamment à se poser des questions. Qu'en est-il de cette version *playback* de la sommation ? Comment cette opération est-elle réalisée et le résultat est-il différent du résultat *offline* ?

## 4.2 Comparaison de traitements numériques

Cette section ne devrait pas, à proprement parler, figurer dans un mémoire sur la sommation numérique car nous allons y comparer, non pas strictement l'opération de sommation, mais les opérations de traitement de signal généralement utilisées lors d'un mixage.

Mais, à la lumière des résultats donnés par les programmes illustrants les erreurs d'arrondi en virgule flottante de la section 2.6.3, et par le plug-in *Quantum*, dans son mode *filtrage*, il apparaît que les calculs effectués par un effet audio numérique codé en virgule flottante simple précision, génèrent des erreurs d'arrondis d'un niveau parfois bien plus élevé que lors de l'opération de sommation testée au chapitre précédent.

Ces résultats nous ont intrigué et nous avons donc voulu savoir quel était *l'impact* des erreurs commises lors d'un mixage, comportant des plug-in qui réalisent leurs calculs en virgule flottante simple précision, par rapport à une même session de mixage mais comportant cette fois des plug-in réalisant leurs calculs en double précision.

Nous expliquons d'abord le protocole utilisé lors de ce test, puis nous donnons ensuite les résultats des analyses et des écoutes que nous avons faites. Enfin, nous concluons par les perspectives ouvertes par ce test.

### 4.2.1 Protocole de test

Nous voulons analyser les différences qui existent entre un mixage réalisé avec des traitements effectuant leurs calculs en 32 bits flottants et le même mixage réalisé avec des traitements effectuant leurs calculs en 64 bits flottants.

Nous avons décidé de réaliser ce test en utilisant un morceau mixé par Gaël Martinet. La session de mixage comporte 8 pistes mono et 5 pistes stéréo. L'enregistrement a été réalisé à la fréquence d'échantillonnage de 88,2 kHz.

Tous les plug-in utilisés sont des plug-in Flux : :. Ainsi, nous pouvons utiliser exactement les mêmes plug-in mais effectuant leurs calculs dans deux précisions différentes.

Nous avons cependant remplacé les filtres de type *state-space*[15] du plug-in *Epure II*, qui est un *equaliseur* comportant 5 bandes de filtres, par des filtres de type *biquad* qui sont utilisés dans le plug-in *Quantum* décrit au chapitre 3.

Les autres plug-in utilisés sont l'*Alchemist*, *Solera II*, *Syrah*, la série des *dynamics Pure*, les réverbérations *Ircam Tools* ainsi que le limiter *Elixir*.

Nous avons donc préparé une session dans un logiciel, qui nous permettait d'insérer des plug-in et d'exporter des fichiers en 64 bits virgule flottante. En l'occurrence, nous avons choisi Reaper 4 car nous n'avions qu'une version *beta* de Sonar X1 qui semblait avoir quelques problèmes avec certains plug-in Flux : :. Le problème de Reaper 4 réside dans le fait qu'il implique un *offset de gain* comme expliqué à la sous-section 4.1.3. Gaël Martinet a résolu ce problème en programmant un plug-in de volume et de panoramique, remplaçant les *faders* et les panoramiques de Reaper. Le code de ce plug-in est donné à l'annexe C.1.

Nous avons exporté une première fois le morceau mixé, avec les plug-in compilés dans une version effectuant les calculs en simple précision.

Puis, nous avons refait un export du morceau mixé avec les plug-in compilés dans leur version double précision[16].

Ensuite, nous avons effectué une conversion de fréquence d'échantillonnage[17] ainsi qu'une troncature sur 16 bits, codés en représentation virgule fixe, avec *dithering*, des deux versions du mixage. Nous avons fait cela afin de savoir s'il restait des différences entre les deux versions testées lorsqu'on réduit la précision comme on le fait pour s'adapter au format CD.

15. type de filtre ayant un rapport signal/bruit constant sur tout le spectre.

16. Les plug-in de traitements Flux : : vendus dans le commerce effectuent tous leurs calculs en double précision.

17. nous avons essayé de choisir un convertisseur le plus *transparent possible, nous avons utilisé le convertisseur Izotope.*

### 4.2.2 Analyse et écoutes des résultats du test

Nous disposons donc de deux fichiers correspondant au morceau mixé avec les plug-in effectuant leurs calculs sur 32 bits et mixé avec les plug-in effectuant leurs calculs sur 64 bits.

Nous avons donc importé les deux fichiers dans une nouvelle session Reaper 4, et nous les avons mis en opposition de phase afin de n'écouter, et d'analyser, que les différences entre ces deux derniers.

On trouve ainsi des différences d'un niveau moyen d'autour de -60 dB. Ce qui reste relativement faible.

Mais en écoutant uniquement la différence entre les deux fichiers on se rend compte que ces signaux sont corrélés avec la musique. Ceci implique le fait qu'on peut donc appeler ce bruit *distorsion*.

Lorsqu'on écoute uniquement la différence [18] entre les deux versions du mixage, on peut entendre tous les instruments du mixage. Les cymbales de la batterie sont distordues ; l'accordéon, la guitare et la caisse claire sont *plongés* dans la réverbération ; la basse apparaît à certains moments, disparait à d'autres.

De plus, de légers et très courts bruits qui ressemblent à des légers craquements apparaissent à certains moments. Cela pourrait être dû aux plug-in contrôlant la dynamique [19], qui calculent moins précisément leurs variables de calculs en simple précision qu'en double, les différences se traduisant par de légers changements de niveau durant les *transitoires*.

L'écoute de cette différence fait apparaître que le traitement de la réverbération, en simple ou double précision, *joue* dans les différences entre les deux versions du mixage. En effet, comme expliqué à la fin de la section 3.4 et à la sous-section 2.6.3, les erreurs commises lors d'un traitement comportant une boucle de *rétro-action*, c'est-à-dire utilisant les valeurs de sortie correspondant aux échantillons précédents pour calculer les valeurs de sortie associées aux échantillons actuels, sont très susceptibles de s'amplifier, de ne pas rester bornées par $\epsilon_{machine}$ (cf. Paragraphe 2.6.2), et donc, de se traduire par un bruit d'un niveau possiblement perceptible par l'oreille humaine.

18. la somme avec une opposition de phase.
19. compresseurs, limiteurs, etc...

En effet, en écoutant la version du mixage réalisée avec les plug-in calculant en simple précision, nous avons la sensation d'un espace légèrement réduit, par rapport à la version double précision. Mais, nous ne sommes pas vraiment objectifs pour parler d'écoute et donc d'une notion subjective, il faudrait donc procéder à des tests subjectifs sérieux afin de pouvoir déterminer si on peut réellement entendre des différences. Mais, on peut pour autant dire que, ces dernières étant d'un niveau assez faible, seul des auditeurs *experts* pourront, peut-être, entendre une différence entre les deux versions.

Néanmoins, les signaux traités par les effets de réverbération étant d'un niveau relativement faible, et la sensation d'espace se faisant avec des informations d'un niveau sonore faible, il est possible que les erreurs décrites au paragraphe précédent changent légèrement la perception de l'espace sonore dans le mixage.

Nous avons enfin analysé et écouté les deux versions du mixage mais dont les fréquences d'échantillonnage ont été converties à 44,1 kHz, et les valeurs associées aux échantillons converties en nombres entiers dans la représentation binaire virgule fixe sur 16 bits [20]. Ces caractéristiques sont nécessaires afin que les données audio puissent être gravées sur un CD audio. Il apparaît que les différences, entre les versions simple et double précision du mixage, sont toujours présentes.

### 4.2.3 Conclusion du test

Ce test a montré qu'il existait des différences entre des traitements audio-numériques calculés en simple précision et ces mêmes traitements calculés en double précision de la représentation virgule flottante.

Ces différences sont relativement faibles mais corrélées au signal musical. On peut cependant rappeler que ces différences ne seront probablement pas perçues par la majorité des auditeurs, mais juste peut-être par certains auditeurs *experts*.

Néanmoins, le nombre de pistes ainsi que le nombre de traitements, utilisés pour ce mixage, étant relativement peu élevé, on peut imaginer que lors d'un mixage comportant un grand nombre de pistes, ainsi qu'un grand nombre de traitements, le niveau des erreurs d'arrondi est plus important. Mais, pour pouvoir l'affirmer, il faudrait refaire ce test avec une session de mixage comportant plus de pistes et plus de traitements.

20. avec dithering

Ce test nous a néanmoins permis de confirmer les résultats des chapitres précédents à propos des traitements dont les erreurs relatives sont susceptibles de dépasser significativement $\epsilon_{machine}$ [21].

## 4.3 Conclusion du chapitre

Dans ce chapitre nous avons, dans un premier temps, partiellement répondu à la question initiale du mémoire à propos de l'existence de différences objectives entre les réalisations par différents logiciels d'un même mixage. La réponse est donc : "Oui, il y a des différences, mais elles sont d'un niveau moyen faible". Les logiciels testés effectuent donc l'opération de sommation quasiment de la même manière.

Dans la deuxième partie de chapitre, nous avons pu comparer l'effet des calculs sur des traitements de signaux audio-numériques d'un mixage, effectués en simple précision par rapport aux mêmes calculs réalisés en double précision (représentation binaire en virgule flottante dans les deux cas). Les précédents chapitres avaient mis en lumière le fait que certains traitements - filtres, réverbérations notamment - étaient assez sensibles à la précision utilisée pour effectuer leurs calculs. Le test, décrit dans la deuxième partie du chapitre, a permis d'illustrer ce phénomène à l'échelle d'un mixage composé d'une petite quinzaine de pistes.

21. cf. Sous-section 2.6.2

# Chapitre 5

# Conclusion générale

Au chapitre 2, nous avons étudié les représentations binaires des nombres, et particulièrement la représentation en virgule flottante puisque c'est celle qui est le plus souvent utilisée dans les logiciels audio-numériques. Nous avons soulevé le problème des erreurs d'arrondi et nous avons compris que les erreurs d'arrondi commises lors du codage de valeurs dans la représentation en virgule flottante étaient plus importantes en simple précision qu'en double précision.

Au chapitre 3, nous avons pu, grâce au plug-in *Quantum*, écouter et analyser les erreurs commises lors de traitements simples : troncature en entier, gain suivi d'une atténuation ; filtrage à l'aide d'un filtre bi-quadratique. Nous en avons conclu que lors des traitements audio-numériques utilisant une *boucle de rétroaction*, les erreurs d'arrondi commises sont susceptibles de s'amplifier et d'atteindre des niveaux relativement importants par rapport au niveau du signal traité. L'utilisation du format double précision de la représentation binaire en virgule flottante permet de diminuer considérablement ces erreurs.

Le chapitre 4 nous a permis, à travers un premier test, de répondre partiellement à la question posée par ce mémoire : "Existe-t-il des différences entre les réalisations d'une même sommation par différents logiciels de mixage audio-numérique ?". La réponse est : "Oui, mais elles sont d'un niveau très faible". L'opération de sommation numérique n'est donc pas fondamentalement différente d'un logiciel à un autre. Un logiciel réalisant l'opération de sommation en double précision de la représentation en virgule flottante, donc utilisant des mots binaires composés de 64 bits, commettra des erreurs d'arrondi d'un niveau plus faible encore.

Par ailleurs, le second test du chapitre 4 a montré qu'il pouvait y avoir des différences entre deux versions d'un même mixage, lorsque les traitements du signal

numérique de l'une des versions sont réalisés en simple précision et que les traitements de l'autre version sont en double précision. En effet, le chapitre sur le plug-in *Quantum* a montré des niveaux d'erreur en simple précision plus importants que les niveaux d'erreur commis lors de l'opération de sommation numérique. Ce sujet sur la sommation numérique nous aura donc orientés vers un autre problème : des erreurs relativement élevées peuvent être commises lors de certains traitements en simple précision de la représentation en virgule flottante.

Mais, la question de l'utilisation de la double précision pour les calculs du traitement du signal audio-numérique prend vraiment sens du point de vue d'un développeur de logiciels. En effet, dans la perspective d'un traitement du signal se voulant respectueux au maximum de la *matière sonore* traitée, il est important de prendre en compte les erreurs potentielles et de réaliser les calculs en double précision.

Enfin, pour résumer, nous avons surtout compris, lors de ce travail que le traitement audio-numérique n'est composé, en pratique, que de questions relevant des mathématiques. Mais, comme dans tout calcul scientifique, il faut prêter attention aux erreurs commises pour être au plus proche de la valeur exacte (théorique) du résultat. Et, qu'à ce titre, l'audio-numérique n'est pas exempt d'erreurs introduites par le recours à une précision, effective, limitée des traitements du signal, et que ces dernières se traduisent par l'introduction de bruits ou de distorsions. Ces distorsions sont, au départ, de faible niveau mais, comme elles peuvent s'amplifier tout au long des traitements réalisés lors d'un mixage, la question de la représentation binaire utilisée doit être toujours prise en compte.

La question de la perception de ces distorsions par l'oreille humaine reste néanmoins à étudier. Toutefois, on peut rappeler que toutes ces considérations ne prennent sens que pour des auditeurs *experts* et que la majorité des auditeurs n'est pas concernée par ces questions. Pour autant, il est, à mon sens, important si ce n'est indispensable pour l'ingénieur du son de connaître les outils qu'il utilise lors de son travail. L'ingénieur du son devrait donc comprendre le fonctionnement général et les limites des logiciels qui traitent les signaux audio-numériques afin de :

- choisir ses outils sur des critères objectifs liés au résultat perceptif sonore que l'on peut attendre des outils choisis ;
- prévoir et choisir, en connaissance de cause, le taux et la nature des distorsions susceptibles d'apparaître au fur et à mesure de son travail ;
- se trouver en mesure de réaliser des choix esthétiques mais aussi de décider de la pertinence (d'un point de vue perceptif notamment) de l'ajout de nouveaux

traitements à toute ou partie de la matière sonore sur laquelle il travaille.

# Annexe A

# Erreur d'arrondi lors d'un calcul itératif - Code du programme

```
// Definitions 32 bits
float x_original = 0.993;
float y = 0.827;
float fGain = 1.9952623; // +6 dB
float fGainInv = 1.f / fGain; // -6 dB
float x_f = x_original;
double erreur_float = (double)x_f - (double)0.993;
// Definitions 64 bits
double x_d_original = 0.9930000;
double y_d = 0.8270000;
double dGain = 1.9952623; // +6 dB
double dGainInv = 1.f / dGain; // -6 dB
double x_d = x_d_original;
double erreur_double = x_d - (double)0.993;
const int nNumberOfIteration = 20;
for (int j=0; j<nNumberOfIteration ; j++)
{
   x_f += y;
   x_f *= fGain;
   x_f *= fGainInv;
   x_f -= y;
   x_d += y_d;
   x_d *= dGain;
   x_d *= dGainInv;
   x_d -= y_d;
```

```
    erreur_float = (double)x_f - (double)x_original;
    erreur_double = x_d - x_d_original;
    fichier << "Iteration " << j+1 << endl;
    fichier << "x_f = " << x_f << endl;
    fichier << "Erreur_absolue_x_float = " << erreur_float
       << " soit " << 20.*log10(fabs(erreur_float)) << " dB
       " << endl;
    fichier << "x_d = " << x_d << endl;
    fichier << "Erreur_absolue_x_double = " << erreur_double
        << " soit " << 20.*log10(fabs(erreur_double)) << "
       dB " << endl;
    fichier << " " << endl;
}
```

# Annexe B

# Sommation en simple, double précision et sommation optimisée

## B.1 Code du programme

```
#include <iostream>
#include <math.h>
#include <fstream>
#include <string>
#include <stdlib.h>
#include <time.h>

using namespace std;

double MAX_f32;
double MIN_f32;

const long nNumberOfIteration = 48000;
const long nNumberOfValues = 256;

double FloatDiffMin = MAX_f32;
double FloatDiffMax = MIN_f32;
double FloatDiffMean = 0.f;

double DoubleDiffMin = MAX_f32;
double DoubleDiffMax = MIN_f32;
double DoubleDiffMean = 0.0;
```

```
double Diff3264Min_A = MAX_f32;
double Diff3264Max_A = MIN_f32;
double Diff3264Mean_A = 0.0;

double Diff3264Min_B = MAX_f32;
double Diff3264Max_B = MIN_f32;
double Diff3264Mean_B = 0.0;

class Abs_less_than_functor
{
   public:
   bool operator()(double i, double j) const
   {
   return (fabs(i) < fabs(j));
   }
};

double Linear2dB(double x)
   {
      return 20.0*log10(x);
   }

int main()
{
   // Ouverture fichier et ecriture entete //
   string const nomfichier("resultats_float_double_256_gcc.
      txt");
   ofstream fichier(nomfichier.c_str(), ios::trunc);
   if(fichier)
   {
      fichier << " float vs double sum " << endl;
      fichier << " compile avec gcc " << endl;
      //fichier << " avec Relax IEEE compliant " << endl;
      fichier << " sans Relax IEEE compliant " << endl;
      fichier << " " << endl;
      fichier << "Nombre de pistes = " << nNumberOfValues
        << endl;
      fichier << "Nombre de samples par piste " <<
         nNumberOfIteration << endl;
      fichier << " " << endl;
```

```
    }
    else
    {
        cout << "Erreur" << endl;//return 1;
    }
    fichier.close();

    srand(time(NULL)); // Changement des seeds pour la
        fontion random

    for (long j = 0; j < nNumberOfIteration; j++)
    {
        float fData[nNumberOfValues];
        for (long i = 0; i < nNumberOfValues; i++)
        {
            // Generation des valeurs aleatoires entre -1 et 1
                //
            fData[i] = ( ( (float)rand() / (float)RAND_MAX) *
                2.f ) - 1.f;
        }

        //Sommation des valeurs non-triees //
        float FloatRes1 = 0.f;
        double DoubleRes1 = 0.0;
        for (long i = 0; i < nNumberOfValues; i++)
        {
            FloatRes1 += fData[i];
            DoubleRes1 += (double)fData[i];
        }

        // Tri des valeurs, "Optimisation" //
        std::sort(fData, fData + nNumberOfValues,
            Abs_less_than_functor());

        // Sommation des valeurs triees //
        float FloatRes2 = 0.f;
        double DoubleRes2 = 0.0;
        for (long i = 0; i < nNumberOfValues; i++)
        {
            FloatRes2 += fData[i];
```

```
            DoubleRes2 += (double)fData[i];
        }

        // Calcul des differences entres les differentes
            versions //
        double FloatDiff = fabs((double)FloatRes1 - (double)
            FloatRes2);
        double DoubleDiff = fabs(DoubleRes1 - DoubleRes2);
        double Diff3264_A = fabs(DoubleRes1 - (double)
            FloatRes1);
        double Diff3264_B = fabs(DoubleRes1 - (double)
            FloatRes2);

        // Calcul des max, min et moyennes des difference //
        FloatDiffMin = min(FloatDiffMin, FloatDiff);
        FloatDiffMax = max(FloatDiffMax, FloatDiff);
        FloatDiffMean += FloatDiff;

        DoubleDiffMin = min(DoubleDiffMin, DoubleDiff);
        DoubleDiffMax = max(DoubleDiffMax, DoubleDiff);
        DoubleDiffMean += DoubleDiff;

        Diff3264Min_A = min(Diff3264Min_A, Diff3264_A);
        Diff3264Max_A = max(Diff3264Max_A, Diff3264_A);
        Diff3264Mean_A += Diff3264_A;

        Diff3264Min_B = min(Diff3264Min_B, Diff3264_B);
        Diff3264Max_B = max(Diff3264Max_B, Diff3264_B);
        Diff3264Mean_B += Diff3264_B;
    }

// Fin du calcul des moyennes //
FloatDiffMean /= (double)nNumberOfIteration;
DoubleDiffMean /= (double)nNumberOfIteration;
Diff3264Mean_A /= (double)nNumberOfIteration;
Diff3264Mean_B /= (double)nNumberOfIteration;

// Ouverture fichier pour eriture des resultats //
fichier.open(nomfichier.c_str(), ios::app);
fichier << " " << endl;
```

```
cout.unsetf(ios::floatfield);
fichier.precision(40);
fichier << fixed << endl;
fichier << "//=========================================" <<
   endl;
fichier << "Difference between sorted and non-sorted sum in
    32 bit float" << endl;
fichier << "//=========================================" <<
   endl;
fichier << "Min = " << FloatDiffMin << " = " << Linear2dB(
   FloatDiffMin) << " dB" << endl;
fichier << "Max = " << FloatDiffMax << " = " << Linear2dB(
   FloatDiffMax) << " dB" << endl;
fichier << "Moyenne = " << FloatDiffMean << " = " <<
   Linear2dB(FloatDiffMean) << " dB" << endl;
fichier << " " << endl;
fichier << "//=========================================" <<
   endl;
fichier << "Difference between sorted and non-sorted sum in
    64 bit float" << endl;
fichier << "//=========================================" <<
   endl;
fichier << "Min = " << DoubleDiffMin << " = " << Linear2dB(
   DoubleDiffMin) << " dB" << endl;
fichier << "Max = " << DoubleDiffMax << " = " << Linear2dB(
   DoubleDiffMax) << " dB" << endl;
fichier << "Moyenne = " << DoubleDiffMean << " = " <<
   Linear2dB(DoubleDiffMean) << " dB" << endl;
fichier << " " <<endl;
fichier << "//=========================================" <<
   endl;
fichier << "Difference between non-sorted sum in 32 bit
   float and non-sorted sum in 64 bit float" << endl;
fichier << "//=========================================" <<
   endl;
fichier << "Min = " << Diff3264Min_A << " = " << Linear2dB(
   Diff3264Min_A) << " dB" << endl;
fichier << "Max = " << Diff3264Max_A << " = " << Linear2dB(
   Diff3264Max_A) << " dB" << endl;
```

```
fichier << "Moyenne = " << Diff3264Mean_A << " = " <<
   Linear2dB(Diff3264Mean_A) << " dB" << endl;
fichier << " " <<endl;
fichier << "//=======================================" <<
   endl;
fichier << "Difference between sorted sum in 32 bit float
   and non-sorted sum in 64 bit float" << endl;
fichier << "//=======================================" <<
   endl;
fichier << "Min = " << Diff3264Min_B << " = " << Linear2dB(
   Diff3264Min_B) << " dB" << endl;
fichier << "Max = " << Diff3264Max_B << " = " << Linear2dB(
   Diff3264Max_B) << " dB" << endl;
fichier << "Moyenne = " << Diff3264Mean_B << " = " <<
   Linear2dB(Diff3264Mean_B) << " dB" << endl;
fichier << " " <<endl;
fichier.close();
}
```

## B.2 Résultats des différentes sommations

```
 float vs double sum
 complile avec compiler intel
 avec floating point model = Precise (/fp:precise)

Nombre de pistes = 128
Nombre de samples par piste 48000

//=================================================================
Difference between sorted and non-sorted sum in 32 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Max = 0.00001335144042968750000000000000000000000 = -97.5 dB
Moyenne = 0.00000115267992562924820000000000000000000 = -118.8 dB

//=================================================================
Difference between sorted and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Max = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Moyenne = 0.00000000000000000000000000000000000000000 = -1.#INF dB

//=================================================================
Difference between non-sorted sum in 32 bit float
and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Max = 0.00001398194581270217900000000000000000000 = -97.1 dB
Moyenne = 0.00000094956836740796760000000000000000000 = -120.4 dB

//=================================================================
Difference between sorted sum in 32 bit float
and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Max = 0.00001003500074148178100000000000000000000 = -99.97 dB
Moyenne = 0.00000069025372310231135000000000000000000 = -123.2 dB
```

```
  float vs double sum
 complile avec compiler intel
 avec floating point model = Fast (/fp:fast)

Nombre de pistes = 128
Nombre de samples par piste 48000

//=================================================================
Difference between sorted and non-sorted sum in 32 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Max = 0.00001144409179687500000000000000000000000 = -98.8 dB
Moyenne = 0.00000115298452631880840000000000000000000 = -118.7 dB

//=================================================================
Difference between sorted and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Max = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Moyenne = 0.00000000000000000000000000000000000000000 = -1.#INF dB

//=================================================================
Difference between non-sorted sum in 32 bit float
and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Max = 0.00001233257353305816700000000000000000000 = -98.2 dB
Moyenne = 0.00000094625307247042661000000000000000000 = -120.5 dB

//=================================================================
Difference between sorted sum in 32 bit float and
non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF0 dB
Max = 0.00000885967165231704710000000000000000000 = -101.1 dB
Moyenne = 0.00000069314998108893636000000000000000000 = -123.2 dB
```

```
 float vs double sum
 compile avec gcc
 avec Relax IEEE compliant

Nombre de pistes = 128
Nombre de samples par piste 48000

//=================================================================
Difference between sorted and non-sorted sum in 32 bit float
//=================================================================
Min = 0.0000000000000000000000000000000000000000 = -inf dB
Max = 0.0000133514404296875000000000000000000000 = -97.5 dB
Moyenne = 0.00000118288149436314899925581749229319 68 = -118.5 dB

//=================================================================
Difference between sorted and non-sorted sum in 64 bit float
//=================================================================
Min = 0.0000000000000000000000000000000000000000 = -inf dB
Max = 0.0000000000000000000000000000000000000000 = -inf dB
Moyenne = 0.0000000000000000000000000000000000000000 = -inf dB

//=================================================================
Difference between non-sorted sum in 32 bit float
and non-sorted sum in 64 bit float
//=================================================================
Min = 0.0000000000000000000000000000000000000000 = -inf dB
Max = 0.0000109076499938964843750000000000000000 = -99.2 dB
Moyenne = 0.00000097133219242095950653756789017201 36 = -120.2 dB

//=================================================================
Difference between sorted sum in 32 bit float
and non-sorted sum in 64 bit float
//=================================================================
Min = 0.0000000000000000000000000000000000000000 = -inf dB
Max = 0.0000077486038208007812500000000000000000 = -102.2 dB
Moyenne = 0.00000071740150451660153793604452962529 02 = -122.9 dB
```

```
 float vs double sum
 compile avec gcc
 sans Relax IEEE compliant

Nombre de pistes = 128
Nombre de samples par piste 48000

//=================================================================
Difference between sorted and non-sorted sum in 32 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -inf dB
Max = 0.00001335144042968750000000000000000000000 = -97.5 dB
Moyenne = 0.00000117554267247517913866697563535224 09 = -118.6 dB

//=================================================================
Difference between sorted and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -inf dB
Max = 0.00000000000000000000000000000000000000000 = -inf dB
Moyenne = 0.00000000000000000000000000000000000000000 = -inf dB

//=================================================================
Difference between non-sorted sum in 32 bit float
and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -inf dB
Max = 0.00001192092895507812500000000000000000000 = -98.5 dB
Moyenne = 0.00000097328921159108478857737659778459 62 = -120.2 dB

//=================================================================
Difference between sorted sum in 32 bit float
and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -inf dB
Max = 0.00000911951065063476562500000000000000000 = -100.8 dB
Moyenne = 0.00000070752700169881182665313238897009 11 = -123.0 dB
```

```
 float vs double sum
 complile avec compiler intel
 avec floating point model = Fast (/fp:fast)

Nombre de pistes = 256
Nombre de samples par piste 48000

//=================================================================
Difference between sorted and non-sorted sum in 32 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Max = 0.00002479553222656250000000000000000000000 = -92.1 dB
Moyenne = 0.00000226700523247321440000000000000000000 = -112.9 dB

//=================================================================
Difference between sorted and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Max = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Moyenne = 0.00000000000000000000000000000000000000000 = -1.#INF dB

//=================================================================
Difference between non-sorted sum in 32 bit float
and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Max = 0.00002275966107845306400000000000000000000 = -92.9 dB
Moyenne = 0.00000186927703907713300000000000000000000 = -114.6 dB

//=================================================================
Difference between sorted sum in 32 bit float
and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Max = 0.00001575145870447158800000000000000000000 = -96.1 dB
Moyenne = 0.00000134042532105619700000000000000000000 = -117.5 dB
```

```
 float vs double sum
 complilÈ avec compiler intel
 avec floating point model = Precise (/fp:precise)

Nombre de pistes = 256
Nombre de samples par piste 48000

//=================================================================
Difference between sorted and non-sorted sum in 32 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Max = 0.00002670288085937500000000000000000000000 = -91.5 dB
Moyenne = 0.00000227549734214941660000000000000000000 = -112.8 dB

//=================================================================
Difference between sorted and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Max = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Moyenne = 0.00000000000000000000000000000000000000000 = -1.#INF dB

//=================================================================
Difference between non-sorted sum in 32 bit float
and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Max = 0.00002238154411315918000000000000000000000 = -93.0 dB
Moyenne = 0.00000190174562158063060000000000000000000 = -114.4dB

//=================================================================
Difference between sorted sum in 32 bit float
and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -1.#INF dB
Max = 0.00001486204564571380600000000000000000000 = -96.5 dB
Moyenne = 0.00000134363884959990780000000000000000000 = -117.4 dB
```

```
 float vs double sum
 compile avec gcc
 avec Relax IEEE compliant

Nombre de pistes = 256
Nombre de samples par piste 48000

//=================================================================
Difference between sorted and non-sorted sum in 32 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -inf dB
Max = 0.00002861022949218750000000000000000000000 = -90.9 dB
Moyenne = 0.00000232492387294769281349550958670757769 = -112.7 dB

//=================================================================
Difference between sorted and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -inf dB
Max = 0.00000000000000000000000000000000000000000 = -inf dB
Moyenne = 0.00000000000000000000000000000000000000000 = -inf dB

//=================================================================
Difference between non-sorted sum in 32 bit float
and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -inf dB
Max = 0.00002527236938476562500000000000000000000 = -91.9 dB
Moyenne = 0.00000192360952496528628876413039017201360 = -114.3 dB

//=================================================================
Difference between sorted sum in 32 bit float
and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -inf dB
Max = 0.00001680850982666015625000000000000000000 = -95.5 dB
Moyenne = 0.00000138680016001065562333751258333736090 = -117.2 dB
```

```
 float vs double sum
 compile avec gcc
 sans Relax IEEE compliant

Nombre de pistes = 256
Nombre de samples par piste 48000

//=================================================================
Difference between sorted and non-sorted sum in 32 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -inf dB
Max = 0.00002479553222656250000000000000000000000 = -92.2 dB
Moyenne = 0.00000231460357705752072920296624758407 23 = -112.7 dB

//=================================================================
Difference between sorted and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -inf dB
Max = 0.00000000000000000000000000000000000000000 = -inf dB
Moyenne = 0.00000000000000000000000000000000000000000 = -inf dB

//=================================================================
Difference between non-sorted sum in 32 bit float
and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -inf dB
Max = 0.00002461671829223632812500000000000000000 = -92.2 dB
Moyenne = 0.00000190397972861925758137973622302618 89 = -114.3 dB

//=================================================================
Difference between sorted sum in 32 bit float
and non-sorted sum in 64 bit float
//=================================================================
Min = 0.00000000000000000000000000000000000000000 = -inf dB
Max = 0.00001555681228637695312500000000000000000 = -96.2 dB
Moyenne = 0.00000139860063791275034070238098699023 87 = -117.1 dB
```

# Annexe C

# Les lois de panoramique

Il existe différentes façons de répartir un signal dans deux bus L et R. Nous avons choisi d'utiliser la loi de panoramique trigonométrique -3 dB.

On utilise les fonctions sinus et cosinus pour définir cette loi de pan. On multiplie l'entrée par le cosinus de $\frac{\pi}{2}$ multiplié par le paramètre $p$ pour donner l'une des sorties et l'on multiplie l'entrée par le sinus de $\frac{\pi}{2}$ multiplié par le paramètre $p$ pour donner l'autre sortie. Avec $0 \leq p \leq 1$ :

$$L\_Output = \cos(\frac{\pi}{2} \cdot p) \cdot Input,$$

$$R\_Output = \sin(\frac{\pi}{2} \cdot p) \cdot Input.$$

Ainsi, quand $p = 0$ :

$$L\_Output = \cos(0) \cdot Input,$$
$$R\_Output = \sin(0) \cdot Input.$$

Donc la variable L_Output est égale à l'entrée et R_Output est égale à 0. Le signal est donc entièrement routé dans le bus L_Output.

Inversement, quand $p = 1$ :

$$L\_Output = \cos(1) \cdot Input,$$

$$R\_Output = \sin(1) \cdot Input,$$

L_Output est égale à 0 et R_Output est égale à l'entrée. Le signal est donc entièrement routé dans le bus R_Output.

Enfin, lorsque $p = 0.5$ :

$$L\_Output = \cos(\frac{\pi}{4}) \cdot Input = \frac{\sqrt{2}}{2} \cdot Input \simeq 0,7071 \cdot, Input$$

$$R\_Output = \sin(\frac{\pi}{4}) \cdot Input = \frac{\sqrt{2}}{2} \cdot Input \simeq 0,7071 \cdot, Input$$

ce qui correspond à une attenuation de 3 dB.

## C.1 Code du plug-in de volume et de panoramique - Partie traitement du signal

```
//=====================
// Main Process
// =====================
Int lMinNumberOfChannels = Flux::Min(
    lNumberOfSourceChannels, lNumberOfTargetChannels);

if ((lNumberOfSourceChannels == 1 || params.nMonoInput) &&
    lNumberOfTargetChannels == 2)
{
    SampleType fPanValue = Math::LinearScale(params.fPan,
        1.0, -100.0, 100.0, 0.0, 1.0);

    SampleType fGainPanL = cos(dPi * 0.5 * fPanValue);
    SampleType fGainPanR = sin(dPi * 0.5 * fPanValue);

    m_Gain[0].Set(params.fLinearGain * fGainPanL);
    m_Gain[1].Set(params.fLinearGain * fGainPanR);

    m_Gain[0].Process(ppfSource[0], ppfTarget[0],
        lNumberOfSamplesPerChannel);
    m_Gain[1].Process(ppfSource[0], ppfTarget[1],
        lNumberOfSamplesPerChannel);
}
else
{
    Basic::Copy(ppfSource, ppfTarget, lMinNumberOfChannels,
        lNumberOfSamplesPerChannel);
```

```
    m_Gain[0].Set(params.fLinearGain);
    m_Gain[0].Process(ppfTarget, lNumberOfTargetChannels,
        lNumberOfSamplesPerChannel);
}

if (params.nRevertPhase)
{
    for (Int lChannel = 0; lChannel < lMinNumberOfChannels;
        lChannel++)
    {
            SampleType* pfTarget    = ppfTarget[lChannel];
        for (Int lSample = 0; lSample <
            lNumberOfSamplesPerChannel; lSample++)
        {
            pfTarget[lSample] = -pfTarget[lSample];
        }
    }
}
```

**Résumé**

Lors d'un mixage audio-numérique, les signaux audio-numériques de chaque piste sont traités puis additionnés entre eux pour constituer un nouveau signal audio-numérique qui sera diffusé ou stocké.

La sommation numérique est l'opération de traitement du signal numérique qui correspond à cette addition des différents signaux audio-numériques à mixer.

Dans ce mémoire, nous voulons déterminer s'il existe des différences entre plusieurs versions d'un même mixage, réalisées par des logiciels de mixage audio-numérique différents. Nous explorons donc les représentations binaires utilisées pour coder les signaux audio-numériques, notamment la représentation binaire en virgule flottante. La programmation d'un plug-in nous permet l'écoute des différences entre des traitements réalisés dans plusieurs précisions en représentation en virgule flottante. Enfin, nous rendons compte des résultats de tests réalisés avec plusieurs logiciels effectuant un même mixage.

Mots clefs : Traitement du signal numérique, Représentation binaire en virgule flottante, Erreurs d'arrondi, Sommation numérique, Mixage audio.

**Abstract**

During a digital audio mixing, audio-digital signals from each track are processed and added to build up a new audio digital signal which will be broadcasted or recorded.

Digital summing is the digital signal processing operation that corresponds to this addition of all the different digital audio signals which has to be mixed.

In this dissertation, we want to determine if some differences exist between various versions of a unique sound produced from different digital audio workstation mixing software. Thus, we explore the binary representations used to code digital audio signals and more particularly the floating point binary representation. The programming of a plug-in enables us to listen to the differences between some digital audio processing operations computed in various precisions of the floating point binary representation. Finally, we show the results of tests performed with some different software computing a unique sound mixing.

Keywords : Digital signal processing, Floating point binary representation, Round-off error, Digital summing, Audio mixing.

# Bibliographie


[1] David Bindel et Jonathan Goodman. *Principles of Scientific Computing Sources of Error*. 2009.

[2] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 1: Basic Architectures*. 2011.

[3] Jonas Ekeroot et Jan Berg. *Audio software development an audio quality perspective*. Audio Engineering Society, 124th Convention,Amsterdam. Convention Paper 7438. 2008.

[4] Dr. Anita Goel. *Binary Arithmetic and Binary Coding Schemes*. Dyal Singh College, University of Delhi.

[5] David Goldberg. « What Every Computer Scientist Should Know About Floating-Point Arithmetic ». Dans : *ACM Computmg Surveys* 23.1 (1991), p. .

[6] Chan Chi Hin. *Floating-point arithmetic*. University of Hong-Kong, Departement of Mathematics. 2011. url : `http://www.math.cuhk.edu.hk/course/math3230b/320bnote3.pdf`.

[7] Prof. W. Kahan. *Lecture Notes on the Status of IEEE 754*. Elect. Eng. and Computer Science University of California. 1996.

[8] Rukku Linna. « Lecture 1: Number representation and errors ». Dans : *Computational Science*. 2011.

[9] Bob Norin Marius Cornea-Hasegan. *IA-64 Floating-Point Operations and the IEEE Standard for Binary Floating-Point Arithmetic*. Rap. tech. Intel Technology Journal Q4, Microprocessor Products Group, Intel Corporation, 1999.

[10] J. Senpau Roca A. Priou. *Numeration et codages*. Institut Universitaire de Technologie Cachan - Departement Geii Electronique. 2003.

[11] Daniel Robert. url : `http://daniel.robert9.pagesperso-orange.fr/Digit/Digit_7TS.html`.

[12] Paul Zimmermann Vincent Lefevre. *Arithmetique flottante*. Institut National de Recherche en Informatique et en Automatique. 2004.

[13] Randy Yates. *Fixed-Point Arithmetic: An Introduction*. Rap. tech. Digital Signal Labs, 2009.