VILLE DE LIÈGE

Institut de Technologie
Enseignement de Promotion sociale

Année académique 2021 – 2022

Développement d'un codec audio AAC : optimisation de l'algorithme MDCT pour l'architecture ARM

Étudiante :
Laura Binacchi

Lieu de stage :
EVS Broadcast Equipment
Rue du Bois Saint-Jean 13, 4102 Ougrée

Maître de stage :
Bernard Thilmant
Software Engineer

Épreuve intégrée présentée pour l'obtention du diplôme de
BACHELIER.E EN INFORMATIQUE ET SYSTÈMES
FINALITÉ : INFORMATIQUE INDUSTRIELLE

Remerciements

Table des matières

Table des figures

Glossaire

ARM, Codec, FFT, fixed point, MDCT, Ne10, NEON, twiddling, wrapper, SIMD, ARM NEON/Advanced SIMD, floating point, fixed point

twiddling : les opérations de twiddling sont les opérations de réarrangement des données de la MDCT effectuées autour de l'appel à la FFT pre-twiddling : nom donné aux opérations de pre-processing effectuées avant l'appel à la FFT post-twiddling : nom donné aux opérations de post-processing effectuées après l'appel de la FFT VFP : nom donné à la FPU de l'ARM FPU broadcast

Introduction

Ce travail présente les résultats d'un stage consacré à l'optimisation de l'encodeur audio AAC. Il s'inscrit dans la continuité du travail d'analyse de Wafaa Heddari, étudiante qui m'a précédée au sein de l'équipe Hardware-Firmware d'EVS. Son stage a débouché sur une synthèse d'optimisations possibles du codec AAC. Après avoir pris connaissance de son travail, mon stage a eu pour but de concrétiser certaines des améliorations possibles du codec.

Il a été décidé que mon stage se concentrerait sur le bloc MDCT de l'encodeur AAC. La MDCT (*Modified Discrete Cosine Transform*) est le bloc de transformation du signal audio du domaine temporel vers le domaine fréquentiel. La compression du codec AAC repose sur l'utilisation d'un modèle psychoacoustique qui détermine les informations à éliminer en partie sur base de l'analyse fréquentielle du signal.

L'algorithme de MDCT développé dans ce travail est un algorithme basé sur un autre type de transformation : la FFT *Fast Fourier Transform*. Mon travail a débuté par l'implémentation, sur base d'un code d'exemple, de cet algorithme. Ce code d'exemple a ensuite été adapté à l'utilisation de la FFT de la librairie *Ne10*.

Une des optimisations de la MDCT consiste à passer d'une arithmétique *floating point* à une *arithmétique fixed point*. La logique à mettre en œuvre pour les opérations en *floating point* est plus lourde que celle des opérations sur des entiers. L'arithmétique *fixed point* offre généralement des performances plus intéressantes pour les applications de traitement du signal. Ce travail montrera que ce n'est pas le cas pour les processeurs équipés d'un FPU (*Floating Point Unit*) performant comme celui du processeur ARM.

L'utilisation d'instructions SIMD (*Single Instruction on Multiple Data*) est l'optimisation qui permettra réellement à l'algorithme de gagner en performances. Les instructions SIMD reposent sur le principe de vectorisation des données pour pouvoir opérer simultanément une même instruction sur plusieurs données. Les opérations SIMD spécifiques au processeur ARM sont appelées ARM NEON ou *Advanced SIMD*.

La MDCT a été développée spécifiquement pour une architecture reposant sur un processeur ARM. C'est l'architecture utilisée pour la nouvelle génération du serveur XT-VIA d'EVS, serveur pour lequel est développé le codec AAC. Tous les tests ont été réalisées sur un Raspberry Pi 4 équipé d'un processeur ARMv7.

Les différentes itérations de la MDCT ont été testées dans le double objectif de gagner le maximum de performances tout en perdant le moins possible en précision. Les données de sortie des MDCT ont systématiquement été validées par comparaison avec les spectres de fréquences générés par un algorithme de référence. Ces comparaisons sont présentées pour chaque étape du développement. La section consacrée au protocole de validation présente les résultats pour la dernière itération de la MDCT générés avec un panel plus complet de signaux d'entrée.

Les performances constituent l'aspect le plus important du développement. La qualité des données audiovisuelles et leur quantité sont de plus en plus importantes. Dans le *broadcast*, les codec sont utilisés en boucle sur de nombreux flux de données en parallèle. La production en *live* a des exigences de temps réel pour les enregistrements et les diffusions.

Les itérations de la MDCT développées au cours de ce travail sont toutes présentées en annexes. L'aboutissement de mon stage est une implémentation de la MDCT basée sur la FFT de la librairie *Ne10* en arithmétique *fixed point* avec optimisations NEON (annexe L).

1 EVS Broadcast Equipment

1.1 Présentation d'EVS et du département R&D

Mon stage s'est déroulé au sein de la société EVS Broadcast Equipment. EVS est une entreprise d'origine liégeoise devenue internationale. Fondée en 1994 par Pierre L'Hoest, Laurent Minguet et Michel Counson, EVS compte aujourd'hui plus de 600 employés dans plus de 20 bureaux à travers le monde mais son siège principal se situe toujours à Liège[1].

EVS est devenu leader dans le monde du broadcast avec ses serveurs permettant l'accès et la diffusion instantanée des données audiovisuelles enregistrées sur ses serveurs. L'entreprise est également célèbre pour ses ralentis instantanés. Ces technologies sont utilisées pour la production live des plus importants événements sportifs dans le monde : le matériel EVS est notamment utilisé pour la retransmission des Jeux Olympiques depuis 1998.

Plus de 50% des employés d'EVS travaillent en recherche et développement afin de répondre au marché du broadcast en constante évolution. Outre ses solutions techniques innovantes, EVS se différencie de ses concurrents par la proximité entretenue avec les clients en leur proposant des solutions à l'écoute de leurs besoins et en leur offrant un service de support de qualité.

C'est en R&D, dans l'équipe Hardware-Firmware, que s'est déroulé mon stage. Sous la direction de Justin Mannesberg, cette équipe se compose d'une vingtaine d'employés spécialisés en développement embarqué et en développement FPGA. La situation particulière dans laquelle s'est déroulé mon stage, en pleine pandémie de Covid et alors que tous les employés étaient confinés, ne m'a pas permis d'interagir avec beaucoup de membres de l'équipe ni de pouvoir observer leur travail. Bernard Thilmant (Software Engineer dans l'équipe Hardware-Firmware) a cependant réussi à m'apporter le soutien nécessaire à la bonne réalisation de mon stage : il m'a permis de m'initier au C++, m'a aidée à ne pas me perdre dans les concepts parfois complexes de l'encodage audio et m'a aidée à apporter la rigueur scientifique nécessaire à la réalisation de mon travail. J'ai également pu bénéficier de l'expertise technique de Frédéric Lefranc (Principal Embedded System Architect dans l'équipe Hardware-Firmware) ainsi que du suivi de Justin Mannesberg (Manager de l'équipe Hardware-Firmware).

1.2 Le serveur XT

Figure 1 – Vues avant et arrière (en configuration IP) de l'XT-VIA

EVS développe et commercialise de nombreux produits allant des serveurs de production aux interfaces permettant d'exploiter des données audiovisuelles ou de monitorer des systèmes de production[2]. Le serveur de production live XT est un des produits emblématiques d'EVS. Il permet de stocker de grandes quantités de données audiovisuelles et d'y accéder en temps réel afin de répondre aux besoins de la production en live. La remote LSM (*Live Slow Motion*) permet d'accéder aux contenus des serveurs XT afin de créer les ralentis pour lesquels EVS est célèbre dans le monde.

Le serveur XT a connu plusieurs versions : XT, XT2, XT2+, XT3 et enfin l'XT-VIA. L'XT-VIA (cf. figure 1), la plus récente version du serveur XT, en quelques informations clés[2] :

- offre un espace de stockage de 18 à 54 TB, soit plus de 130h d'enregistrement en UHD-4K ;
- dispose de 2 à plus de 16 canaux selon le format choisi : 2 canaux en UHD-8K (4320p), 6 canaux en UHD-4K (2160p) et plus de 16 canaux en FHD and HD (720p, 1080i, 1080p) ;
- permet une configuration hybride de ses entrées et sorties en IP (10G Ethernet SFP+, 100G en option, ST2022-6, ST2022-7, ST2022-8, ST2110, NMOS IS-04, IS-05, EMBER+, PTP) ou SDI (1.5G-SDI, 3G-SDI et 12G-SDI) ;
- supporte de nombreux formats d'encodage vidéo : UHD-4K (XAVC-Intra et DNxHR), HD/FHD (XAVC-I, AVC-I, DNxHD et ProRes), PROXY (MJPEG et H264) ;
- peut enregistrer 192 canaux audio non compressés et supporte les standards AES et MADI ;
- offre de nombreuses possibilités de connexion avec du matériel EVS ou non.

C'est pour la dernière génération du serveur XT, l'XT-VIA, que le codec AAC est développé. La compression avec perte de données de ce codec permet d'optimiser l'espace occupé par les données audio sans en altérer la qualité perçue. Outre la qualité audio, les performances de l'encodage sont importantes à prendre en compte pour permettre l'enregistrement de plusieurs canaux en parallèle tout en conservant un traitement de l'information qui tienne le temps réel. L'optimisation des performances doit tenir compte de l'architecture de l'XT-VIA : le processeur ARM remplace le processeur Intel de ses prédécesseurs avec des différences importantes dans les fonctions intrinsèques.

2 L'encodage audionumérique : généralités

2.1 Le son

2.2 La numérisation d'un signal

3 Les codecs audio

3.1 Définition d'un codec

Un codec est un procédé logiciel composé d'un encodeur (***co**der*) et d'un décodeur (***dec**oder*)[3]. Un codec audio permet donc, d'une part, de coder un signal audio dans un flux de données numériques et, d'autre part, de décoder ces données afin de restituer le signal audio.

Les codecs sont dits avec perte (*lossy*) ou sans perte (*lossless*). Le PCM est par exemple un codec sans perte puisqu'il encode la totalité des informations sonores dans la bande de fréquences humainement audibles. Ce type de codec permet de conserver la qualité de l'audio mais nécessite en contrepartie un espace de stockage conséquent, même avec une compression des données.

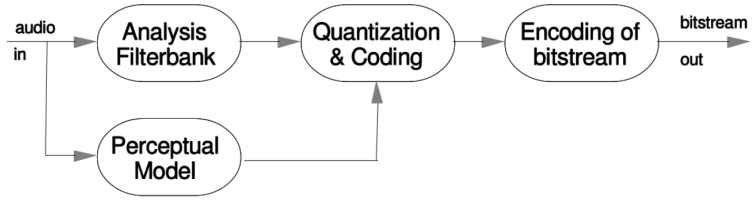
Afin de réduire l'espace de stockage nécessaire, les codecs avec perte permettent de supprimer une partie des données audio. C'est le cas des codecs définis par les normes MPEG dont fait partie le codec AAC.

3.2 Les codecs MPEG

MPEG (*Moving Picture Experts Group*) désigne une alliance de différents groupes de travail définissant des normes d'encodage, de compression, de décompression et de transmission de média audio, vidéo et graphiques[4]. MPEG est actif depuis 1988 et a produit depuis de nombreuses normes.

Les codecs audio qui implémentent les normes MPEG ont pour point commun d'être des codecs avec perte de données basés sur un modèle psychoacoustique. Le premier est le MP3, défini par la norme MPEG-1 Layer-3 ISO/IEC 11172-3 :1993. Le codec AAC est conçu en 1997 pour remplacer le MP3. Il est défini par les normes MPEG-2 partie 7 ISO/IEC 13818-7 :2006[5] et MPEG-4 partie 3 ISO/IEC 14496-3 :2019[6].

Les normes MPEG définissent les grandes lignes de l'encodage et du décodage ainsi que le format du conteneur mais pas l'implémentation du codec qui peut de ce fait être plus ou moins performant. Les codecs MPEG sont typiquement composés des blocs suivants[7] :



(a) Schéma fonctionnel de l'encodeur

(b) Schéma fonctionnel du décodeur

Figure 2 – Vue simplifiée d'un codec MPEG basé sur un modèle psychoacoustique

L'encodeur est composé des blocs suivants :

filter bank la banque de filtres décompose le signal temporel d'entrée en différentes composantes fréquentielles

perceptual model le modèle psychoacoustique utilise le signal temporel et/ou sa décomposition fréquentielle pour éliminer les données audio dont l'absence ne nuira pas à la qualité perçue à l'écoute

quantization and coding la quantification attribue une valeur numérique aux données du spectre de fréquences : elles sont typiquement codées avec une méthode entropique qui peut être optimisée avec le modèle psychoacoustique

encoding of bitstream les données sont formatées en un flux contenant typiquement le spectre de fréquences codé et des informations supplémentaires permettant l'encodage

Le décodeur a un fonctionnement inverse : le flux de données est décodé (**decoding of bitstream**), les composantes fréquentielles du signal sont retrouvées par l'opération inverse à la quantification (**inverse quantization**) et ces sous-bandes fréquentielles sont finalement rassemblées pour reconstituer le signal temporel (**synthesis filter bank**).

Le fonctionnement du décodeur ne sera pas plus développé dans ce travail car le bloc MDCT fait partie de la banque de filtres de l'encodeur. Le fonctionnement spécifique de l'encodeur AAC sera par contre détaillé dans la section 4.

3.3 Les modèles psychoacoustiques

Le section précédente a défini les codecs MPEG comme étant basés sur un modèle psychoacoustique. La psychoacoustique est une branche de la psychophysique qui étudie la manière dont l'oreille humaine perçoit le son[8]. Cette discipline permet d'améliorer la compression d'un signal audio en éliminant les sons qui sont captés par un microphone mais qui ne peuvent pas être perçus par l'oreille humaine et les avancées dans cette discipline permettent de développer des encodeurs audio de plus en plus performants. Les codecs basés sur un modèle psychoacoustique sont toujours des codecs avec perte puisqu'une partie des informations auditives sera définitivement perdue, ce qui ne nuit pour autant pas à la qualité perçue du son.

L'encodage audionumérique tient déjà compte des seuils de fréquences humainement audibles pour limiter les données audio enregistrées : nous l'avons vu dans la section 2.2, aucun son n'est perçu en-deça de 20 Hz ou au-delà de 20 kHz. La psychoacoustique permet de mieux dessiner la limite entre ce qui est humainement audible ou non afin d'éliminer un maximum des informations non pertinentes et ainsi augmenter le facteur de compression des données : le facteur de compression des codecs MPEG est environ 15 fois supérieur à celui du CD[9].

Les effets de masque sont au centre des différents modèles psychoacoustiques utilisés pour la compression audio. L'enjeu afin d'obtenir le meilleur taux de compression est de calculer le plus finement possible les seuils de masquage, i.e. la limite entre les informations pertinentes et celles qui peuvent être éliminées. Les effets de masque dans le domaine fréquentiel (*spectral masking effects*) sont parmi les plus utilisés mais il en existe d'autres, e.g. dans le domaine temporel.

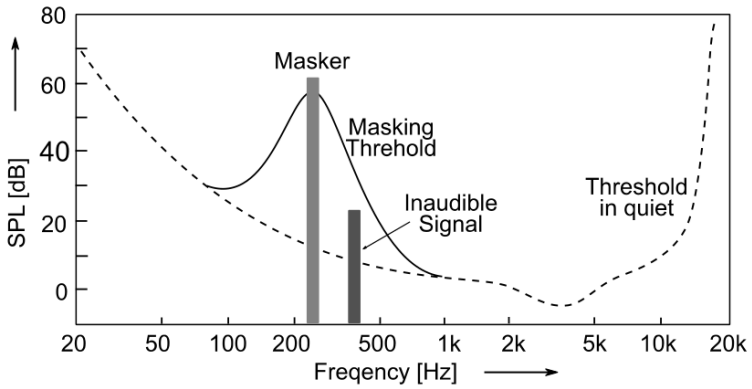


FIGURE 3 – Effets de masque dans le domaine fréquentiel

La figure 3 représente différents effets de masque du domaine fréquentiel.

4 Le codec AAC

4.1 Fonctionnement de l'encodeur AAC

4.2 Le bloc MDCT

- overlap

5 Environnement de développement

Les tests des différentes itérations de la MDCT ont été faits sur un Raspberry Pi 4. Le Raspberry est un *single-board computer* (SBC) développé par la Raspberry Foundation et Broadcom depuis 2012. Le modèle 4B utilisé pour ce travail (figure 4) date de 2019. Il a été choisi car il possède un processeur ARM, processeur pour lequel les optimisations de la MDCT doivent être développées[10].

FIGURE 4 – Raspberry Pi 4

Les informations sur le CPU du Raspberry Pi 4 se trouvent dans le fichier `/proc/cpuinfo` (**annexe A**) :

- le Raspberry Pi 4 dispose de 4 CPU ;
- chacun de ces processeurs est un ARMv7 ;
- les processeurs disposent, entre autres, des instruction NEON et des instructions VFP.

Les instructions ARM NEON sont les instructions SIMD (*Single Instruction on Multiple Data*) de l'ARMv7. C'est l'utilisation de ces instructions dans la dernière itération de la MDCT qui va lui permettre de réellement gagner en performances.

Le Raspberry Pi 4 possède également des instructions VFP (*Vector Floating Point*) qui offrent des performances très intéressantes pour les opérations en *float*. Les instructions VFP ne sont pas liées au instructions NEON : la plupart des ARM possèdent les deux mais ce n'est pas toujours le cas. Les mesures de performances de différentes MDCT développées dans ce travail mettront plusieurs fois en évidence l'influence de ces instructions sur les très bonnes performances observées dans les algorithmes en *floating point*.

Le développement de ce travail s'est fait uniquement sous OS Linux et principalement en *remote* sur le Raspberry : seule la première itération de la MDCT a pu être testée sur une machine équipée d'un processeur Intel. Il aurait été intéressant de maintenir une implémentation de référence de la MDCT compatible Intel mais l'utilisation de la librairie *Ne10* ne l'a pas permis.

Le code a été développé dans Visual Studio Code. Cet éditeur de texte offre des extensions pour le développement en C et C++. Il permet de se connecter facilement en SSH au Raspberry pour le développement en *remote*.

Le code est compilé avec GCC (version 8.3.0) avec les commandes CMake présentées à l'**annexe B** : un fichier CMake principal est placé à la racine du projet et permet d'appeler les fichiers CMake de la librairie *Ne10* et du projet *audio_encoding* contenant les codes et les tests des différentes MDCT. Le fichier CMake utilisé pour la génération des exécutables de test n'est pas présenté d'un bloc dans ce travail. Les commandes utilisées pour générer les exécutables seront présentées dans les annexes à chaque fois sous le code qu'elles permettent de compiler.

Le projet est développé en C++. Ce choix peut paraître étonnant pour le développement d'un codec, traditionnellement développé en C. Hormis quelques spécificités de C++, comme l'utilisation des classes, le code présenté dans ce travail est très procédural puisqu'il consiste principalement à développer une seule fonction : le bloc MDCT. Le développement en C++ m'a permis d'apprendre ce langage majoritairement utilisé pour les développements chez EVS.

6 Algorithmes MDCT de référence

La première étape de ce travail consiste à développer des algorithmes de référence afin de valider les différentes MDCT implémentées par la suite. Ces algorithmes de référence sont développés en algorithmique flottante sur base de la formule mathématique de la MDCT. Ils permettent de générer des spectres de fréquences en *float* ou en *integer* afin de valider les données de sortie des MDCT optimisées.

6.1 Description mathématique de la MDCT

La transformation effectuée par la MDCT est donnée par l'équation suivante[11] :

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

X_k avec $k \in [0, N[$ pour une fenêtre d'entrée de 2N échantillons

x_n avec $n \in [0, 2N[$: la fenêtre d'entrée

$F : R^{2N} \rightarrow R^N$ la MDCT est une fonction linéaire qui pour 2N nombres réels en entrée produit N nombres réels en sortie

À cette équation s'ajoute le facteur d'échelle $\frac{2}{\sqrt{2N}}$.

La MDCT a été implémentée avec une fenêtre d'entrée de $2N = 1024$ échantillons : $N = 512$ échantillons utiles et $N = 512$ échantillons pour l'*overlap*. Le bloc de sortie, i.e. le spectre de fréquences de la fenêtre d'entrée a une taille de $N = 512$ composantes fréquentielles en nombres complexes, ou de $\frac{N}{2} = 256$ composantes fréquentielles en nombres réels. Ces valeurs, utilisées à de très nombreux endroits du code, sont rassemblées dans le header `mdct_constants.h` présenté dans l'**annexe C**. Ce fichier contient également d'autres valeurs pré-calculées sur base de la taille de la fenêtre d'entrée.

La section suivante présente deux implémentations simples de cette formule. Ces implémentations ne pourraient pas être utilisées sans avoir été optimisées car elles seraient beaucoup trop lentes pour un codec qui doit tenir le temps réel sur plusieurs canaux.

La complexité de l'implémentation de cette formule est de $O(N^2)$ opérations (où N est la taille de la fenêtre d'entrée). Cette complexité peut être ramenée à $O(N \log N)$ opérations par une factorisation récursive. La complexité peut également être diminuée en se basant sur une autre transformation, e.g. une DFT (*Discrete Fourier Transorm*) ou une autre DCT (*Discrete Cosine Transform*). La complexité sera alors de $O(N)$ opérations de *pre-* et *post-processing* en plus de la complexité de la DFT ou de la DCT choisie[11]. C'est cette dernière option qui a été retenue pour ce travail.

6.2 Implémentations des MDCT de référence en *floating point* et *fixed point*

La formule mathématique de la MDCT a été implémentée très simplement en algorithmique flottante avec la possibilité d'obtenir le spectre de fréquences codé en *float*, *double* ou *integer* sur 32 bits (**annexe D**). L'objectif de ces implémentations est de pouvoir valider les spectres de fréquences calculés par les implémentations optimisées de la MDCT. Les MDCT de référence serviront également à mesurer la précision des MDCT optimisées.

La première implémentation de l'équation de la MDCT est présentée dans l'**annexe D.1** et représentée par la figure 5. Les calculs et les résultats peuvent être effectués et obtenus aussi bien en *float* (32 bits) qu'en *double* (64 bits) grâce à l'utilisation d'un *template*. Le signal temporel a la même précision (*float* ou *double*) que le spectre généré.

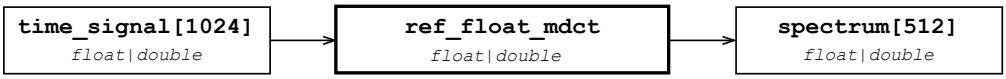


FIGURE 5 – Fonction `ref_float_mdct`

La seconde fonction de référence est présentée dans l'**annexe D.2** et représentée par la figure 6. Elle permettra de vérifier les résultats des implémentations optimisées en *fixed point*. Tous les calculs sont faits en algorithmique flottante (*double* pour plus de précision) avant de transtyper le spectre de fréquences en *integer* sur 32 bits.

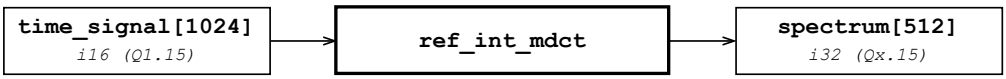


FIGURE 6 – Fonction `ref_int_mdct`

La représentation *fixed point* du spectre de fréquences correspond à une notation Qx.15 signée. Le Q15 a été choisi car il était souhaité que l'algorithme MDCT soit implémenté entièrement en 16 bits. Cela n'a pas été possible à cause d'une trop grande perte de précision (cf. section 9 consacrée au passage en arithmétique *fixed point* de la MDCT). La mise à échelle Q15 a été conservée mais une autre représentation pourrait facilement être mise en place selon les besoins du bloc suivant.

6.3 Validation des MDCT de référence

Les MDCT de référence sont validées en générant le spectre de fréquences d'un signal connu pour pouvoir observer que celui-ci corresponde bien à ce qui est attendu. Tout au long de ce travail, les MDCT ont été validées avec divers signaux en entrée : des signaux à une ou plusieurs fréquences connues ou des données aléatoires. La section 11.1 consacrée au protocole de validation donne plus de détails sur la génération des spectres de fréquences produits par les MDCT et leur interprétation. Afin de faciliter la lecture des résultats et les comparaisons, ce travail présentera principalement des tests effectués avec un signal à 440 Hz en entrée des MDCT.

L'**annexe E** présente les codes permettant de générer les signaux sinusoïdaux destinés à être donnés en entrée aux MDCT. Les signaux sinusoïdaux sont produits en *floating point* (**annexe E.1**) ou en *fixed point* (**annexe E.2**). Le signal est paramétrable :

- en taille ou nombre d'échantillons : la fenêtre d'entrée de la MDCT sera toujours de 1024 échantillons ;
- en amplitude : paramètre utilisé pour limiter le signal à une certaine *range* de valeurs ;
- en fréquence : les résultats présentés dans ce travail utilisent le plus souvent une fréquence de 440 Hz mais les MDCT ont été testées avec de nombreux autres signaux ;
- en déphasage : ce paramètre n'a pas été utilisé ;
- et en fréquence d'échantillonnage toujours à 48 kHz.

Dans ce travail, la *range* du signal est toujours comprise entre −0.9 et 0.9, que le signal soit codé en *floating point* ou en *fixed point*. Cette limitation de la *range* vise à reproduire la *headroom* du signal temporel, i.e. l'espace réservé aux valeurs limites du signal : le signal réel est clippé et ne couvre pas l'ensemble des valeurs possibles. Le fait de ne pas coder le signal sur toutes les valeurs possibles est moins important pour les algorithmes travaillant en *floating point* mais il sera essentiel pour la mise à échelle des données en représentation *fixed point* (section 9).

Les signaux codés en *float* ou en *double* serviront à valider les implémentations intermédiaires de la MDCT. L'objectif reste cependant de développer une MDCT capable de traiter des données entières sans transtypage. Le signal en *integer* servira à valider les dernières itérations de la MDCT. Ce signal est codé en 16 bits (représentation Q15 signée) afin de correspondre aux données réelles qui seront reçues par l'algorithme.

La figure 7 offre une représentation graphique des signaux sinusoïdaux en *floating point* utilisés pour tester les MDCT en *floating point* :

- les deux signaux sont équivalents : le signal généré en *float* (en pointillés) se superpose parfaitement au signal généré en *double* ;
- la période du signal est de 2.3 ms correspondant à la fréquence de 440 Hz attendue ;
- les valeurs du signal sont comprises entre −0.9 et 0.9 pour respecter une *headroom*.

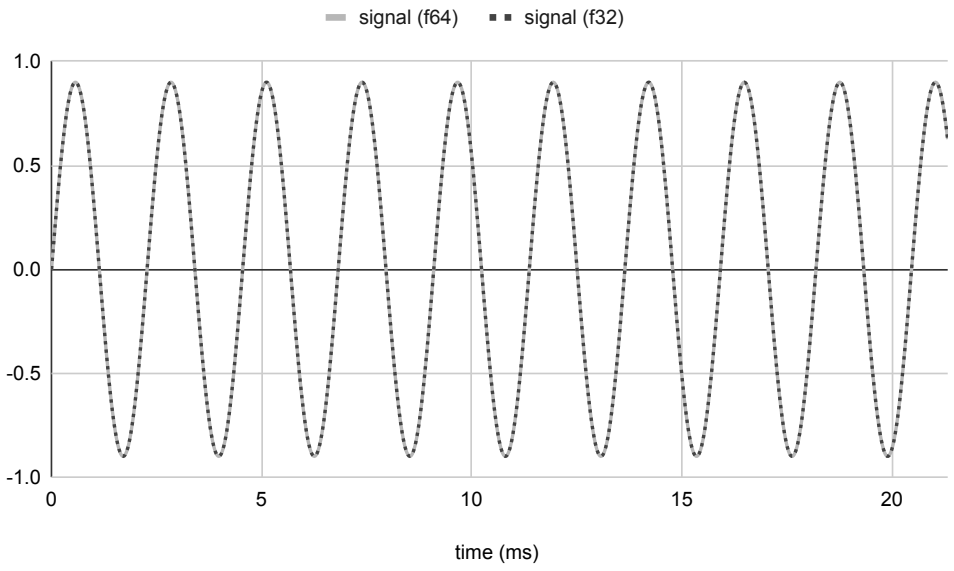


Figure 7 – Signal sinusoïdal à 440 Hz codé en *floating point* (*float* et *double*)

Les MDCT de référence en *floating point* prenant en entrée ces signaux sinusoïdaux à 440 Hz génèrent les spectres de fréquences présentés à la figure 8 :

- les deux MDCT génèrent le même spectre : le spectre généré par l'algorithme en *float* (en pointillés) se superpose parfaitement à celui généré en *double* ;
- la représentation graphique des spectres de fréquences met bien en évidence une composante fréquentielle principale aux alentours de 440 Hz (la section 11.1 expliquera pourquoi la résolution du spectre de fréquences ne permet pas de trouver exactement une fréquence à 440 Hz).

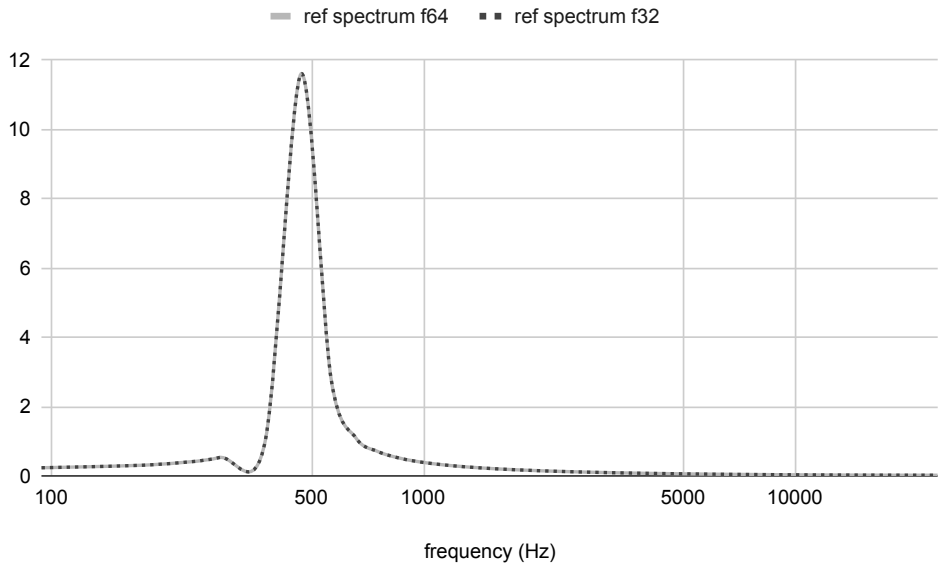


Figure 8 – Spectres de fréquences générés par les MDCT de référence en *floating point* (440 Hz)

Le résultat obtenu est déjà satisfaisant puisque la fréquence mise en évidence par le spectre est bien celle du signal d'entrée. Cependant, l'implémentation d'une fonction de fenêtre appliquée au signal d'entrée permettrait d'améliorer encore le spectre de fréquences généré par la MDCT. La section 12 consacrée aux améliorations possibles de ce travail revient sur cette fonction de fenêtre.

Le signal sinusoïdal en *fixed point* utilisé pour la validation des MDCT *fixed point* est représenté par la figure 9 :

- le signal prend la même forme que ceux en *floating point* (figure 7) ;
- la période du signal de 2.3 ms correspond à la fréquence de 440 Hz attendue ;
- le signal varie entre 29491 et −29491, valeurs qui corespondent aux valeurs limites du signal en *floating point* mises à échelle pour une représentation *fixed point* Q15.

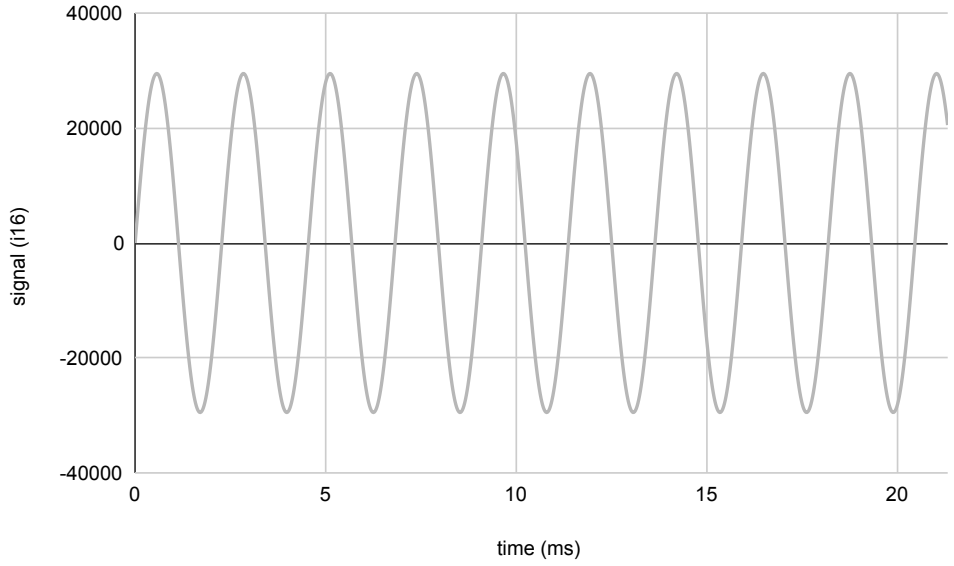


FIGURE 9 – Signal sinusoïdal à 440 Hz codé en *fixed point* (Q15)

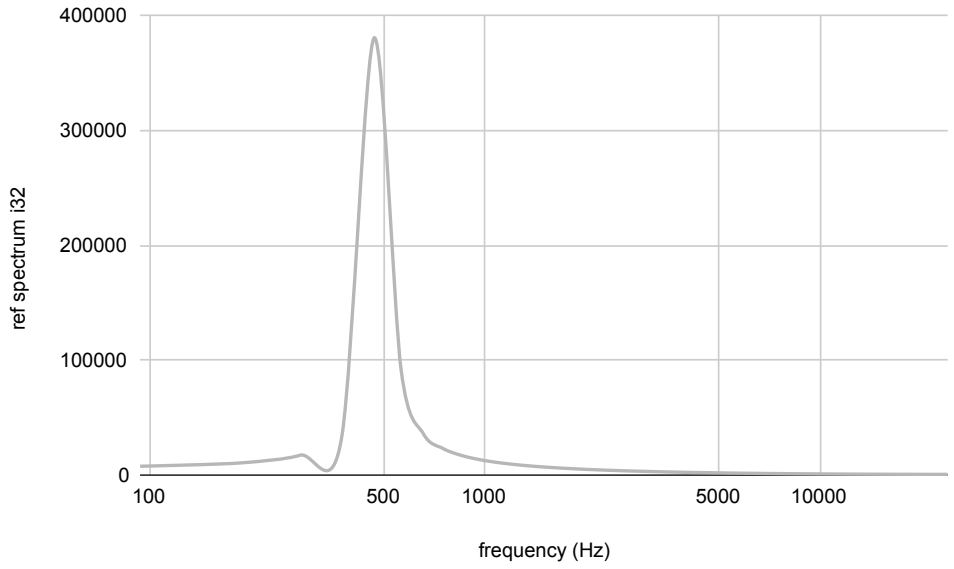


FIGURE 10 – Spectre de fréquences généré par la MDCT de référence en *fixed point* (440 Hz)

La représentation graphique du spectre de fréquences généré par la MDCT de référence en *fixed point* à partir du signal à 440 Hz est présentée par la figure 10 :

- le spectre prend la même forme que le spectre de fréquence généré par les MDCT en *floating point* (figure 8) ;
- le spectre met bien en évidence une composante fréquentielle principale aux alentours de 440 Hz ;
- les valeurs des composantes fréquentielles sont beaucoup plus grandes que celles du spectre en *floating point* puisqu'elles sont mises à échelle Q15.

7 Algorithme MDCT basé sur la FFT

Il a été décidé en amont de ce travail d'implémenter une MDCT basée sur une FFT existante. À cette fin, un code d'exemple m'a été fourni[12]. Ce code d'exemple fait appel à la FFT de la librairie *FFTW3*. Cette FFT est disponible en *float* ou en *double* mais cette section ne présentera que son utilisation en *float*.

Le principe de cette implémentation de la MDCT est de pré-processer le signal temporel pour pouvoir le faire passer dans une FFT déjà optimisée puis de post-processer la sortie de la FFT afin de reconstituer le spectre de fréquences de la MDCT (figure 11). Seules les opérations de *pre-* et de *post-processing* (aussi appelées *pre-* et *post-twiddling*) devront alors être optimisées.

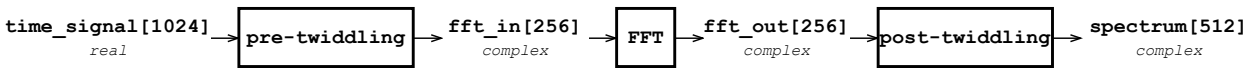


Figure 11 – Schéma fonctionnel de la MDCT

Les FFT de *FFTW3* ne sont disponibles qu'en arithmétique *floating point*. Cette librairie ne pourra donc pas être conservée pour les itérations suivantes de la MDCT. L'implémentation de la MDCT présentée dans cette section vise uniquement à valider l'algorithme de MDCT fourni puisqu'il constitue la base de toutes les MDCT développées dans ce travail.

7.1 Utilisation de la librairie *FFTW3*

FFTW est une librairie développée au MIT afin de proposer des algorithmes de FFT[13]. *FFTW 3.3.9* était la dernière version disponible de la librairie au moment de la réalisation de ce travail. Elle a récemment été mise à jour avec la version 3.3.10.

La librairie s'installe simplement en téléchargeant les sources sur le site de *FFTW*[14] et en les compilant avec le Makefile fourni. La librairie fonctionne par défaut en *double* mais elle peut également fonctionner en *float* en activant l'option `–enable-float` à la configuration. Le header `<fftw3.h>` doit ensuite être inclus dans les fichiers source faisant appel à la FFT. Enfin, la librairie doit être linkée à l'exécutable lors de la compilation.

7.2 Implémentation de la MDCT basée sur la FFT de la librairie *FFTW3*

L'implémentation de la MDCT basée sur la FFT de *FFTW3* en *float* sur 32 bits est présentée à l'**annexe F**. Le code est structuré de la manière suivante :

- Le header définit la classe `fftw3_mdct_f32` : les données qu'elle contient et les prototypes de ses fonctions (constructeur, destructeur et fonction MDCT) ;
- le constructeur initialise les facteurs de *pre-* et de *post-processing* et la configuration de la FFT ;
- le destructeur libère la mémoire allouée dans le construucteur ;
- la fonction MDCT opère les opérations de *pre-processing*, fait appel à la FFT et opère les opérations de *pos-processing*.

Le header de la classe `fftw3_mdct_f32` (**annexe F.1**) définit les données suivantes :

- `fft_plan` : la configuration de la FFT de *FFTW3* ;
- `fft_in` et `fft_out` : les tableaux contenant les données d'entrée et de sortie de la FFT ;
- le tableau `twiddle` contenant les facteurs utilisés pour le *pre-* et le *post-processing*.

Le constructeur (**annexe F.2**) initialise :

- les facteurs de *twiddling* : le tableau contient en alternance un facteur calculé par un cosinus (aux index pairs) et un facteur calculé par un sinus (aux index impairs) ;
- la configuration de la FFT conformément à la documentation de *FFTW3*.

Le destructeur (**annexe F.3**) libère la mémoire allouée aux tableaux d'entrée et de sortie de la FFT et à la configuration de la FFT.

La fonction MDCT de la classe `fftw3_mdct_f32` est présentée dans l'**annexe F.4**. Elle opère les fonctions de *pre-twiddling* sur le signal temporel, appelle la FFT et opère les opérations de *post-twiddling* comme exposé par la figure 11. La fonction prend en entrée un signal temporel codé en *float* pour générer un spectre de fréquences lui aussi en *float* (figure 12).

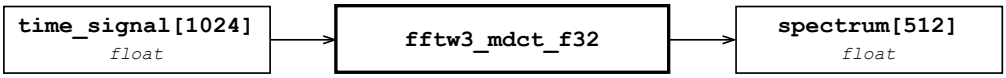


Figure 12 – Fonction `fftw3_mdct_f32`

Les opérations de *pre-twiddling* ont pour but de réduire de moitié la taille de la fenêtre d'entrée. Les transformations effectuées par les calculs fournis tranforment la fenêtre de 1024 échantillons temporels (réels) en une fenêtre de 256 nombres complexes. Ces nombres sont placés dans le tableau `fft_in` qui alterne les parties réelles et imaginaires des nombres complexes qu'il contient.

La FFT de *FFTW3* en *float 32* est appelée sur ce tableau. Le résultat de la FFT est placé dans le tableau `fft_out` lui aussi en nombre complexes. Le fait d'avoir réduit d'un quart la taille de la fenêtre d'entrée a permis de diminuer la complexité de l'algorithme.

Le spectre de fréquences est reconstitué par les opérations de *post-twiddling* en combinant le tableau de sortie de la FFT et les facteurs de *twiddling*. Ce spectre de fréquences contient également des nombres complexes dans un tableau à une dimension alternant les parties réelles et imaginaires de ces nombres.

7.3 Validation de la MDCT *FFTW3 float 32*

La MDCT *FFTW3 float 32* est validée par comparaison avec l'algorithme de référence développé dans la section précédente, en *double* pour plus de précisions. La MDCT a été validée avec de nombreux signaux en entrée, y compris avec des données aléatoires. Pour faciliter la lecture des résultats et la comparaison avec les résultats présentés dans les autres sections, le spectre de fréquences présenté à la figure (13) a été généré à partir d'un signal sinusoïdal à 440 Hz.

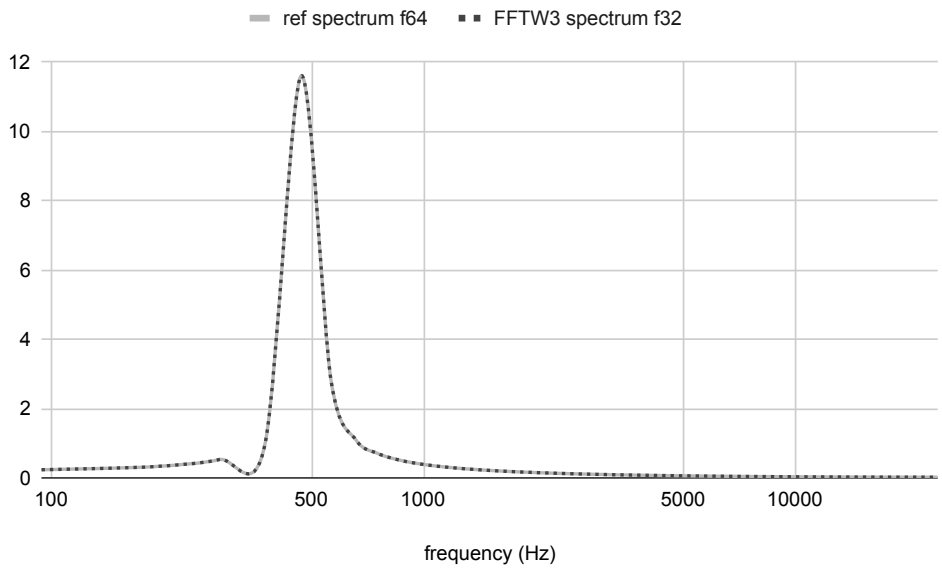


FIGURE 13 – Spectre de fréquences généré par la MDCT FFTW3 *float* comparé au spectre de référence (440 Hz)

Le spectre généré par la MDCT *FFTW3 float 32* se superpose parfaitement au spectre de fréquences de référence. Il fait bien apparaître une composante fréquentielle principale aux alentours de 440 Hz.

L'algorithme basé sur la FFT validé, il est maintenant possible de passer à la suite du développement de ce projet. Puisque *FFTW3* ne travaille qu'en *floating point*, la prochaine étape est d'intégrer une autre librairie capable d'opérer une FFT en *fixed point*.

8 Intégration de la librairie *Ne10*

8.1 Choix de la librairie

La librairie *FFTW3* utilisée pour l'itération précédente de la MDCT ne propose pas de FFT en algorithmique entière. Le passage à une autre librairie était donc nécessaire et le choix s'est porté sur la librairie *Ne10* qui propose différentes FFT en *fixed point*.

Le projet *Ne10* propose toute une série de fonctions mathématiques et physiques de base ainsi que des fonctions de traitement de signal et de traitement d'image. La librairie est spécifiquement développée pour les architectures ARM possédant les instructions ARM NEON (ARMv7 et ARMv8-A)[15].

Ne10 propose à la fois des fonctions développées en *plain C* et des fonctions optimisées avec les instructions ARM NEON : les deux types de fonctions seront utilisés pour développer une MDCT optimisée et pour conserver une MDCT de référence en *plain C*. Maintenir une MDCT *plain C* permettra d'avoir une référence pour la mesure des performances de l'algorithme *fixed point* mais pourrait aussi s'avérer utile pour une utilisation de l'encodeur AAC sur un processeur ARM ne disposant pas des instructions NEON.

L'utilisation de la librairie *Ne10* est soumise à la licence *3-Clause BSD*, license permissive qui permet un usage commercial des produits intégrant la librairie et qui ne contraint pas à en distribuer le code source[16].

8.2 Implémentation de la MDCT basée sur la FFT *Ne10* en *float 32*

La librairie *Ne10* s'installe simplement en suivant les instructions données par la documentation : clone du projet GitHub, run du CMake et build du projet[15]. La librairie ne peut cependant être installée que sur une plateforme Linux, Android ou iOS reposant sur un processeur ARM. À partir de ce moment, il n'est donc plus possible de maintenir une implémentation de référence de la MDCT pour une architecture Intel.

Ne10 propose des algorithmes de FFT *real to complex* et *complex to complex* en *floating point* (32 bits) ou en *integer* (32 bits et 16 bits). L'objectif est évidemment de passer toute la MDCT en *integer* mais pour un premier test de la librairie, l'algorithme de la section précédente a tout d'abord été repris en remplaçant la FFT de *FFTW3* par la fonction `ne10_fft_c2c_1d_float32_c` de *Ne10* : FFT en *complex to complex* en *float 32*.

L'**annexe H** présente le code de cette implémentation par la classe `ne10_mdct_f32_c`. L'**annexe H.1** montre que ce code est construit sur le même modèle que le code utilisant la librairie *FFTW3*. La classe contient :

- la configuration de la FFT ;
- les tableaux de données d'entrée et de sortie de la FFT ;
- le tableau de facteurs de *twiddling*.

La représentation des données d'entrée et de sortie de la FTT n'est plus un tableau alternant les parties réelles et imaginaires des nombres complexes mais un tableau de nombre complexes. Ces nombres sont définis par une structure propre à la librairie *Ne10* contenant deux nombres entiers : le premier pour représenter la partie réelle du complexe et le second sa partie imaginaire.

L'**annexe H.2** montre l'initialisation de la MDCT dans le constructeur de la classe. Le constructeur initialise les tableaux de facteurs de twiddling de le même manière que l'algorithme basé sur la FFT de *FFTW3*. La configuration de la FFT de *Ne10* se fait conformément au code d'exemple donné par la documentation de la librairie[15].

Le destructeur présenté à l'**annexe H.3** permet de libérer la mémoire allouée pour la FFT en appelant la fonction adéquate de la librairie *Ne10*.

La fonction MDCT présentée dans l'**annexe H.4**, comme pour l'itération précédente, permet de transformer un signal temporel en *float* en un spectre de fréquences en *float* (figure 14).

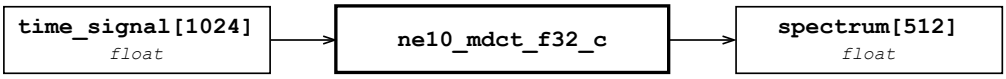


FIGURE 14 – Fonction `ne10_mdct_f32`

La MDCT de la classe `ne10_mdct_f32` :

- effectue les opérations de *pre-processing* ou *pre-twiddling* : ce sont les mêmes que celle de la MDCT *FFTW3* en arithmétique *floating point* sur 32 bits ;
- appelle l'algorithme de FFT : la FFT de *Ne10* prend en paramètres les tableaux contenant les données d'entrée et de sortie, la configuration de la FFT et un *integer* à `0` pour réaliser la FFT ou à `1` pour réaliser l'opération inverse ;
- effectue les opérations de *post-processing* ou *post-twiddling* : ici aussi les mêmes que celles de la MDCT *FFTW3* en arithmétique *floating point* sur 32 bits.

L'algorithme développé ici ne diffère donc pas de l'algorithme présenté à la section précédente. Son développement est trivial mais il permet de tester et de valider le fonctionnement de la librairie *Ne10*.

8.3 Validation de la MDCT *Ne10 float 32*

L'utilisation de la librairie *Ne10* est validée par comparaison du spectre de fréquences qu'elle génère avec le spectre généré par la MDCT de référence en *double*. Parmi les nombreux signaux d'entrée testés, les résultats sont présentés dans cette section avec un signal d'entrée à 440 Hz afin de faciliter la comparaison avec les autres spectres de fréquences présentés dans ce travail.

La représentation graphique de la comparaison des deux spectres à la figure 15 montre que le spectre de fréquences produit par la MDCT basée sur la FFT de *Ne10* en *float 32* se superpose parfaitement au spectre de fréquences généré par la MDCT de référence. La principale composante fréquentielle relevée se situe bien aux alentours de 440 Hz.

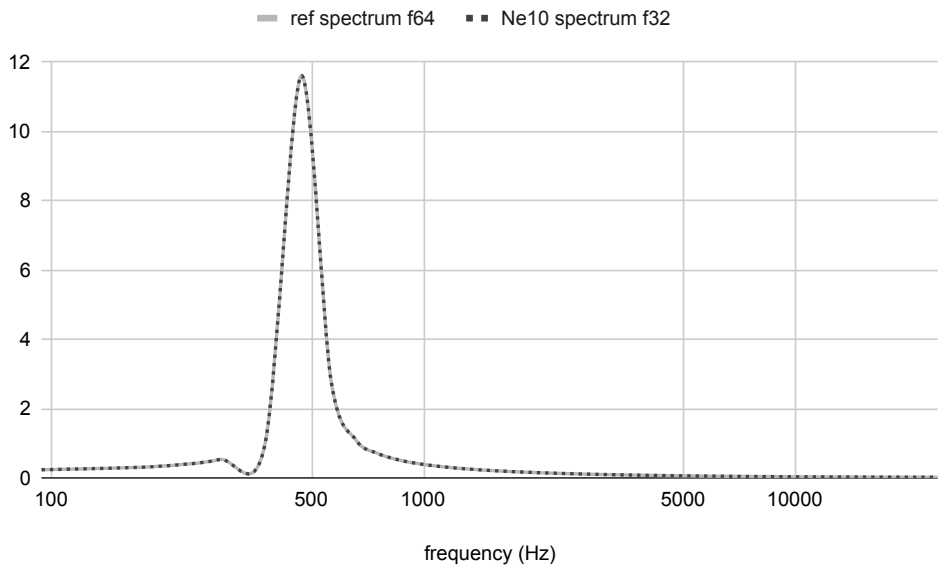


FIGURE 15 – Spectre de fréquences généré par la MDCT Ne10 float 32 comparé au spectre de référence (440 Hz)

8.4 Performances des FFT

Une fois l'utilisation de la librairie *Ne10* validée, des mesures de performances ont été réalisées sur les différentes FFT proposées par *Ne10*. Les FFT pour lesquelles les mesures de performances ont été effectuées sont celles qu'il était alors envisagé d'utiliser pour les futures itérations de la MDCT. Leurs temps d'exécution moyen ont été mesurés avec une comparaison entre les implémentations en *plain C* ou avec optimisations NEON.

La mesure de ces performances a été effectuée par le code présenté dans l'**annexe I** : le même code (**annexe I.1**) permet de générer les différents exécutables de mesure des performances des FFT en fonction de la variable de préprocesseur définie à la compilation (**annexe I.2**).

À titre comparatif, les performances de la FFT de *FFTW3* en *float 32* ont également été mesurées. L'**annexe J** présente le code utilisé pour la mesure des performances de cette FFT (**annexe J.1**) et les commandes utilisées pour le compiler (**annexe J.2**).

Les tests de performances ont été effectués sur 10 000 000 d'exécutions pour chaque FFT. Les données d'entrées des FFT sont générées aléatoirement et sont différentes pour chaque exécution. Les tests de performances ont été réalisés via un script permettant de lancer tous les exécutables de test les uns à la suite des autres. Ce script a été lancé sur le Raspberry via une connexion SSH. Aucune autre action n'a été réalisée sur le Raspberry durant l'exécution des tests.

Les tests de performances ont également été réalisés avec d'autres signaux en entrée, e.g. un signal sinusoïdal simple répété à chaque exécution. Les résultats de ces tests sont équivalents aux résultats des tests avec données aléatoires en entrée.

La table 1 présente les résultats de ces tests de performances.

FFT	Temps d'exécution moyen (ns)	Écart type (ns)
Ne10 f32 plain C	5538.36	0.61161×10^{-6}
Ne10 f32 NEON	3013.71	0.11833×10^{-6}
Ne10 i32 plain C	10678.9	2.12600×10^{-6}
Ne10 i32 NEON	3396.91	0.84808×10^{-6}
Ne10 i16 plain C	9569.07	0.01582×10^{-6}
Ne10 i16 NEON	2002.41	0.08582×10^{-6}
FFTW3 f32	4661.46	0.13637×10^{-6}

Table 1 – Tests de performances des algorithmes FFT

Conformément à ce qui pouvait être attendu, les FFT optimisées avec les instructions ARM NEON sont, en moyenne, plus rapides que leur équivalent en *plain C*.

La FFT en *integer* sur 16 bits avec optimisations NEON est la plus rapide de toutes : ce résultat n'est pas étonnant puisque les instructions SIMD permettent de réaliser plus d'opérations en parallèle avec des données codées sur moins de bits (cf. section 10 consacrée aux instructions ARM NEON). Elle aurait été la FFT la plus intéressante à utiliser mais la section 9 montrera pourquoi cela n'a pas été possible.

Contrairement aux attentes initiales, les FFT en *integer* sur 32 bits sont, en moyenne, plus lentes que les FFT en *float*. En *plain C*, la FFT en *integer* est deux fois plus lente avec un temps d'exécution moyen de 10 µs contre 5 µs pour la FFT en *float*. Avec les optimisations NEON, la FFT en *integer* est légèrement plus lente avec un temps d'exécution moyen de 3.4 µs contre 3 µs pour la FFT en *float*.

Ces résultats s'expliquent par la modernité du processeur ARM qui est capable de réaliser les opérations en *floating point* de manière très performante[17]. Sur 32 bits, une instruction en *integer* ne sera pas significativement plus rapide qu'une instruction en *float* comme cela pouvait être le cas pour des processeurs plus anciens.

En plus de ne pas permettre de gagner en performances, l'arithmétique *fixed point* va rendre l'algorithme plus lent qu'en *floating point*. En effet, l'arithmétique *fixed point* nécessite généralement plus d'opérations que l'arithmétique *floating point* pour arriver au même résultat, e.g. une multiplication en *floating point* devra être remplacée par une multiplication et une rotation de bits en *fixed point* (cf. section 9). La différence est moins marquée sur les FFT avec optimisations NEON car les instructions SIMD permettent parfois d'effectuer plusieurs de ces opérations en une seule.

Enfin, en *float* sur 32 bits, la FFT de *FFTW3* est plus performante que celle de *Ne10* avec un temps d'exécution moyen de 4.7 µs contre 5.5 µs pour la FFT de *Ne10* en *float 32 plain C*. La FFT de *Ne10* est cependant plus performante avec les optimisations NEON, que ce soit en *float* sur 32 bits (3 µs) ou en *integer* sur 32 bits (3.4 µs).

9 Algorithme MDCT en arithmétique *fixed point*

Le passage d'une arithmétique flottante à une arithmétique entière est une des optimisations envisagées par l'analyse préalable à ce travail. Le bénéfice attendu est double : l'arithmétique entière est généralement plus rapide et un bloc MDCT en arithmétique entière serait mieux intégré à l'ensemble de l'encodeur.

La première de ces attentes n'a pas pu être rencontrée. En effet, comme les résultats des tests de performances sur les FFT l'ont mis en évidence, le processeur ARMv7 du Raspberry Pi 4 supporte de nombreuses instructions en *floating point* sur 32 bits de manière très performante[17]. Or, plusieurs instructions *fixed point* sont souvent nécessaires pour remplacer une seule instruction en *floating point*. Plutôt que de gagner en temps d'exécution, le passage en *fixed point* a ralenti la MDCT.

Cependant, le passage en *fixed point* permet d'économiser un transtypage du *integer* vers le *float* en entrée de la MDCT et inversement en sortie. Avec un temps d'exécution au moins équivalent en *fixed point* qu'en *floating point*, le passage en *fixed point* permettrait donc tout de même d'améliorer les performances de l'ensemble de l'encodeur AAC.

Enfin, les données reçues à l'entrée de la MDCT sont codées en *integer* sur 16 bits alors que les *float* sont codés au minimum sur 32 bits. Garder le maximum de données et d'opérations en 16 bits permettrait de gagner en performance au moment de l'utilisation des instructions SIMD.

9.1 Arithmétique *fixed point*

La représentation *fixed point* est une alternative au *floating point* pour le codage des nombres décimaux[18]. Le principe est de réserver un certain nombre de bits pour coder la partie entière et un autre nombre de bits pour coder la partie décimale du nombre. Ce travail utilise deux notations pour la représentation en *fixed point*, toujours signée :

- *Qm* où *m* est le nombre de bits réservés aux décimales, e.g. une notation Q15 sur 16 bits permet de représenter un nombre signé ne contenant que des décimales et pas de partie entière puisqu'un bit est réservé pour le signe ;
- *Qx.y* où x est le nombre de bits réservés à la partie entière (avec signe) et y le nombre de bits réservés à la partie décimale, e.g. une notation Q1.15 équivaut à une notation Q15 sur 16 bits.

Pour passer la MDCT en algorithmique *fixed point*, il faut tout d'abord prêter attention à choisir la représentation adéquate. Par exemple, les données d'entrée de la MDCT sont comprises entre -0.9 et 0.9. Elles peuvent donc être représentées en Q15. Si elles avaient été comprises ente -1 et 1, la conversion en représentation Q15 aurait produit un dépassement sur l'une des valeurs limites et par conséquent une perte d'information.

Des dépassements peuvent également se produire lors des opérations arithmétiques :

- une **addition** ou une **soustraction** peut causer un dépassement d'un bit, e.g. la somme d'un nombre sur 32 bits et d'un nombre sur 16 bits nécessite potentiellement 33 bits pour être codée ;
- le résultat d'une **multiplication** ou d'une **division** peut devoir être codé sur un nombre de bits équivalent à la somme des nombres de bits composants les deux nombres multipliés ou divisés, e.g. le produit d'un nombre de 16 bits multiplié par un nombre de 32 bits nécessite potentiellement 48 bits pour être codé.

Ces dépassements sont théoriques. En fonction des données réelles à traiter, il est possible de ne pas respecter à la lettre les règles énoncées plus haut. C'est le cas par exemple avec les facteurs de *twiddling* dont on sait qu'ils valent au maximum un quart de la valeur d'un sinus ou d'un cosinus et qui sont codés en Q15 : il est alors certain que l'addition de deux de ces nombres ne causera pas de dépassement.

Là où l'implémentation en *floating point* était triviale, puisqu'elle demandait simplement de reprendre le code d'exemple fourni et de l'adapter à l'utilisation de la librairie *Ne10*, l'implémentation en *fixed point* devient plus complexe : il faut prêter attention à coder les nombres dans les bonnes *ranges* et à implémenter les différentes opérations arithmétiques de sorte à ne pas causer de dépassements.

9.2 Implémentation de la MDCT basée sur la FFT Ne10 en arithmétique *fixed point*

L'**annexe K** présente l'implémentation de la MDCT *Ne10 fixed point* en *plain C*. La classe `mdct_ne10_i32_c` a la même structure que l'implémentation en *floating point*. Le header présenté dans l'**annexe K.1** montre que seul le type des données a changé :

- La configuration et les tableaux d'entrée et de sortie de la FFT sont définis avec les types `int32`, et non plus `float32`, de la librairie *Ne10* ;
- Le tableau de facteurs de *twiddling* passe du `float` au `int16_t` ;
- La fonction MDCT prend en paramètres un signal temporel en `int16_t` et renvoie un spectre en `int32_t` au lieu des tableaux de `float` (figure 16).

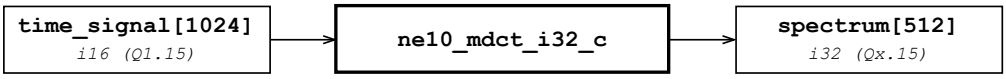


Figure 16 – Fonction `ne10_mdct_i32_c`

9.2.1 Utilisation de la FFT *Ne10* en *int 32*

L'utilisation de la librairie *Ne10* en *integer* est très peu différente de son utilisation en *float* :

- La configuration est initialisée dans le constructeur de la classe `mdct_ne10_i32_c` (**annexe K.2**) en *complex to complex* en *integer 32* avec en paramètre la taille de la fenêtre de la FFT.
- L'espace alloué à la configuration est libéré dans le destructeur de la classe `mdct_ne10_i32_c` (**annexe K.3**) avec la fonction appropriée de la librairie *Ne10* ;
- La fonction de FFT est appelée avec en paramètres le tableau contenant les données d'entrée, le tableau dans lequel sera calculé le résultat de la FFT, la configuration préalablement initialisée, un *integer* qui indique à la fonction de réaliser la FFT et non son opération inverse et, en plus de la FFT en *float*, un facteur de mise à échelle mis ici à 0.

Il est à noter que l'initialisation du facteur de mise à échelle de la fonction de FFT n'est pas documentée dans la librairie *Ne10*. L'effet de ce facteur sur la FFT n'est pas non plus indiqué. Ce facteur a été initialisé à 0 après avoir testé la FFT de *Ne10* dans le but d'obtenir les mêmes valeurs qu'en *float* mises à une échelle Q15.

La documentation incomplète de la librairie a été une des difficultés de ce travail. En dehors des quelques codes d'exemple disponibles, il est très compliqué de savoir quelles données sont attendues et sont produites par les fonctions de *Ne10*. Il est également très difficile de trouver des ressources externes sur l'utilisation de *Ne10*.

Le manque de documentation est également ce qui a empêché l'utilisation de la FFT en *int16* plutôt qu'en *int32*. Des opérations en 16 bits pour tout le bloc MDCT auraient été plus performantes. Malheureusement, si la documentation de *Ne10* dit bien travailler en Q15 pour du 16 bits et en Q31 pour du 32 bits, elle ne dit pas quelle *headroom* prévoir, quelles *ranges* de valeurs sont acceptées, si les opérations saturent ou non, etc. En testant la FFT 16 bits avec des données codées en Q15, trop de valeurs fausses étaient produites par la FFT. L'utilisation du facteur de mise à échelle n'a pas non plus permis d'obtenir de résultats satisfaisants.

Pour ne pas produire de dépassement dans la fonction de FFT, il a fallu prévoir 8 bits de *headroom* pour la FFT. Pour des données codées sur 16 bits, cela signifie qu'il ne resterait plus que 8 bits de données utiles pour le signal, dont 1 bit pour le signe. Cette perte de précision trop importante a forcé l'utilisation de la FFT en 32 bits.

9.2.2 Initialisation des *twiddle factors*

Le tableau de facteurs de `twiddle` est initialisé dans le constructeur de la classe `mdct_ne10_i32_c` (**annexe K.2**). Ses valeurs sont calculées en *double* puis converties en *integer* (représentation Q15).

Il aurait été posible de convertir les opérations d'initialisation en *fixed point* mais l'optimisation du constructeur n'est pas nécessaire. En effet, puisque la taille de fenêtre d'entrée ne varie pas, l'appel de l'initialisation ne sera fait qu'une fois et la MDCT est ensuite appelée en boucle avec les mêmes facteurs de *twiddling*.

9.2.3 Opérations de *pre-twiddling* et de *post-twiddling*

L'essentiel du travail pour cette implémentation a été le passage des opérations de *pre-* et de *post-processing* en arithmétique *fixed point*. Le résultat de ce travail se trouve dans l'**annexe K.3** qui présente la fonction MDCT basée sur la FFT de *Ne10* en *plain C*.

Les opérations de *pre-twiddling* transforment le signal temporel et les facteurs de *twiddling*, tous deux en représentation Q15, en un nombre en représentation Q1.23 sur 32 bits à donner en entrée de la FFT de *Ne10* en *int32*. La représentation Q1.23 est celle qui permet de garder le maximum de précision tout en s'assurant de ne pas produire de dépassement dans la FFT en gardant 8 bits de *headroom*.

Le *post-twiddling* récupère les données en Q9.23 produites par la FFT de *Ne10* : la précision de 23 décimales donnée en entrée, le bit de signe et les 8 bits de *headroom*. En combinaison avec les facteurs de *twiddling* codés en Q15, elles sont transformées en un spectre de fréquences codé en Q9.15 sur 32 bits.

Les opérations de *pre-processing* suivent toujours le même schéma : les parties réelles ou imaginaires des données d'entrée de la FFT sont la somme de deux produits d'un facteur de *twiddling* et de la somme de deux échantillons du signal temporel. Voici en exemple la transformation de la première opération de l'arithmétique *floating point* vers l'arithmétique *fixed point* :

$$fft\_in[i/2].r = r0 * c + i0 * s;$$

devient

$$fft\_in[i/2].r = (((r0 * c) + 64) >> 7) + (((i0 * s) + 64) >> 7);$$

où $r0 = time\_signal[MDCT\_M32 - 1 - i] + time\_signal[MDCT\_M32 + i];$

et $i0 = time\_signal[MDCT\_M2 + i] - time\_signal[MDCT\_M2 - 1 - i];$

r0 est la somme de deux échantillons du signal temporel codés en Q1.15 : il correspond donc à une notation **Q2.15** ici codée sur 32 bits plutôt que de perdre un bit de précision pour garder la valeur sur 16 bits ;

c *twiddle factor*, est codé en Q1.15, mais il est connu que le facteur de mise à l'échelle pour une fenêtre de 1024 échantillons temporels l'a réduit à $\frac{1}{4}$ de la range possible du Q1.15, noté **Q1.15/4** ;

r0*c le produit d'un nombre en Q2.15 avec un nombre Q1.15 est théoriquement un Q2.30 mais tient en pratique sur du **Q1.30/2** puisque le *twiddle factor* est en Q1.15/4 ;

((r0*c)+64)»7 le résultat en Q1.30/2 est ramené à du **Q1.23/2** par une opération de shift avec arrondi ($64 = 2^6$, ajouter un bit en $7^{\text{ème}}$ position en partant du LSB permet d'arrondir la valeur du $8^{\text{ème}}$ bit) ;

((i0*s)+64)»7 l'opération est équivalente à la précédente et est donc également codée en **Q1.23/2** ;

(((r0*c)+64)»7) + (((i0*s)+64)»7) l'addition de deux Q1.23/2 donne un **Q1.23**, codé sur 32 bits, il permet de conserver les 8 bits de *headroom* nécessaire à la FFT.

Les opérations de *post-processing* suivent elles aussi toujours le même schéma : les valeurs du spectre de fréquences sont calculées en additionnant deux produits, chacun étant le résultat d'une multiplication entre un facteur de *twiddling* et une donnée de sortie de la FFT. Voici en exemple la conversion de la première opération de *post-twiddling* en *fixed point* :

$$spectrum[i] = -r0 * c - i0 * s;$$

devient

$$spectrum[i] = ((((-r0 + 128) >> 8) * c + 16384) >> 15) - ((((i0 + 128) >> 8) * s + 16384) >> 15);$$

où $r0$ et $i0$ sont les parties réelle et imaginaire des données de sortie de la FFT

r0 donnée de sortie de la FFT, est codé en **Q9.23** ;

(-r0+128)»8 $r0$ est ramené en **Q9.15** par un *shift* de 8 avec arrondi ;

c *twiddle factor*, est codé en Q1.15, mais il est connu que le facteur de mise à l'échelle pour une fenêtre de 1024 échantillons temporels l'a réduit à $\frac{1}{4}$ de la range possible du Q1.15, noté **Q1.15/4** ;

((-r0+128)»8)*c la multiplication d'un Q9.15 par un Q1.15/4 donne un Q10.30/4 ou **Q9.30/2** ;

(((-r0+128)»8)*c+16384)»15 le résultat en Q9.30/2 est ramené à du **Q9.15/2** par un *shift* de 15 avec arrondi ;

(((i0+128)»8)*s+16384)»15 le second terme effectue les mêmes opérations pour un résultat en **Q9.15/2** ;

((((-r0+128)»8)*c+16384)»15) - ((((i0+128)»8)*s+16384)»15) la différence de deux Q9.15/2 est un Q10.15/2 ou **Q9.15**.

9.3 Validation de la MDCT *Ne10 fixed point plain C*

L'algorithme est validé par comparaison avec l'algorithme de référence. Le protocole de validation est détaillé dans la section 11.1 : les spectres sont générés par les différentes MDCT sur base du même signal temporel pour que tous les spectres puissent être comparés entre eux. Afin de s'assurer de la cohérence des données produites par la MDCT, le signal d'entrée est un signal sinusoïdal : le spectre de fréquences ne doit donc mettre en évidence qu'une seule composante fréquentielle.

La figure 17 montre la comparaison entre le spectre de fréquences généré par la MDCT basée sur la FFT Ne10 en plain C et celui de référence en *integer* sur 32 bits (calculs en *double* mis à échelle Q15) pour un signal d'entrée à 440 Hz.

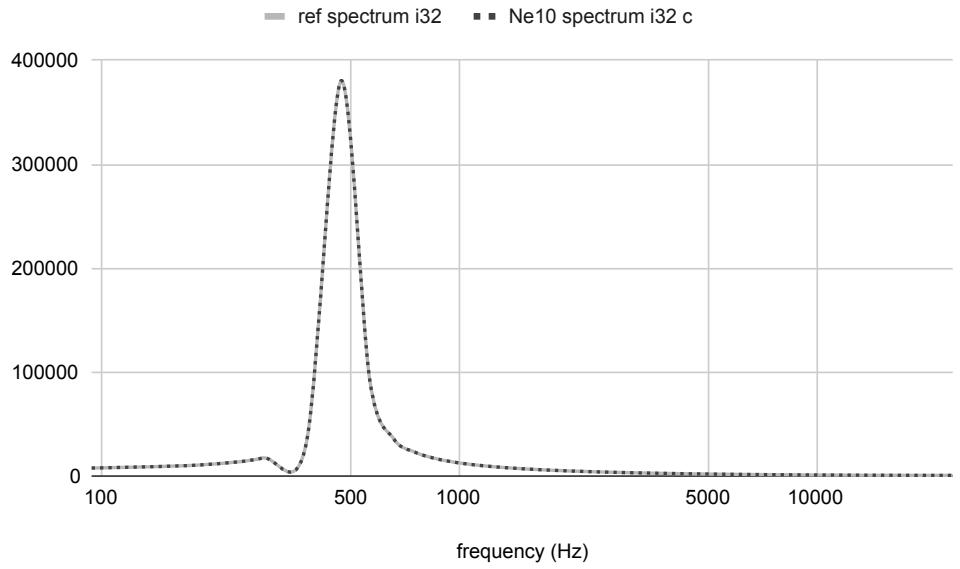


Figure 17 – Spectre de fréquences généré par la MDCT Ne10 i32 plain C comparé au spectre de référence (440 Hz)

Le spectre de fréquences généré par la MDCT *Ne10 i32 plain C* (en pointillés) se superpose parfaitement au spectre de référence. La composante fréquentielle la plus haute se situe bien aux alentours de 440 Hz.

9.4 Performances de l'arithmétique *fixed point*

Les performances de la MDCT *fixed point* en *plain C* sont moins bonnes que celles de l'implémentation en *floating point* (cf. section 11.2 pour plus d'explications sur la mesure des performances) :

- La MDCT NEON *float 32 plain C* a un temps d'exécution moyen de 9 µs ;
- La MDCT NEON *integer 32 plain C* a un temps d'exécution moyen de 22 µs.

Le temps d'exécution notablement plus conséquent s'explique par le fait que le processeur ARMv7 supporte de nombreuses instructions *floating point* sur 32 bits. Contrairement à des processeurs plus anciens, l'ARMv7 n'est donc pas plus efficace en arithmétique *fixed point*.

De plus, la section 9.2 a montré qu'une seule instruction en *floating point* était remplacée par plusieurs opérations *fixed point*. Une multiplication est, par exemple, remplacée par une multiplication, une rotation et une addition pour l'arrondi. Puisque les opérations sur des *integer* ne sont pas plus rapides que sur des *float*, il n'est donc pas étonnant que la MDCT *fixed point* soit plus lente que la MDCT *floating point*.

L'algorithme *fixed point* utilise également des opérations de *cast* d'*integer* en 32 bit vers du 64 bit. Ces opérations ont pour but d'éviter un dépassement qui se produit dans les opérations de *post-twiddling* et fausse les résultats obtenus. Sans ces opérations de *cast*, l'algorithme *fixed point* a un temps moyen d'exécution de 15 µs.

Ces explications sont cohérentes avec les mesures des performances des différentes FFT (section 8.4) : la FFT de *Ne10* en *int 32 plain C* a un temps d'exécution moyen de 11 µs contre 6 µs pour la FFT NEON *float 32 plain C* et 5 µs pour la FFT *FFTW3 float 32*. À la perte de 5 µs dans l'exécution de la FFT s'ajoute donc 1 µs dû aux opérations de *pre-* et *post-processing* supplémentaires et 7 µs pour les opérations de *cast*.

10 Optimisations à l'architecture ARM

La dernière optimisation de la MDCT implémentée dans ce travail consiste à optimiser l'algorithme au processeur ARM par l'utilisation des instructions ARM NEON. Ce sont des instructions SIMD (*Single Instruction on Multiple Data*), ou vectorisées, i.e. elles permettent de réaliser certaines opérations arithmétiques en une seule instruction effectuée en parallèle sur plusieurs données, contrairement aux opérations scalaires des itérations précédentes de la MDCT qui effectuent les opérations les unes à la suite des autres. Ce sont les fonctions intrinsèques (*intrinsic*) spécifiques au processeur ARMv7 qui est celui du Raspberry Pi 4.

Les sections précédentes ont montré que les instructions sur 32 bits de l'ARMv7 sont aussi performantes en *float* qu'en *integer* et que l'arithmétique *fixed point* est toujours un peu plus lente que l'arithmétique *floating point*. Il n'est donc pas raisonnable d'attendre qu'une MDCT en *fixed point* soit plus performante que son équivalent en *floating point*.

Il est cependant possible d'aller chercher des performances en codant les données sur moins de bits : si le *float* est codé au minimum sur 32 bits, ce n'est pas le cas de l'*integer* qui peut être codé sur 16 bits. L'enjeu est alors de maintenir les données sur le plus petit espace tout en prêtant attention aux dépassements et à la perte de précision.

La différence de temps d'exécution entre le *fixed point* et le *floating point* peut également être atténuée par l'utilisation de certaines instructions SIMD : une seule instruction vectorielle peut combiner plusieurs instructions scalaires, e.g. les multiplications peuvent se faire entre deux nombres codés sur 16 bits et coder le résultat sur 16 bits (en ne gardant que les bits de poids fort) et ainsi éviter une instruction de *shift*. Les instructions avec saturation permettent également d'éviter les dépassements sans mettre en place de mécanisme coûteux en temps d'exécution, comme c'était par exemple le cas pour l'algorithme *fixed point plain C* avec un *cast* vers un *integer* 64 bits.

Le tableau de facteurs de *twiddling*, et plus généralement les différents vecteurs sur lesquels seront appelées les instructions SIMD, peuvent être organisés de manière à limiter le nombre d'opérations nécessaires. Une multiplication entre deux vecteurs multipliera le premier nombre avec le premier, le deuxième avec le deuxième, etc. L'utilisation des instructions SIMD est d'autant plus performante si les données ont été initialisées, ou placées par les opérations précédentes, dans le bon ordre plutôt que de devoir réorganiser les données avant l'appel de chaque instruction.

Enfin, pour rappel, l'optimisation des algorithmes *fixed point* ne s'arrête pas à l'algorithme en tant que tel puisqu'une MDCT *fixed point* permet d'éviter le transtypage des données d'entrée (reçues en Q15) et des données de sortie (le bloc de quantification devant également être implémenté en *fixed point*).

10.1 Instructions ARM NEON

Le jeu d'instructions NEON, ou *Advanced SIMD* est le jeu d'instructions SIMD spécifique à l'architecture ARMv7. Les instruction SIMD permettent d'exécuter une même instruction (*Single Instruction*) sur des plusieurs données (*Multiple Data*), par opposition aux instruction SISD (*Single Instruction on Single Data*) ou MIMD (*Multiple Instruction on Multiple Data*). La vectorisation des données permet de réaliser simultanément une instruction sur celles-ci (figure 18)[19]. Des contraites fortes s'appliquent à l'utilisation des instructions SIMD en raison de leur mode de fonctionnement propre. Les algorithmes de traitement de signal, comme c'est le cas ici avec la MDCT, se prêtent bien à la vectorisation.

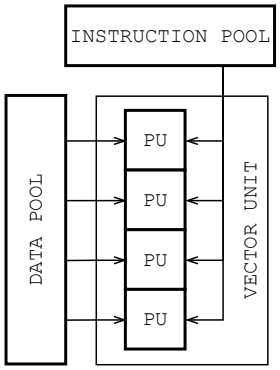


FIGURE 18 – Fonctionnement d'une instruction SIMD (*Single Instruction on Multiple Data*)

Les instructions SIMD sont appelées sur des registres spécifiques. La banque de registres NEON compte 32 registres de 64 bits. Ces registres sont partagés entre NEON et VFP (*Vector Floating Point*, jeu d'instructions SIMD pour *float* sur 32 bits). Les 32 registres peuvent aussi être vus comme 16 registres de 128 bits en associant les registres de 63 bits deux à deux[17]. La figure 19 montre la correspondance entre les registres NEON et VFP et montre comment les 32 registres de 64 bits (D0 à D31, D pour *doubleword*) peuvent être associés pour former 16 registres de 128 bits (Q0 à Q15, Q pour *quadword*).

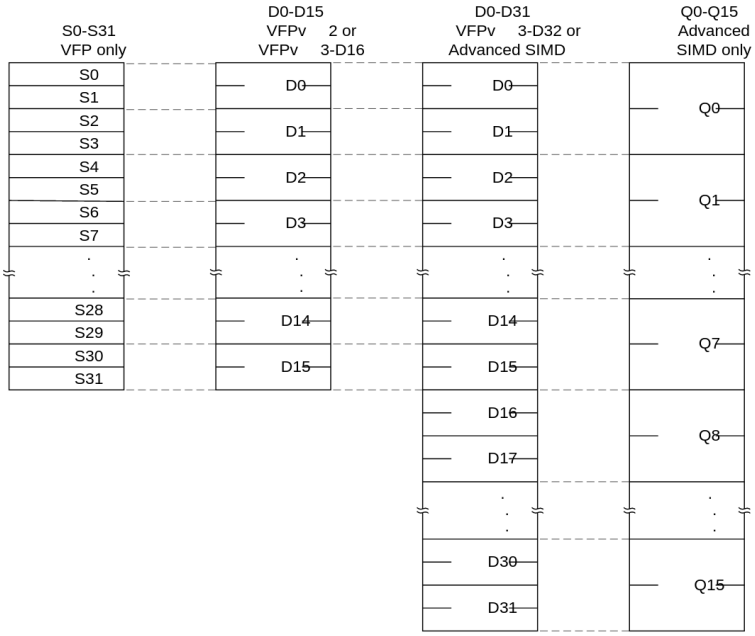


FIGURE 19 – Banque de registres partagée entre NEON et VFP

Les instructions NEON permettent de travailler avec des *integer* de 8, 16, 32 ou 64 bits, signés et non signés. Pour le développement de la MDCT, seules des représentations signées en 16 ou en 32 bits seront utilisées. Le 8 bits n'est pas assez précis pour de l'audio, il est généralement utilisé pour de la vidéo. Le 64 bits serait désavantageux par rapport à un algorithme en *float* sur 32 bits : son utilisation doit donc être évitée.

Les données doivent être chargées dans les registres NEON avec des instructions de *load* spécifiques. Ces instructions de chargement ne permettent pas de charger n'importe quelles données dans n'importe quel ordre : les données à charger doivent être consécutives et devront être rangées dans le bon ordre avant leur chargement. Après avoir exécuté les instructions SIMD souhaitées, les données sont déchargées des registres avec des instructions de *store*.

Les instructions NEON permettent de réaliser une même instruction (*Single Instruction*) sur plusieurs données à la fois (*Multiple Data*) après que celles-ci aient été vectorisées. Pour les instructions sur deux vecteurs, les instructions sont appliquées sur les nombres dans l'ordre dans lequel ils ont été chargés dans les vecteurs. Pour une addition, le premier nombre du premier vecteur est additionné au premier nombre du second vecteur, le deuxième nombre du premier vecteur est additionné au deuxième nombre du second vecteur, etc. Par exemple, l'instruction VADD.I16, schématisée à la figure 20, additionne deux vecteurs de 128 bits contenant chacun 8 nombres entiers signés de 16 bits.

Les résultats des instructions SIMD sont placés dans des vecteurs dans le même ordre. Le résultat de l'instruction VADD.I16 est placé dans un troisième vecteur de 128 bits contenant également 8 nombres entiers signés de 16 bits. Les résultats des instructions SIMD peuvent également être placés dans des vecteurs contenant deux fois moins de données afin de conserver plus de précision : par exemple, le résultat de la multiplication de deux vecteurs contenant chacun quatre *integer* codés sur 16 bits peut être placée dans un vecteur contenant quatre *integer* en 32 bits.

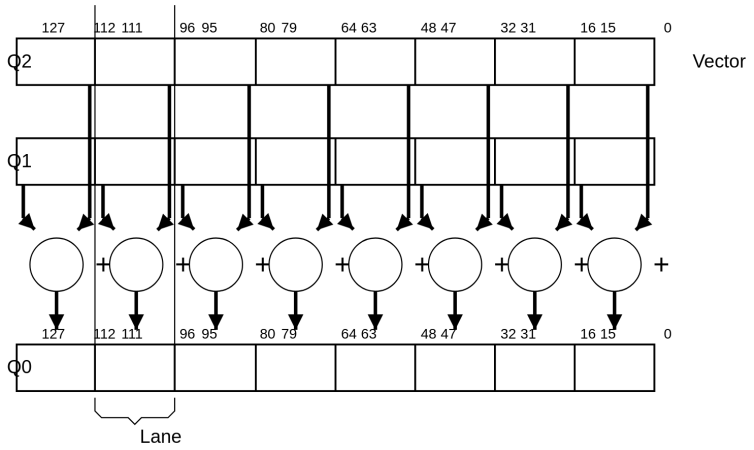


Figure 20 – Instruction NEON VADD.I16

GCC rend disponible l'accès aux *intrinsic* NEON via le *header* `<arm_neon.h>`[20]. L'utilisation de ces instructions est activée avec l'option `-mfpu=neon` à la compilation. Les noms donnés à aux fonctions de ce *header* permettent de facilement comprendre le rôle de ces fonctions. C'est la cas, par exemple, de la fonction `vmull_s16`, utlisée à de nombreuses reprises dans ce travail :

v le `v` en début d'instruction indique qu'il s'agit d'une fonction *Advanced SIMD* ;

mul indique que l'instruction appelée est un multiplication ;

l indique que le résultat de la multiplication sera entièrement conservé, et pas uniquement les bits de poids fort ;

s16 la multiplication est effectué sur un vecteur de nombres signés de 16 bits : le résultat sera donc un vecteur de nombres signés de 32 bits.

10.2 Implémentation de la MDCT basée sur la FFT de *Ne10* en arithmétique *fixed point* avec optimisations NEON

L'**annexe L** présente l'implémentation de la MDCT *Ne10 fixed point* avec optimisations NEON. Le *header* de la classe `mdct_ne10_i32_neon` (**annexe L.1**) contient les mêmes définitions de fonctions que l'implémentation *plain C* (figure 21). La fonction MDCT a la même signature avec optimisations NEON qu'en *plain C*. La configuration de la FFT et les tableaux d'entrée et de sortie de la FFT sont également les mêmes qu'en *plain C*. La différence se trouve au niveau des facteurs de *twiddling* qui sont désormais séparés en quatre tableaux.

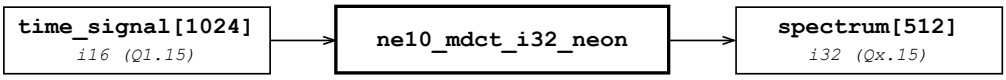


FIGURE 21 – Fonction `ne10_mdct_i32_neon`

10.2.1 Séparation du tableau de facteurs de *twiddling*

La figure 22 montre comment le tableau de 512 *twiddle factors* est séparé en deux tableaux de 256 éléments : un tableau (*twiddle start*) contenant les 256 premiers éléments dans le même ordre que le tableau originel et un tableau (*twiddle end*) contenant les 256 derniers éléments du tableau originel rangés du dernier au 256ème.

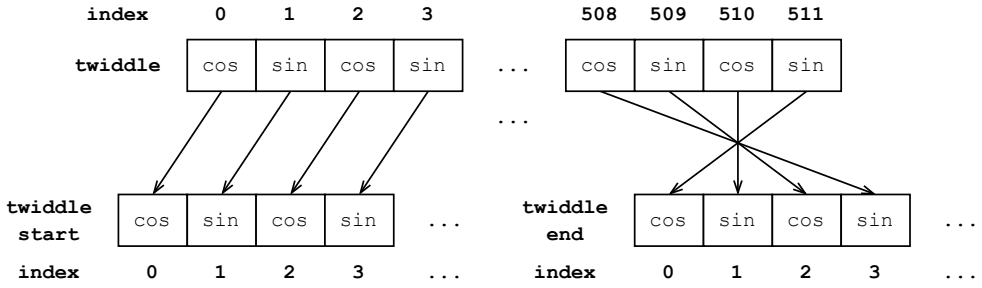


FIGURE 22 – Séparation du tableau de facteurs de *twiddling*

Comme pour l'itération précédente de la MDCT, ces facteurs sont calculés dans le constructeur (**annexe L.2**) en *double* avant d'être convertis en *integer*. Chacun des deux tableaux de *twiddling factors* est codé de deux manière différentes dans deux tableaux différents :

- les facteurs de *pre-twiddling* sont codés sur 16 bits en représentation Q15 ;
- les facteurs de *post-twiddling* sont codés sur 32 bits en représentation Q31.

Cette séparation vise à optimiser la MDCT en adaptant au préalable les facteurs de *twiddling* à la manière dont ils vont être utilisés dans les opérations de *pre-* et de *post-twiddling*. En effet, la MDCT sera toujours appelée de la même manière, avec la même configuration, puisque la taille de la fenêtre d'entrée ne varie pas. L'initialisation n'est appelée qu'une fois avant d'appeler en boucle la MDCT : les opérations supplémentaires qu'elle contient sont donc finalement beaucoup moins coûteuses à l'initialisation qu'à chaque appel de la MDCT.

10.2.2 Utilisation de la FFT optimisée NEON

Le passage de la FFT en *plain C* à la FFT optimisée NEON est assez simple puisque les deux fonctions ont la même signature et sont appelées de la même manière. Pour pouvoir remplacer l'appel de la fonction *plain C* par l'appel de la fonction optimisée, il suffit d'initialiser au préalable la configuration de la FFT avec la fonction appropriée, i.e. `ne10_fft_alloc_c2c_int32_neon` au lieu de `ne10_fft_alloc_c2c_int32_c` (**annexe L.2**). L'espace mémoire alloué à la configuration est libéré de la même manière qu'en *plain C* (**annexe L.3**).

10.2.3 Organisation des données du signal temporel

Comme le facteurs de *twiddling*, les échantillons du signal temporel doivent être organisés afin de pouvoir appeler efficacement les instructions SIMD dessus. Pour trouver l'organisation des données les plus efficaces, j'ai décomposé les opérations de *pre-twiddling* pour pouvoir y identifier les schémas répétitifs. J'ai ensuite validé l'organisation des opérations dans un tableur. Le détail de ces opérations est donné en commentaire dans l'**annexe L.4**.

Le tableau contenant les échantillons du signal temporel est décomposé en quatre partie : de `t1` à `t4`. Ces données sont chargées dans les vecteurs NEON avec l'opération `vld2_s16` : instruction *Advanced SIMD* (**v**) de chargement (**ld**) dans deux vecteurs (avec un désentrelacement de **2**) de données signées codées sur 16 bits (**s16**). Le désentrelacement charge une donnée sur deux dans un vecteur et les données restantes dans un second vecteur (figure 23)[21].

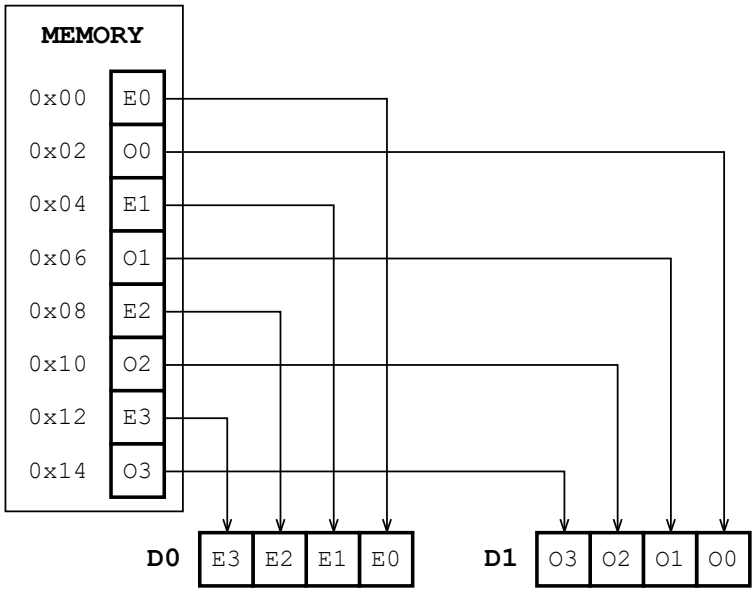


Figure 23 – Fonctionnement de l'instruction `VLD2.16{D0, D1}, [R0]`

Les données des tableaux `t2` et `t4` sont inversées pour être rangées de la dernière à la première. Cette opération se fait en appelant deux instructions SIMD de *reverse* :

- `vrev32_s16` qui inverse les données au sein des registres de poids fort et les données au sein des registres de poids faible ;
- `vrev64_s32` qui inverse les registres de poids fort avec les registres de poids faible.

La figure 24 montre l'effet de cette double instruction de *reverse* sur un vecteur de quatre données.

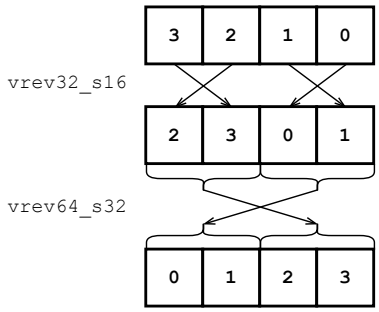


FIGURE 24 – Inversion des données dans un vecteur de quatre éléments

10.2.4 Pre-processing

Les opérations de *pre-twiddling* suivent la même logique que celles de l'itérations, à la différence qu'elles sont effecuées sur plusieurs données en même temps. La première de ces instructions en *plain C* :

$$fft\_in[i/2].r = (((r0 * c) + 64) >> 7) + (((i0 * s) + 64) >> 7);$$

devient

$$\begin{aligned} &start.val[0] = vshrq\_n\_s32(vaddq\_s32(\\ &\quad vaddq\_s32(vmull\_s16(t4.val[1], start\_tw.val[0]), vmull\_s16(t3.val[0], start\_tw.val[0])), \\ &\quad vaddq\_s32(vmull\_s16(t1.val[0], start\_tw.val[1]), vmull\_s16(vneg\_s16(t2.val[1]), start\_tw.val[1]))), 7); \end{aligned}$$

t1.val[x], t2.val[x], t3.val[x], t4.val[x] vecteurs contenant quatre échantillons du signal temporel ;

start_tw.val[x] vecteur contenant quatre facteurs de *twiddling* ;

vmull_s16(t4.val[1], start_tw.val[0]) instruction ARM SIMD (**v**) qui multiplie (**mul**) `t4.val[1]` par `start_tw.val[0]`, *integers* signés sur 16 bits (**s16**), et garde les 32 bits de résultat (**l**) ;

vaddq_s32(vmull_s16(...), vmull_s16(...)) addition (**add**) de deux vecteurs contenant quatre entiers signés de 32 bits (**s32**), ces vecteurs sont le résultat de deux opérations de multiplication ;

vaddq_s32(vaddq_s32(...), vaddq_s32(...)) la même instruction SIMD que celle appelée à l'étape précédente addition les résultats de deux additions et place le résulat dans un vecteur de quatre entiers signés codés sur 32 bits ;

vshrq_n_s32(vaddq_s32(...), 7) rotation de bits vers la droite (**sh**ift **r**ight) réalisée afin de conserver la *headroom* nécessaire pour la FFT.

Les résultats des opérations de *pre-twiddling* sont déchargés dans le tableau d'entrée de la FFT avec l'instruction `vst2q_s32`. C'est une instruction *Advanced SIMD* (**v**) de *store* (**st**) qui entrelace deux vecteurs (**2**) d'entiers signés de 32 bits (**s32**).

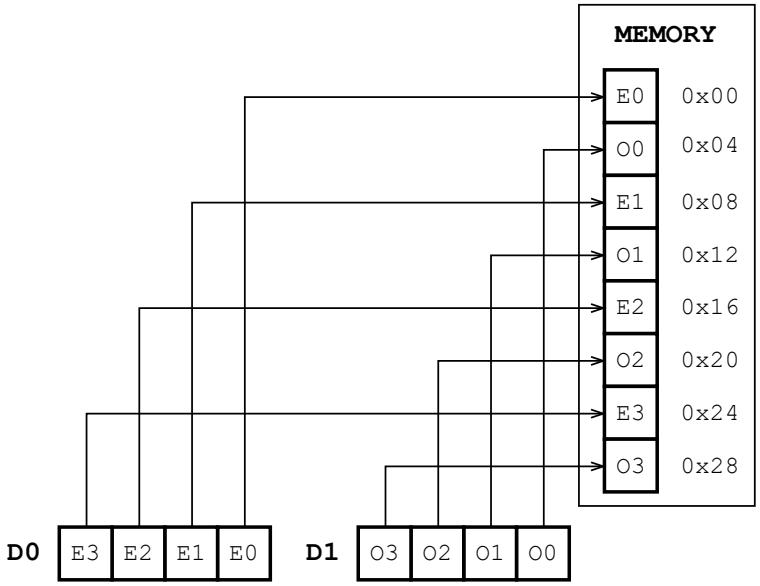


FIGURE 25 – Fonctionnement de l'instruction `VST2.32 {D0, D1}, [R0]`

10.2.5 Post-processing

Les données de sortie de la FFT sont chargées dans des vecteurs, certaines dans l'ordre dans lequel elles se trouvent en mémoire et certaines dans l'ordre inverse. Les instructions SIMD utilisées pour leur chargement sont un peu différentes de celles utilisées pour le *pre-twiddling* puisque les données de sortie de la FFT sont codées sur 16 bits mais le principe reste le même.

Les opérations de *post-processing* de l'itération précédent sont adaptées à l'utilisation d'instructions SIMD, Ainsi, la première des opérations de *post-processing* :

$$spectrum[i] = ((((-r0 + 128) >> 8) * c + 16384) >> 15) - ((((i0 + 128) >> 8) * s + 16384) >> 15);$$

devient

$$\begin{aligned} spectrum\_start.val[0] = vshrq\_n\_s32(&vaddq\_s32(\\ &vqrdmulhq\_s32(vnegq\_s32(fft\_out\_start.val[0]), start\_tw.val[0]), \\ &vqrdmulhq\_s32(vnegq\_s32(fft\_out\_start.val[1]), start\_tw.val[1])), 8); \end{aligned}$$

fft_out_start.val[x] vecteurs contenant quatre résultats de sortie de la FFT ;

start_tw.val[x] vecteur contenant quatre facteurs de *twiddling* ;

vnegq_s32(fft_out_start.val[x]) instruction *Advanced SIMD* (**v**) qui transforme en leur valeur négative (**neg**) quatre nombres entiers signés de 32 bits (**s32**) pour une vue en 128 bits du registre (**q***uadword*) ;

vqrdmulhq_s32(vnegq_s32(fft_out_start.val[x]), start_tw.val[x]) intruction ARM SIMD (**v**) qui opère sur une vue en 128 bits des registres (**q**) une multiplication (**mul**) sur des entiers signés en 32 bits (**s32** : le résultat de cette opération est un vecteur de quatre entiers signés sur 32 bits (**q**) qui ne conserve que les bits de poids forts (**h**) du produit de la multiplication ;

vaddq_s32(vaddq_s32(vqrdmulhq_s32(...), vqrdmulhq_s32(...)) les deux vecteurs de quatre éléments entiers sur 32 bits sont additionnés (**add**) avec pour résulat un autre vercteur de quatre entiers signés sur 32 bits pour une représentation en 128 bits des vecteurs ;

vshrq_n_s32(vaddq_s32(...), 8) le résultat final subit une rotation vers la droite (**sh**ift **r**ight) de 8 bits pour revenir à une échalle Q15.

Les résultats des opérations de *post-twiddling* sont déchargées des registres NEON par les mêmes opérations de *store* (`vst2q_s32`) que les résultats des opérations de *pre-twiddling*.

10.3 Validation

L'implémentation de la MDCT *fixed point* avec optimisations NEON a été testée de la même manière que toutes les autres MDCT (cf. section 11.1). Parmi les nombreux tests effectués, le figure 26 présente le spectre de fréquence que génère cette MDCT avec un signal à 440 Hz en entrée.

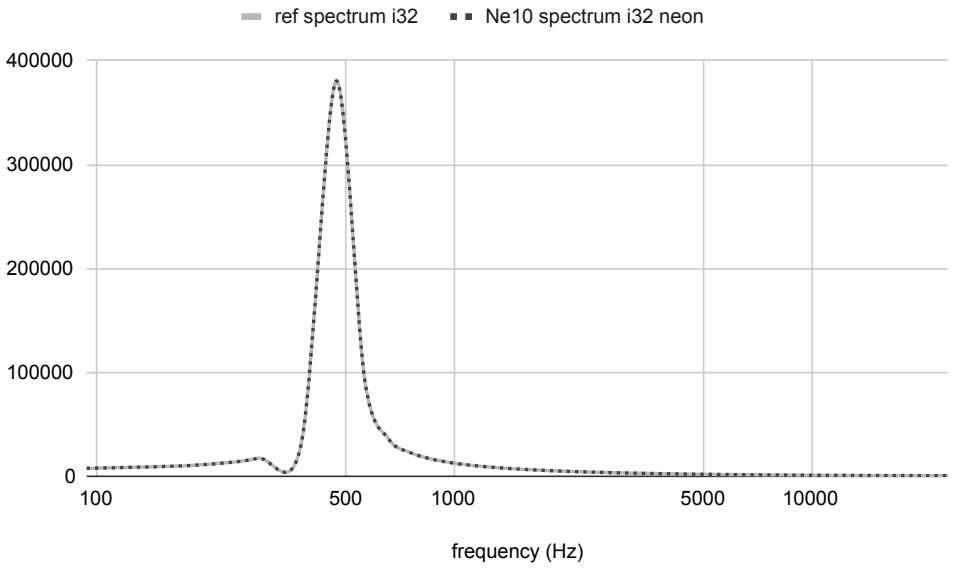


Figure 26 – Spectre de fréquences généré par la MDCT Ne10 i32 NEON comparé au spectre de référence (440 Hz)

Le spectre généré par la MDCT *Ne10 i32 neon* se superpose bien au spectre de référence. La principale composante fréquentielle relevée est, comme attendu, aux alentours de 440 Hz

10.4 Performances

Les performances de cette dernière itération de la MDCT sont cette fois satisfaisantes. Avec un temps d'exécutions moyen de 5.8 µs, elle est la plus rapide des MDCT développées dans ce travail. Elle est évidemment plus rapide que l'algorithme en *fixed point plain C* qui était le plus lent de tous les algorithmes optimisés avec 22 µs de temps d'exécution moyen. Mais elle est surtout plus rapide que les MDCT en *floating point* (9 µs pour la MDCT basée sur la FFT de *Ne10* et 7.4 µs pour celle basée sur la FFT de *FFTW3*).

La section suivante revient plus en détail sur le protocole de mesure des performances et les comparaisons entre les différentes MDCT.

11 Analyse des résultats

11.1 Protocole de validation

Les sections précédentes ont montré que les différentes itérations de la MDCT ont été validées à chaque étape du développement de ce travail. Cette validation consiste essentiellement en une vérification manuelle des données de sortie de la MDCT sur base de divers signaux en entrée.

L'**annexe M** présente le code de test utilisé pour ces vérifications. Les tests génèrent différents types de signaux : signaux sinusoïdaux à une ou plusieurs fréquences et données aléatoires. Les différentes MDCT génèrent des spectres de fréquences qui sont sortis dans un fichier CSV afin de pouvoir être exploités sous forme graphique.

Avec un signal sinusoïdal connu en entrée, il est facile de vérifier que le spectre ne contienne bien qu'une seule composante fréquentielle, que la vérification se fasse en lisant les données brutes à la sortie de l'algorithme ou par une analyse graphique de celles-ci. Ce travail a présenté les spectres de fréquences générés par les MDCT avec un signal à 440 Hz. Les MDCT ont bien entendu été testées avec d'autres signaux. La figure 27 présente, par exemple, le spectre de fréquences d'un signal sinusoïdal à 10 kHz.

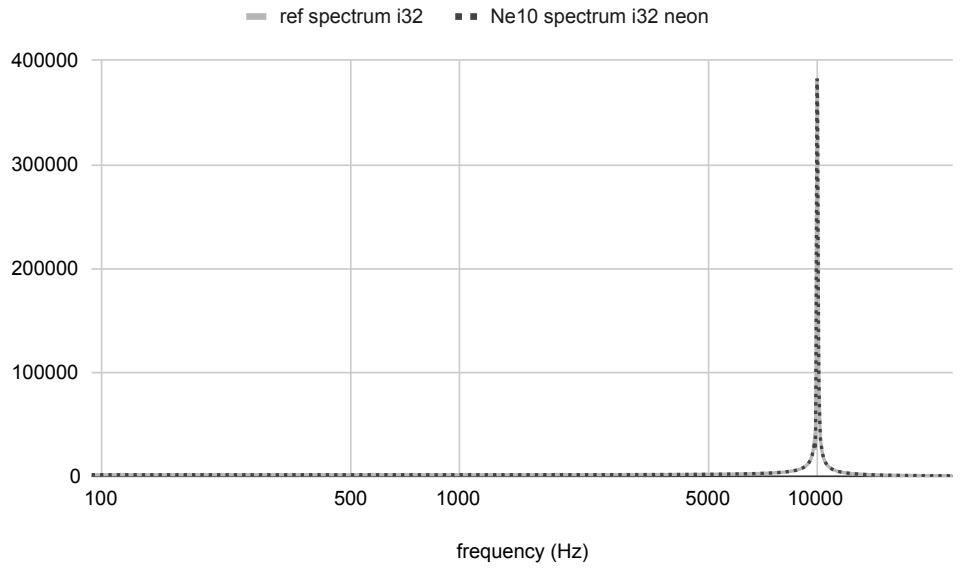


Figure 27 – Spectre de fréquences généré par la MDCT Ne10 i32 NEON comparé au spectre de référence (20 kHz)

La figure 27 montre que le spectre de fréquences généré par la MDCT optimisée se superpose bien au spectre de fréquences de référence. Toutes les MDCT optimisées ont été validées par comparaison du spectre de fréquences : comparaison visuelle, comme ici avec un graphique, ou calcul de la différence entre les données des deux spectres. Cette validation a pu mettre en évidence les dépassements produits et de les corriger.

Les spectres de fréquences ont également été comparés avec les spectres de fréquences générés par la MDCT basée sur la FFT de *FFTW3*, directement en *float* ou mis à échelle Q15. Ces comparaisons ne sont pas reproduites dans ce travail à cause du manque de lisibilité de graphiques superposant trois spectres de fréquences.

La composante fréquentielle principale mise en évidence par le spectre de fréquences de la figure 27 ne se situe pas exactement à 10 kHz. En effet, la résolution du spectre de fréquences n'offre pas une analyse aussi fine du signal. La résolution pour un spectre de fréquences de 256 composantes fréquentielles (512 éléments en nombres complexes), pour une fréquence d'échantillonnage de 48 kHz est de $\frac{Fs/2}{size} = \frac{24000}{256} = 93.75$ Hz.

Pour rappel, les données du spectre de fréquences produit par la MDCT sont des complexes contenus dans un tableau à une dimension alternant les parties réelles aux parties imaginaires des composantes fréquentielles. Le code de test de l'**annexe M** calcule donc le module de ces composantes fréquentielles avant de les écrire dans le fichier CSV utilisé pour générer les différents graphiques présentés dans ce travail.

En plus de signaux à une seule fréquence, les MDCT ont été testées avec des signaux à multiples fréquences. La figure 28 représente un de ces signaux. Il est composé de deux sinus d'amplitude différentes :

- un signal sinusoïdal d'amplitude 0.9 à 440 Hz ;
- un signal sinusoïdal d'amplitude 0.45 à 880 Hz.

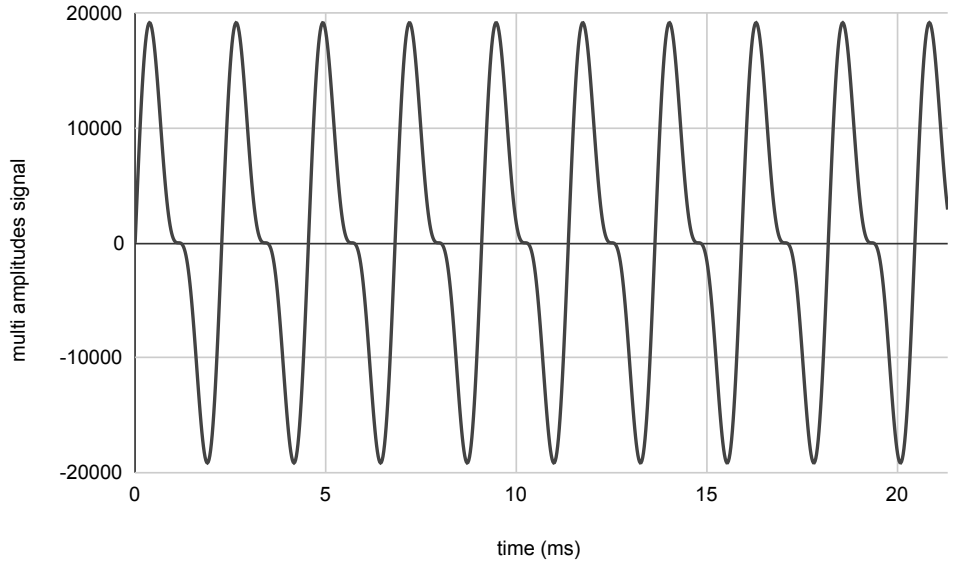


Figure 28 – Signal temporel composé de deux sinus d'amplitude différente à 440 et 880 Hz

La figure 29 montre le spectre de fréquences produit par la dernière itération de la MDCT avec ce signal en entrée. Les composantes fréquentielles mises en évidence par le graphique sont bien celles attendues : 440 et 880 Hz.

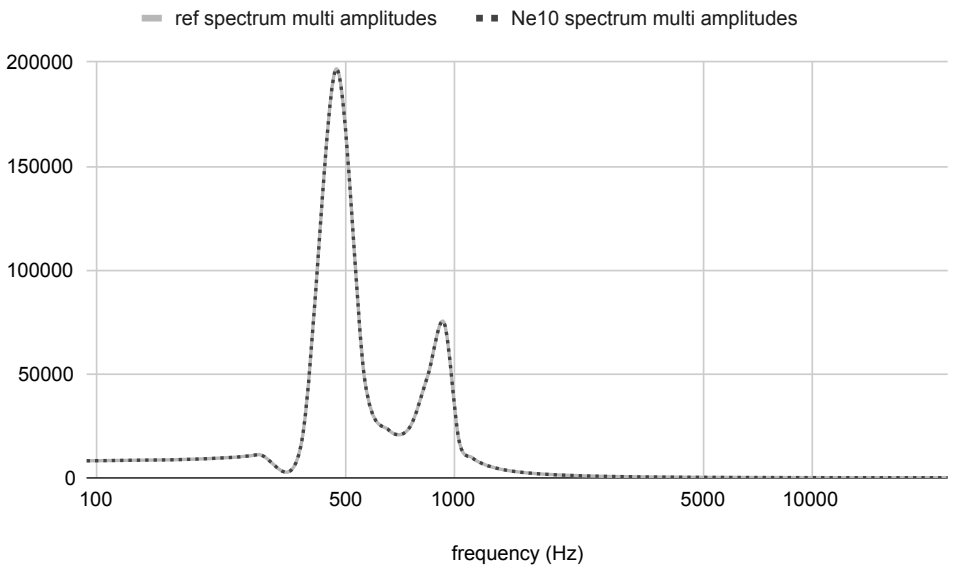


FIGURE 29 – Spectre de fréquences généré par la MDCT Ne10 i32 NEON pour un signal composé de deux sinus d'amplitude différente

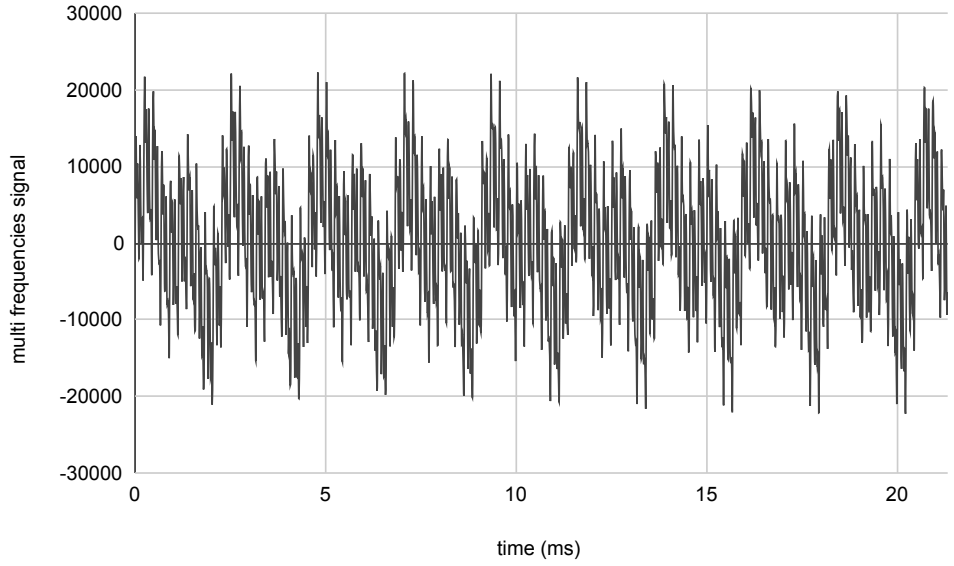


FIGURE 30 – Signal temporel composé de sinus à 440 Hz, 880 Hz, 4.4 kHz, 8.8 kHz et 17.6 kHz

Un signal composé de multiples fréquences de même amplitude (30) confirme la validité de la MDCT. La figure 31 met bien en évidence les composantes fréquentielles attendues à 440 Hz, 880 Hz, 4.4 kHz, 8.8 kHz et 17.6 kHz.

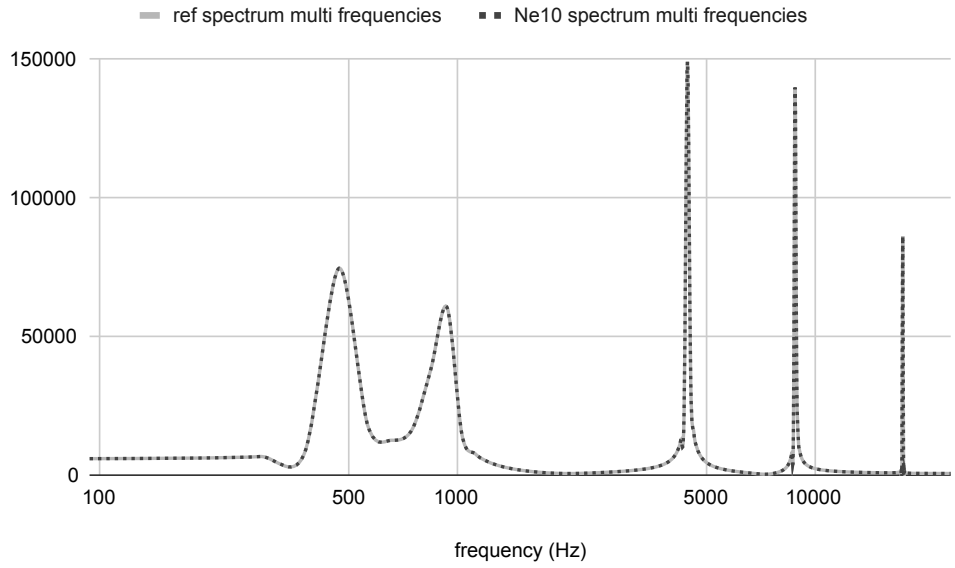


Figure 31 – Validation de la génération d'un spectre composé de multiples fréquences

Les MDCT ont enfin été testées sur base de données d'entrée générées aléatoirement. Les résultats de ces tests ont été validés en calculant la différence entre les valeurs du spectre généré par la MDCT de référence et les valeurs du spectre de la MDCT testée pour s'assurer que l'écart entre les deux soit acceptable.

Les tests de validation auraient pu être améliorés en automatisant la vérification, e.g. en générant une fois les données de référence attendues pour permettre le développement d'un code de test qui compare automatiquement les données de références avec les données à vérifier. En effet, devoir relancer les tests et vérifier les données à chaque fois qu'une modification est faite dans le code peut s'avérer laborieux et mettre en place des tests automatiques aurait permis de gagner un temps précieux.

11.2 Performances des MDCT

La mesure des performances a pour but de valider le bloc MDCT avant de l'intégrer au codec AAC. Le cahier des charges du stage ne contenait pas d'objectif à atteindre en terme de performances, ni absolu (e.g. un temps d'exécution maximal à respecter dans des conditions données), ni relatif (e.g. gagner un certain pourcentage de performances par rapport à une MDCT de référence).

L'objectif en terme de temps d'exécution n'étant pas fini, il a été décidé de tenter de gagner le maximum de performances sur le temps de mon stage. Le critère de réussite est dès lors d'obtenir des performances au moins équivalentes pour la version finale de la MDCT que pour ses itérations précédentes.

Le temps d'exécution de la MDCT *fixed point* doit évidemment être inférieur au temps d'exécution de l'algorithme de référence puisque celui-ci ne contient aucune optimisation. Ce temps peut toutefois être équivalent au temps d'exécution de la MDCT *floating point* : à performances équivalentes, l'algorithme *fixed point* rendra tout de même l'encodeur AAC plus performant en économisant les transtypages *integer-float* à l'entrée et à la sortie du bloc MDCT. En effet, les données sont reçues par la MDCT en *integer* et devront être traitées en *integer* par le bloc de quantification à la sortie de la MDCT.

Le temps d'exécution des différentes itérations de la MDCT a été mesuré sur base du code de l'**annexe N.1** compilé par les commandes CMake présentées dans l'**annexe N.2**. Le code permet de générer plusieurs exécutables en fonction de la variable de préprocesseur définie à la compilation, afin de pouvoir tester :

- La MDCT *Ne10 float 32 plain C* ;
- La MDCT *Ne10 integer 32 plain C* ;
- La MDCT *Ne10 integer 32 NEON* ;
- La MDCT de référence en *float 32*.

Le code de l'**annexe O.1**, compilé avec les commandes de l'**annexe O.2** génère un exécutable permettant de mesurer le temps moyen d'exécution de la MDCT *FFTW3 float 32*.

Les tests de performances sont lancés sur le Raspberry via une connexion SSH. Un script permet d'appeler tous les exécutables avec les paramètres voulus et les résultats affichés en console sont redirigés vers un fichier texte. Les tests sont réalisés en isolation, i.e. aucun processus non nécessaire au fonctionnement du Raspberry n'est lancé durant la réalisation des tests pour ne pas interférer et risquer d'allonger le temps d'exécution. Les mesures de performances ont été prises avec des exécutables compilés en mode *release* avec l'option `CMAKE_BUILD_TYPE=RELEASE`.

Les résultats présentés dans la table 2 ont été mesurés sur 10 000 000 d'exécutions (10 000 pour l'algorithme de référence). Afin de ne pas introduire d'aléatoire dans ces mesures, les MDCT ont toutes été testées avec le même signal temporel en entrée : un signal sinusoïdal à 440 Hz.

MDCT	**Temps d'exécution moyen (ns)**	**Écart type (ns)**
Ne10 i32 NEON	5796.35	0.02259×10^{-6}
Ne10 i32 C	22066.8	3.31025×10^{-6}
Ne10 f32	9020.1	0.02251×10^{-6}
FFTW3 f32	7446.84	1.29403×10^{-6}
Reference f32	29212.6×10^{3}	0.77552×10^{-6}

Table 2 – Tests de performances des algorithmes MDCT (données d'entrée identiques)

Les résultats montrent que la MDCT optimisée avec les instructions ARM NEON est bien plus rapide que les autres MDCT développées avec un temps d'exécution moyen de 5796.35 ns. L'objectif du stage est donc bien atteint.

L'algorithme MDCT de référence est le plus lent de tous avec un temps d'exécution moyen de 29 212.6 µs. Ce résultat est tout à fait normal puisque cet algorithme a été développé sans optimisation particulière.

La MDCT *fixed point* en *plain C* est la plus lente des MDCT optimisées avec un temps d'exécution moyen de 22 066.8 ns. La section 9.4 permet de comprendre en quoi ce résultat est cohérent puisque le processeur ARMv7 supporte de nombreuses instructions en *float* sur 32 bits et que l'arithmétique *fixed point* nécessite souvent plusieurs opérations là où une seule est nécessaire en *floating point*.

Ces résultats montrent que l'implémentation des instructions ARM NEON était nécessaire afin d'obtenir des performances acceptables. Sans l'implémentation d'une MDCT optimisée avec ces instructions, l'utilisation d'une MDCT *fixed point* aurait été compromis par les résultats insatisfaisants de l'implémentation *plain C*.

Enfin, pour les implémentation *floating point plain C*, la MDCT *Ne10* est un peu plus lente que la MDCT *FFTW3* avec un temps d'exécution moyen de 9020.1 ns contre 7446.84 ns. Ce résultat est cohérent avec les performances des différentes FFT mesurées à la section 8.4 : la FFT de *FFTW3* en *float 32* est en effet environ 2 µs plus rapide que la FFT de *Ne10* en *float 32 plain C*.

Ces tests de performances donnent des résultats du même ordre de grandeur avec des signaux différents donnés en entrée des MDCT testées. À titre, la table 3 montre les résultats des tests de performances réalisés dans les mêmes conditions que ceux présentés plus haut mais avec des données aléatoires différentes à chaque exécution en entrée des MDCT.

MDCT	**Temps d'exécution moyen (ns)**	**Écart type (ns)**
Ne10 i32 NEON	5738.84	0.29399×10^{-6}
Ne10 i32 C	21805.5	0.28367×10^{-6}
Ne10 f32	8948.4	0.26262×10^{-6}
FFTW3 f32	7574.63	0.13941×10^{-6}
Reference f32	29253.1×10^{3}	4.58387×10^{-6}

n

Table 3 – Tests de performances des algorithmes MDCT (données d'entrée variables)

12 Pistes d'amélioration

L'algorithme MDCT ne constitue qu'une première étape de la construction d'un encodeur AAC. Pour pouvoir être testé plus complètement, e.g. sur des fenêtres d'échantillons provenant d'un signal réel, toute une mécanique de lecture des données avec overlap aurait dû etre mise en place.

Après les lectures des données et leur passage dans la MDCT, le bloc suivant à implémentaer est le bloc de quantification. Les mêmes optimisations que celles aportées à la MDCT doivent lui être apportées avec l'utilisation d'une arithmétique *fixed point* reposant sur les opérations SIMD du proesseur ARM.

La MDCT doit être complétée avec une fonction de fenêtre. Cette fonction est typiquement une fonction basée sur un sinus ou un cosinus au travers de laquelle le signal temporel est passé avant d'être analysé par la MDCT. Cette fonction a pour but d'améliorer la transformation effectuée par la MDCT en atténuant l'effet de coupure aux extrémités de la fenêtre d'entrée de la MDCT.

Le fonction de fenêtre utilisée pour l'encodeur AAC, aussi aussi appelée MLT (*Mudulated Lapped Transform*) (figure 32), est donnée pour une fenêtre de taille $2N$, par l'équation :

$$w_n = sin\left[\frac{\pi}{2N}\left(n + \frac{1}{2}\right)\right]$$

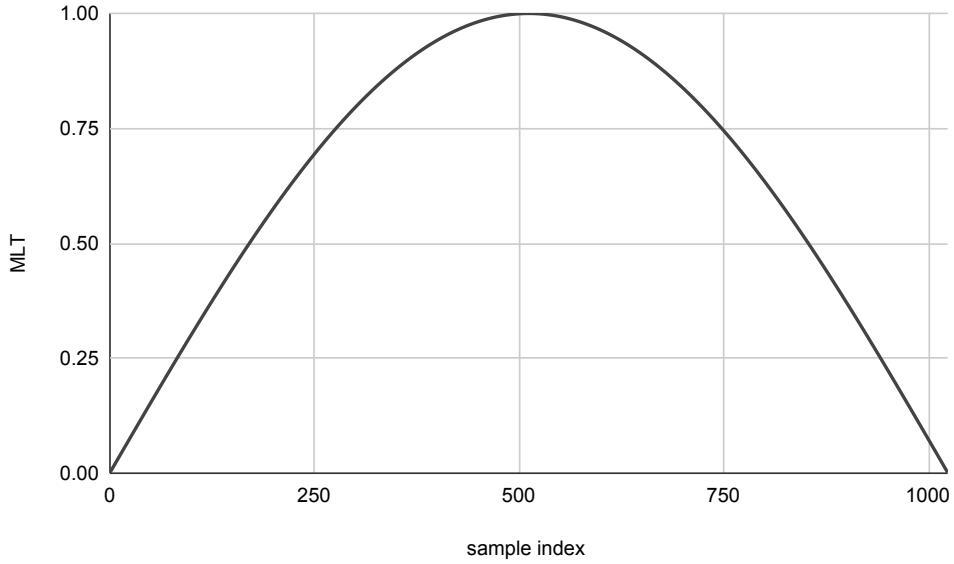


Figure 32 – Window function (MLT)

La figure 33 montre la comparaison entre les spectres de fréquences générés par la MDCT *Ne10 i32 neon* pour le même signal d'entrée, passé ou non au travers de la fonction de fenêtre pour le même signal sinusoïdal à 440 Hz que

celui montré dans la plupart des exemple de ce travail. La comparaison a également été faite avec quelque autres signaux. À chaque fois, la composante fréquentielle relevée est un peu plus nette avec la fonction de fenêtre.

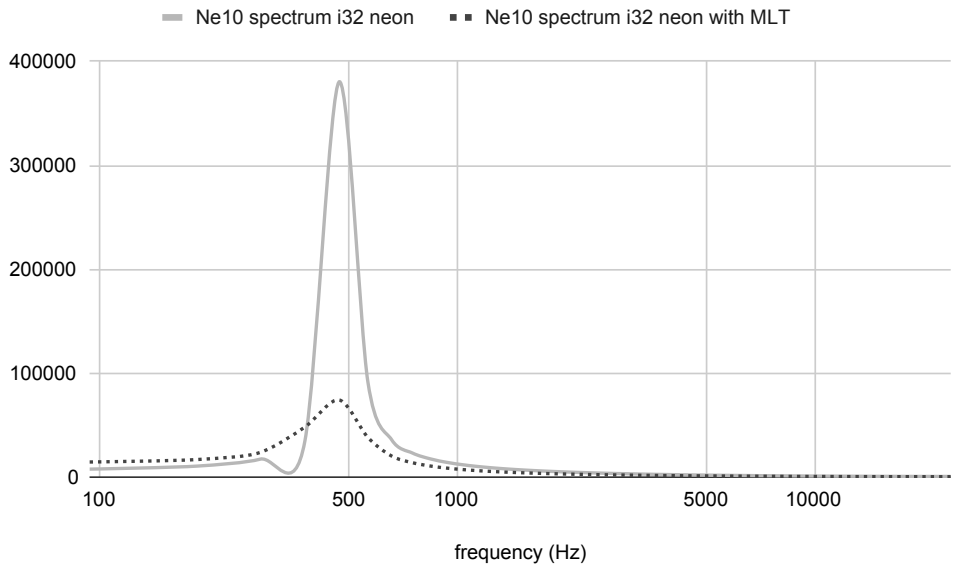


Figure 33 – Comparaison des spectres de fréquences générés par la MDCT avec et sans MLT (440 Hz)

Le graphique montre également une valeur plus basse pour la fréquence relevée sur base du signal passé par la MLT. Cette valeur tient désormais sur du Q15 sans dépassement. Il aurait été intéressant, si mon stage avait duré plus longtemps, de vérifier si le spectre de fréquences produit après le passage dans la MLT permettait toujours d'avoir ce type de résultats. Les performances de la MDCT pourraient alors encore être améliorées en ramenant plus d'opérations en 16 bits.

Pour pouvoir être intégrée intelligemment à la MDCT, la fonction de fenêtre aurait dû être combinée aux facteurs de *pre-twiddling* de la MDCT. Il aurait alors fallu relancer tous les tests afin de valider le nouvel algorithme. De la même manière, pour rendre l'encodeur plus performant, les premières opérations du bloc de quantification pourraient être effectuées en même temps que le *post-twiddling*.

Avec des tests automatisés, j'aurais probablement eu plus de temps à la fin de mon stage pour mettre en place cette fonction de fenêtre de manière sûre. Si je devais recommencer ce type de travail aujourd'hui, avec l'expérience acquise durant mon stage, je réfléchirai très probablement en début de développement à une manière plsu simple de valider les donnée, et peut-être également les performances.

Conclusion

Par rapport au cahier des charges reçu en début de stage :

- j'ai implémenté une MDCT basée sur une FFT (sur base d'un code d'exemple fourni)
- j'ai bien mis en place l'arithmétique fixed point mais les résultats n'étaient pas satisfaisants
- j'ai utilisé les instructions ARM NEON et j'ai obtenu un algorithme plus rapide que tous les autres algorithmes développés avant
- première brique d'un encodeur complet

Points positifs et négatifs :

- découverte d'EVS (technologies modernes, esprit d'entreprise collaboratif et axé sur la qualité de ce qui est délivré), du domaine audiovisuel
- rapport à la théorie
- frustration par rapport au manque de temps à pouvoir consacrer aux aspects théoriques

Ce que j'ai appris :

- traitement de signal audio

Références

[1] EVS Website, "Page d'accueil d'EVS Broadcast Equipment." [https://evs.com], consulté le 21 avril 2022.

[2] EVS Website, "Page de présentation des produits commercialisés par EVS Broadcast Equipment." [https://evs.com/products], consulté le 21 avril 2022.

[3] Wikipedia, "Codec." [https://en.wikipedia.org/wiki/Codec], consulté le 2 mai 2022.

[4] Wikipedia, "Moving picture experts group." [https://en.wikipedia.org/wiki/Moving_Picture_Experts_Group], consulté le 30 mars 2022.

[5] "Information technology – Generic coding of moving pictures and associated audio information – Part 7 : Advanced Audio Coding (AAC)," standard, International Organization for Standardization, 2006. [https://www.iso.org/standard/43345.html].

[6] "Information technology – Coding of audio-visual objects – Part 3 : Audio," standard, International Organization for Standardization, 2019. [https://www.iso.org/standard/76383.html].

[7] K. Brandenburg, "Mp3 and aac explained," *AES 17th International Conference on High Quality Audio Coding*, 1999.

[8] Wikipedia, "Psychoacoustics." [https://en.wikipedia.org/wiki/Psychoacoustics], consulté le 2 mars 2022.

[9] J. Herre and S. Dick, "Psychoacoustic models for perceptual audio coding—a tutorial review," *Applied Sciences*, vol. 9, p. 2854, 07 2019.

[10] "Site internet Raspberry Pi." [https://www.raspberrypi.com/].

[11] Wikipedia, "Modified discrete cosine transform." [https://en.wikipedia.org/wiki/Modified_discrete_cosine_transform], consulté le 17 septembre 2021.

[12] "Code d'exemple d'une MDCT basée sur une FFT." [https://www.dsprelated.com/showcode/196.php].

[13] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[14] "Site de la librairie FFTW." [http://www.fftw.org/].

[15] Project Ne10 Website, "Documentation du projet Ne10." [http://projectne10.github.io/Ne10/doc/], consulté le 9 mai 2022.

[16] "The 3-Clause BSD License." [https://opensource.org/licenses/BSD-3-Clause], consulté le 9 mai 2022.

[17] "Instruction Set Assembly Guide for Armv7 and earlier Arm architectures – Version 2.0 – Reference Guide." [https://developer.arm.com/documentation/100076/0200].

[18] E. Oberstar, "Fixed-Point Representation & Fractional Math Revison 1.2," 08 2007.

[19] Wikipedia, "Single instruction, multiple data." [https://en.wikipedia.org/wiki/Single_instruction,_multiple_data], consulté le 29 mai 2022.

[20] "Liste des intrinsèques ARM NEON supportée par GCC." [https://gcc.gnu.org/onlinedocs/gcc-4.6.1/gcc/ARM-NEON-Intrinsics.html].

[21] ARM Community blogs, "Série de tutoriels *Coding for Neon*," 2013. [https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/coding-for-neon---part-1-load-and-stores].

Liste des annexes

A Raspberry Pi 4 CPU info

Informations sur les CPU du Raspberry Pi 4 contenues dans le fichier /proc/cpuinfo.

```
$ cat /proc/cpuinfo
processor       : 0
model name      : ARMv7 Processor rev 3 (v7l)
BogoMIPS        : 108.00
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae
 evtstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 3

processor       : 1
model name      : ARMv7 Processor rev 3 (v7l)
BogoMIPS        : 108.00
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae
 evtstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 3

processor       : 2
model name      : ARMv7 Processor rev 3 (v7l)
BogoMIPS        : 108.00
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae
 evtstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 3

processor       : 3
model name      : ARMv7 Processor rev 3 (v7l)
BogoMIPS        : 108.00
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae
 evtstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 3

Hardware        : BCM2711
Revision        : b03112
Serial          : 1000000022221a34
Model           : Raspberry Pi 4 Model B Rev 1.2
```

B CMake principal

Fichier CMake principal placé à la racine du projet. Il permet de compiler :

- le projet *audio_encoding* contenant les différentes MDCT et leurs tests : les commandes CMake de ce sous-projet sont présentées dans les annexes suivantes sous le code qu'elles permettent de compiler ;
- la librairie *Ne10* : les variables suivantes sont initialisées conformément aux recommendations de la documentation pour la compilation de la librairie :
  - NE10_LINUX_TARGET_ARCH est initialisée à armv7 (l'architecture du Raspberry Pi 4) ;
  - GNULINUX_PLATFORM est initialisée à ON ;
  - BUILD_DEBUG est initialisée à ON si le projet est compilé en mode *debug*.

```
cmake_minimum_required(VERSION 3.13)

set(NE10_LINUX_TARGET_ARCH armv7)
set(GNULINUX_PLATFORM ON)
if (CMAKE_BUILD_TYPE STREQUAL "DEBUG")
    set(BUILD_DEBUG ON)
endif (CMAKE_BUILD_TYPE STREQUAL "DEBUG")

add_subdirectory(audio_encoding)
add_subdirectory(Ne10)
```

C Valeurs constantes des MDCT

Le fichier `mdct_constants.h` rassemble les valeurs constantes des MDCT pour une fenêtre d'entrée de 1024 échantillons.

```
// Sampling frequency: 48kHz
#define FS                  48000

// Window length and derived constants
#define MDCT_WINDOW_LEN     1024
#define MDCT_M              (MDCT_WINDOW_LEN>>1)     // spectrum size
#define MDCT_M2             (MDCT_WINDOW_LEN>>2)     // fft size
#define MDCT_M4             (MDCT_WINDOW_LEN>>3)
#define MDCT_M32            (3*(MDCT_WINDOW_LEN>>2))
#define MDCT_M52            (5*(MDCT_WINDOW_LEN>>2))
```

D Algorithmes de référence

D.1 MDCT de référence en *floating point*

Algorithme de référence basé sur la formule mathématique de la MDCT. Le *template* permet de réaliser les calculs en *float* ou en *double*.

```
#include <cmath>

#include "mdct_constants.h"

template<typename FLOAT>
void ref_float_mdct(FLOAT *time_signal, FLOAT *spectrum)
{
    FLOAT scale = 2.0 / sqrt(MDCT_WINDOW_LEN);
    FLOAT factor1 = 2.0 * M_PI / static_cast<FLOAT>(MDCT_WINDOW_LEN);
    FLOAT factor2 = 0.5 + static_cast<FLOAT>(MDCT_M2);
    for (int k = 0; k < MDCT_M; ++k)
    {
        FLOAT result = 0.0;
        FLOAT factor3 = (k + 0.5) * factor1;
        for (int n = 0; n < MDCT_WINDOW_LEN; ++n)
        {
            result += time_signal[n] * cos((static_cast<FLOAT>(n) + factor2) * factor3);
        }
        spectrum[k] = scale * result;
    }
}
```

D.2 MDCT de référence en *fixed point*

Algorithme de référence basé sur la formule mathématique de la MDCT. Le spectre est calculé en *double* puis converti en *integer* sur 32 bits en représentation Q15.

```
#include <cassert>

#include "ref_mdct.h"

void ref_int_mdct(int16_t *time_signal, int32_t *spectrum)
{
    double scale = sqrt(MDCT_WINDOW_LEN) / 2.0;      // MDCT scale (2/sqrt(WIN_LEN)) + Q15 scale
    double factor1 = 2.0 * M_PI / MDCT_WINDOW_LEN;
    double factor2 = 0.5 + MDCT_M2;
    for (int k = 0; k < MDCT_M; ++k)
    {
        double result = 0.0;
        double factor3 = (k + 0.5) * factor1;
        for (int n = 0; n < MDCT_WINDOW_LEN; ++n)
        {
            result += time_signal[n] * cos((n + factor2) * factor3);
        }
        assert(round(result*scale) == static_cast<int32_t>(round(result*scale)));
        spectrum[k] = static_cast<int32_t>(round(result/scale));
    }
}
```

E Génération d'un signal sinusoïdal

E.1 Génération d'un signal sinusoïdal en *floating point*

Code de génération d'un signal sinusoïdal en *float* ou en *double*. Le *template* permet de générer ces deux types de signaux avec le même code.

```
#include <cmath>

template<typename FLOAT>
void sin_float(FLOAT *out, int n_samples, double amplitude,
    double frequency, double phase_shift, int sampling_frequency)
{
    FLOAT omega = 2.0 * M_PI * frequency / static_cast<FLOAT>(sampling_frequency);
    for (int i = 0; i < n_samples; ++i)
    {
        out[i] = amplitude * sin(static_cast<FLOAT>(i) * omega + phase_shift);
    }
}
```

E.2 Génération d'un signal sinusoïdal en *fixed point*

La génération du signal sinusoïdal en *integer* fait appel à la génération du signal sinusoïdal en *double* avant de convertir le résultat en *integer* (représentation Q15).

```
#include <cstring>

#include "sin_wave.h"

void sin_int(int16_t *out, int n_samples, double amplitude,
    double frequency, double phase_shift, int sampling_frequency)
{
    double scale = 1.0;
    if (abs(amplitude) < 1.0) scale *= amplitude;

    double *temp_sin = static_cast<double *>(malloc(n_samples*sizeof(double)));
    memset(temp_sin, 0, n_samples*sizeof(double));

    sin_float<double>(temp_sin, n_samples, scale, frequency, phase_shift, sampling_frequency);

    for (int i = 0; i < n_samples; ++i)
    {
        out[i] = static_cast<int16_t>(temp_sin[i]*pow(2.0, 15.0));
    }
}
```

F Implémentation de la MDCT basée sur la FFT de *FFTW3*

F.1 Header

Header de la classe `mdct_fftw3_f32` : MDCT basée sur la FFT de la librairie *FFTW3* en *float* (32 bits). La classe contient les structures de données `fft_in` et `fft_out`, le tableau de facteurs de `twiddle` utilisé pour le pre- et le post-processing et la configuration de la FFT (`fft_plan`). L'implémentation des fonctions de ce header est présentée dans les annexes suivantes.

```
#include <fftw3.h>

#include "mdct_constants.h"

class fftw3_mdct_f32
{
    private:
    fftwf_plan fft_plan;       // FFT configuration
    fftwf_complex *fft_in;     // FFT input buffer
    fftwf_complex *fft_out;    // FFT output buffer
    float twiddle[MDCT_M];

    public:
    fftw3_mdct_f32();
    ~fftw3_mdct_f32();
    void mdct(float *time_signal, float *spectrum);
    void imdct(float *spectrum, float *time_signal);
};
```

F.2 Constructeur

Initialisation de la MDCT dans le constructeur de la classe `mdct_fftw3_f32` :

- Le tableau de `twiddle` est initialisé en *float* sur 32 bits ;
- La FFT de *FFTW3* est initialisée en une dimension (pour l'audio) avec la taille de la FFT réduite à un quart de la taille de la fenêtre d'entrée par le pre-processing et avec l'option `FFTW_MEASURE` plus lente à l'initialisation mais qui permet d'optimiser le temps d'exécution de la FFT ;
- Les tableaux contenant les données d'entrée (`fft_in`) et de sortie (`fft_out`) de la FFT sont alloués dynamiquement avec la fonction de *FFTW3* et ils sont passés en paramètre à la configuration de la FFT.

```
#include <cmath>

fftw3_mdct_f32::fftw3_mdct_f32()
{
    float alpha = M_PI / (8.f * MDCT_M);
    float omega = M_PI / MDCT_M;
    float scale = sqrt(sqrt(2.f / MDCT_M));

    for (int i = 0; i < MDCT_M2; ++i)
    {
        float x = omega*i + alpha;
        twiddle[2*i] = scale * cos(x);
        twiddle[2*i+1] = scale * sin(x);
    }
```

```
    fft_in = (fftwf_complex *)fftwf_malloc(sizeof(fftwf_complex) * MDCT_M2);
    fft_out = (fftwf_complex *)fftwf_malloc(sizeof(fftwf_complex) * MDCT_M2);
    fft_plan = fftwf_plan_dft_1d(MDCT_M2, fft_in, fft_out, FFTW_FORWARD, FFTW_MEASURE);
}
```

F.3 Destructeur

Destructeur de la classe `mdct_fftw3_f32` qui permet de libérer la mémoire allouée aux tableaux d'entrée et de sortie de la FFT et à sa configuration avec les fonctions appropriées fournies par la librairie *FFTW3*.

```
fftw3_mdct_f32::~fftw3_mdct_f32()
{
    fftwf_destroy_plan(fft_plan);
    fftwf_free(fft_in);
    fftwf_free(fft_out);
}
```

F.4 Fonction MDCT

Implémentation de l'algorithme de MDCT basé sur la FFT de la librairie *FFTW3* :

- Initialisation du tableau d'entrée de la FFT : les opérations de *pre-twiddling* permettent de réduire la fenêtre d'entrée de la FFT ;
- Appel de la fonction FFT de *FFTW3* ;
- Calcul du spectre de fréquences : les opérations de *post-twiddling* permettent de calculer le spectre à partir des données de sortie de la FFT et des facteurs de *twiddle*.

```
void fftw3_mdct_f32::mdct(float *time_signal, float *spectrum)
{
    float *cos_tw = twiddle;
    float *sin_tw = cos_tw + 1;

    /* odd/even folding and pre-twiddle */
    float *xr = (float *)fft_in;
    float *xi = xr + 1;

    for (int i = 0; i < MDCT_M2; i += 2)
    {
        float r0 = time_signal[MDCT_M32-1-i] + time_signal[MDCT_M32+i];
        float i0 = time_signal[MDCT_M2+i] - time_signal[MDCT_M2-1-i];

        float c = cos_tw[i];
        float s = sin_tw[i];

        xr[i] = r0*c + i0*s;
        xi[i] = i0*c - r0*s;
    }

    for(int i = MDCT_M2; i < MDCT_M; i += 2)
    {
        float r0 = time_signal[MDCT_M32-1-i] - time_signal[-MDCT_M2+i];
        float i0 = time_signal[MDCT_M2+i] + time_signal[MDCT_M52-1-i];
```

```
        float c = cos_tw[i];
        float s = sin_tw[i];

        xr[i] = r0*c + i0*s;
        xi[i] = i0*c - r0*s;
    }

    /* complex FFT of size MDCT_M2 */
    fftwf_execute(fft_plan);

    /* post-twiddle */
    xr = (float *)fft_out;
    xi = xr + 1;

    for (int i = 0; i < MDCT_M; i += 2)
    {
        float r0 = xr[i];
        float i0 = xi[i];

        float c = cos_tw[i];
        float s = sin_tw[i];

        spectrum[i] = -r0*c - i0*s;
        spectrum[MDCT_M-1-i] = -r0*s + i0*c;
    }
}
```

G Validation de la MDCT *FFTW3* en *float 32*

G.1 Code source

Test de la MDCT basée sur la FFT de *FFTW3* en *float 32* avec un signal d'entrée sinusoïdal à 440 Hz :

- Génération et affichage d'un signal sinusoïdal à 440 Hz ;
- Calcul et affichage du spectre de fréquences de ce signal ;
- Opération inverse de la MDCT et affichage du signal temporel calculé à partir du spectre.

Les différentes données sont écrites dans un fichier CSV afin de pouvoir les exploiter sous forme graphique.

```
#include <iomanip>
#include <iostream>
#include <fstream>

#include <cstring>

#include "mdct_constants.h"
#include "fftw3_mdct_f32.h"
#include "sin_wave.h"

/**
    * @brief MDCT algorithm calling the FFT of the fftw3 library
    * Code based on https://www.dsprelated.com/showcode/196.php
    * Change the MDCT_WINDOW_LEN to test the MDCT with other spectrum sizes
    */
int main(void)
{
    // input time signal (440 Hz)
    float time_in[MDCT_WINDOW_LEN];
    sin_float(time_in, MDCT_WINDOW_LEN, 0.9, 440.0, 0.0, FS);

    // output time signal (generated by the IMDCT)
    float time_out[MDCT_WINDOW_LEN];
    memset(time_out, 0, MDCT_WINDOW_LEN*sizeof(float));

    // frequency spectrum (generated by the MDCT)
    float spectrum[MDCT_M];
    memset(spectrum, 0, MDCT_M*sizeof(float));

    // perform the MDCT and IMDCT
    fftw3_mdct_f32 fftw3_mdct;
    fftw3_mdct.mdct(time_in, spectrum);
    fftw3_mdct.imdct(spectrum, time_out);

    // print the results in CSV file
    std::ofstream csv_file;
    csv_file.open("fftw3_mdct_f32.csv");
    csv_file << "time (ms),signal in,signal out,frequency (Hz),spectrum" << std::endl;

    for (int i = 0; i < MDCT_M; ++i)
    {
        csv_file << i*1000.0/FS << "," << time_in[i] << "," << time_out[i] << ","
            << (i+1.0)*FS/MDCT_WINDOW_LEN << "," << std::abs(spectrum[i]) << "," << std::endl;
    }
```

```
    for (int i = MDCT_M; i < MDCT_WINDOW_LEN; ++i)
    {
        csv_file << i*1000.0/(FS) << "," << time_in[i] << "," << time_out[i] << "," << std::endl;
    }

    csv_file.close();

    return 0;
}
```

G.2 Compilation

Commandes CMake permettant de compiler le code d'exemple.

```
# MDCT using the fftw3 library f32
add_executable(fftw3_mdct_f32 test/validation/fftw3_example.cpp
    src/fftw3_mdct_f32.cpp src/sin_wave.cpp)
target_link_libraries(fftw3_mdct_f32 fftw3f)
```

H Implémentation de la MDCT basée sur la FFT de *Ne10* en *floating point*

H.1 Header

Header de la classe `mdct_ne10_f32_c` : MDCT basée sur la FFT de la librairie *Ne10* en *float* (32 bits). La classe contient les structures de données `fft_in` et `fft_out`, le tableau de facteurs de `twiddle` utilisé pour le pre- et le post-processing et la configuration de la FFT (`cfg`). L'implémentation des fonctions de ce header est présentée dans les annexes suivantes.

```
#pragma once

#include "mdct_constants.h"
#include "NE10.h"

class ne10_mdct_f32_c
{
    private:
    ne10_fft_cfg_float32_t cfg;                                                  // Ne10 configuration
    ne10_fft_cpx_float32_t fft_in[MDCT_M2] __attribute__((aligned(16)));         // Ne10 FFT input buffer
    ne10_fft_cpx_float32_t fft_out[MDCT_M2] __attribute__((aligned(16)));        // Ne10 FFT output buffer
    float twiddle[MDCT_M] __attribute__((aligned(16)));                          // twiddle factors

    public:
    ne10_mdct_f32_c();
    ~ne10_mdct_f32_c();
    void mdct(float *time_signal, float *spectrum);
};
```

H.2 Constructeur

Initialisation de la MDCT dans le constructeur de la classe `mdct_ne10_f32_c` :

- Le tableau de `twiddle` est initialisé en *float* sur 32 bits ;
- La configuration de la FFT de *Ne10* est initialisée en *complex to complex* en *float 32* avec en paramètre la taille de la fenêtre de la FFT réduite à un quart de la taille de la fenêtre d'entrée.

```
ne10_mdct_f32_c::ne10_mdct_f32_c()
{
    float alpha = M_PI / (8.0 * static_cast<float>(MDCT_M));
    float omega = M_PI / static_cast<float>(MDCT_M);
    float scale = sqrt(sqrt(2.0 / static_cast<float>(MDCT_M)));
    for (int i = 0; i < MDCT_M2; ++i)
    {
        float x = omega * i + alpha;
        twiddle[2*i] = static_cast<float>(scale * cos(x));
        twiddle[2*i+1] = static_cast<float>(scale * sin(x));
    }

    cfg = ne10_fft_alloc_c2c_float32_c(MDCT_M2);
}
```

H.3 Destructeur

Destructeur de la classe `mdct_ne10_f32_c` qui permet de libérer la mémoire allouée à la configuration de la FFT avec la fonction appropriée de la librairie *Ne10*.

```
ne10_mdct_f32_c::~ne10_mdct_f32_c()
{
    ne10_fft_destroy_c2c_float32(cfg);
}
```

H.4 Fonction MDCT

Implémentation de l'algorithme de MDCT basé sur la FFT de la librairie *Ne10* en *float 32* et en *plain C* :

- Initialisation du tableau d'entrée de la FFT : les opérations de *pre-twiddling* permettent de réduire la fenêtre d'entrée de la FFT ;
- Appel de la fonction FFT de *Ne10* ;
- Calcul du spectre de fréquences : les opérations de *post-twiddling* permettent de calculer le spectre à partir des données de sortie de la FFT et des facteurs de *twiddle*.

```
void ne10_mdct_f32_c::mdct(float *time_signal, float *spectrum)
{
    // pre-twiddling
    float *cos_tw = twiddle;
    float *sin_tw = cos_tw + 1;
    for (int i = 0; i < MDCT_M2; i += 2)
    {
        float r0 = time_signal[MDCT_M32-1-i] + time_signal[MDCT_M32+i];
        float i0 = time_signal[MDCT_M2+i] - time_signal[MDCT_M2-1-i];

        float c = cos_tw[i];
        float s = sin_tw[i];

        fft_in[i/2].r = r0*c + i0*s;
        fft_in[i/2].i = i0*c - r0*s;
    }

    for (int i = MDCT_M2; i < (MDCT_M); i += 2)
    {
        float r0 = time_signal[MDCT_M32-1-i] - time_signal[-MDCT_M2+i];
        float i0 = time_signal[MDCT_M2+i] + time_signal[MDCT_M52-1-i];

        float c = cos_tw[i];
        float s = sin_tw[i];

        fft_in[i/2].r = r0*c + i0*s;
        fft_in[i/2].i = i0*c - r0*s;
    }

    // FFT
    ne10_fft_c2c_1d_float32_c(fft_out, fft_in, cfg, 0);
```

```
    // post-twiddling
    for (int i = 0; i < (MDCT_M); i += 2)
    {
        float r0 = fft_out[i/2].r;
        float i0 = fft_out[i/2].i;

        float c = cos_tw[i];
        float s = sin_tw[i];

        spectrum[i] = -r0*c - i0*s;
        spectrum[(MDCT_M)-1-i] = -r0*s + i0*c;
    }
}
```

I Mesure des performances des FFT de *Ne10*

I.1 Code source

Code permettant de tester la vitesse d'exécution moyenne de différentes FFT proposées par la librairie *Ne10*. La moyenne est calculée sur 10 000 000 d'exécutions. Les données d'entrée de la FFT sont générées aléatoirement et sont différentes pour chaque exécution. Les variables de préprocesseur définies à la compilation permettent à partir du même code de mesurer le temps d'exécution moyen avec écart type :

- de la FFT *complex to complex* en *float 32* en *plain C* ou avec les optimisations NEON ;
- de la FFT *complex to complex* en *integer 32* en *plain C* ou avec les optimisations NEON ;
- de la FFT *complex to complex* en *integer 16* en *plain C* ou avec les optimisations NEON.

```
#include <iomanip>
#include <iostream>
#include <limits>

#include <cmath>
#include <cstring>

#include "mdct_constants.h"
#include "Timers.h"
#include "NE10.h"

#ifdef F32          // 32 bits floating point arithmetic

#define INPUT_RANGE             1.8
#define INPUT_DATA              ne10_fft_cpx_float32_t
#define OUTPUT_DATA             ne10_fft_cpx_float32_t
#define FFT_CONFIG              ne10_fft_cfg_float32_t
#define DESTROY_CONFIG          ne10_fft_destroy_c2c_float32

#ifdef NEON
#define ALLOC_CONFIG            ne10_fft_alloc_c2c_float32_neon
#define PERFORM_FFT             ne10_fft_c2c_1d_float32_neon
#else
#define ALLOC_CONFIG            ne10_fft_alloc_c2c_float32_c
#define PERFORM_FFT             ne10_fft_c2c_1d_float32_c
#endif

#elif I32           // 32 bits fixed point arithmetic

#define INPUT_RANGE             std::numeric_limits<int16_t>::max()*2
#define INPUT_DATA              ne10_fft_cpx_int32_t
#define OUTPUT_DATA             ne10_fft_cpx_int32_t
#define FFT_CONFIG              ne10_fft_cfg_int32_t
#define DESTROY_CONFIG          ne10_fft_destroy_c2c_int32

#ifdef NEON
#define ALLOC_CONFIG            ne10_fft_alloc_c2c_int32_neon
#define PERFORM_FFT             ne10_fft_c2c_1d_int32_neon
#else
#define ALLOC_CONFIG            ne10_fft_alloc_c2c_int32_c
#define PERFORM_FFT             ne10_fft_c2c_1d_int32_c
#endif
```

```
#else                // 16 bits fixed point arithmetic

#define INPUT_RANGE             std::numeric_limits<int16_t>::max()*2
#define INPUT_DATA              ne10_fft_cpx_int16_t
#define OUTPUT_DATA             ne10_fft_cpx_int16_t
#define FFT_CONFIG              ne10_fft_cfg_int16_t
#define ALLOC_CONFIG            ne10_fft_alloc_c2c_int16
#define DESTROY_CONFIG          ne10_fft_destroy_c2c_int16

#ifdef NEON
#define PERFORM_FFT             ne10_fft_c2c_1d_int16_neon
#else
#define PERFORM_FFT             ne10_fft_c2c_1d_int16_c
#endif

#endif

#define RUNS            10000000
#define FFT_SCALE_FLAG  0

int main()
{
    // print which FFT will be tested
#ifdef F32
#ifdef NEON
    std::cout << "FFT Ne10 f32 NEON" << std::endl;
#else
    std::cout << "FFT Ne10 f32 plain C" << std::endl;
#endif

#elif I32
#ifdef NEON
    std::cout << "FFT Ne10 i32 NEON" << std::endl;
#else
    std::cout << "FFT Ne10 i32 plain C" << std::endl;
#endif
#else
#ifdef NEON
    std::cout << "FFT Ne10 i16 NEON" << std::endl;
#else
    std::cout << "FFT Ne10 i16 plain C" << std::endl;
#endif
#endif

    // seed the random
    srand(static_cast<unsigned>(time(0)));

    // initialize the configuration
    FFT_CONFIG cfg = ALLOC_CONFIG(MDCT_M2);

    // start the loop executing the FFTs
    int64_t *runtimes = static_cast<int64_t *>(malloc(RUNS * sizeof(int64_t)));
    for (int i = 0; i < RUNS; ++i)
    {
        // initialize an empty spectrum
        OUTPUT_DATA spectrum[MDCT_M2] __attribute__((aligned(16)));
        memset(&spectrum, 0, (MDCT_M2)*sizeof(OUTPUT_DATA));
```

```
        // generate random input data
        INPUT_DATA time_signal[MDCT_M2] __attribute__((aligned(16)));
        for (int i = 0; i < MDCT_M2; ++i)
        {
            time_signal[i].r = INPUT_RANGE * rand() / RAND_MAX - INPUT_RANGE / 2;
            time_signal[i].i = INPUT_RANGE * rand() / RAND_MAX - INPUT_RANGE / 2;
        }

        // perfom the FFT and measure the run time
        EvsHwLGPL::CTimers timer;
        timer.Start();
#ifdef F32
        PERFORM_FFT(time_signal, spectrum, cfg, 0);
#else
        PERFORM_FFT(time_signal, spectrum, cfg, 0, FFT_SCALE_FLAG);
#endif
        timer.Stop();
        runtimes[i] = timer.GetTimeElapsed();
    }

    // clean
    DESTROY_CONFIG(cfg);

    // compute the average
    double avg = 0.0;
    for (int i = 0; i < RUNS; ++i) avg += static_cast<double>(runtimes[i]);
    avg = avg / static_cast<double>(RUNS);
    std::cout << "average run time:    " << avg << " ns" << std::endl;

    // compute the standard deviation
    double dev = 0.0;
    for (int i = 0; i < RUNS; ++i) dev += static_cast<double>(runtimes[i]) - avg;
    dev = dev * dev / static_cast<double>(RUNS);
    dev = sqrt(dev);
    std::cout << "standard deviation: " << dev << " ns" << std::endl;

    return 0;
}
```

I.2 Compilation

Commandes CMake utilisées pour générer les exécutables permettant de mesurer le temps d'exécution de différentes FFT proposées par la librairie *Ne10*. En fonction des variables de préprocesseur définies, les exéctables suivants sont générés :

- `run_fft_f32_c` est généré si la variable `F32` est définie pour mesurer le temps d'exécution de la FFT *float 32 plain C* ;
- `run_fft_i32_c` est généré si la variable `I32` est définie pour mesurer le temps d'exécution de la FFT *integer 32 plain C* ;
- `run_fft_i16_c` est généré par défaut pour mesurer le temps d'exécution de la FFT *integer 16 plain C* ;
- `run_fft_f32_neon` est généré si les variables `F32` et `NEON` sont définies pour mesurer le temps d'exécution de la FFT *float 32* avec optimisations NEON ;
- `run_fft_i32_neon` est généré si les variables `I32` et `NEON` sont définies pour mesurer le temps d'exécution de la FFT *integer 32* avec optimisations NEON ;

- `run_fft_i16_neon` est généré si la variable NEON est définie pour mesurer le temps d'exécution de la FFT *integer 16* avec optimisations NEON.

```
# Ne10 FFT performance (float32 plain C)
add_executable(run_ne10_fft_f32_c test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_f32_c PUBLIC -DF32)
target_link_libraries(run_ne10_fft_f32_c NE10)

# Ne10 FFT performance (int32 plain C)
add_executable(run_ne10_fft_i32_c test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_i32_c PUBLIC -DI32)
target_link_libraries(run_ne10_fft_i32_c NE10)

# Ne10 FFT performance (int16 plain C)
add_executable(run_ne10_fft_i16_c test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_i16_c PUBLIC -DI16)
target_link_libraries(run_ne10_fft_i16_c NE10)

# Ne10 FFT performance (float32 with neon optimizations)
add_executable(run_ne10_fft_f32_neon test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_f32_neon PUBLIC -DF32 -DNEON)
target_link_libraries(run_ne10_fft_f32_neon NE10)

# Ne10 FFT performance (int32 with neon optimizations)
add_executable(run_ne10_fft_i32_neon test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_i32_neon PUBLIC -DI32 -DNEON)
target_link_libraries(run_ne10_fft_i32_neon NE10)

# Ne10 FFT performance (int16 with neon optimizations)
add_executable(run_ne10_fft_i16_neon test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_i16_neon PUBLIC -DI16 -DNEON)
target_link_libraries(run_ne10_fft_i16_neon NE10)
```

J Mesure des performances des FFT de *FFTW3*

J.1 Code source

Code permettant de tester la vitesse d'exécution moyenne de la FFT en *float 32* de la librairie *FFTW3*. La moyenne est calculée sur 10 000 000 d'exécutions. Les données d'entrée de la FFT sont générées aléatoirement et sont différentes pour chaque exécution.

```
#include <iomanip>
#include <iostream>
#include <cmath>

#include <fftw3.h>

#include "mdct_constants.h"
#include "Timers.h"

#define RUNS 10000000

int main()
{
    // print which FFT will be tested
    std::cout << "FFT FFTW3 f32 plain C" << std::endl;

    // seed the random
    srand(static_cast<unsigned>(time(0)));

    // start the loop executing the FFTs
    int64_t *runtimes = static_cast<int64_t *>(malloc(RUNS * sizeof(int64_t)));
    for (int i = 0; i < RUNS; ++i)
    {
        // initialize an empty spectrum
        fftwf_complex *fft_out = (fftwf_complex *)fftwf_malloc(sizeof(fftwf_complex) * MDCT_M2);

        // generate random input data
        fftwf_complex *fft_in = (fftwf_complex *)fftwf_malloc(sizeof(fftwf_complex) * MDCT_M2);
        float *x = (float *)fft_in;
        for (int i = 0; i < MDCT_M2; ++i)
        {
            x[i] = 1.8f * rand() / RAND_MAX - 1.8f / 2.0f;
        }

        // initialize the configuration
        fftwf_plan fft_plan = fftwf_plan_dft_1d(MDCT_M2, fft_in, fft_out,
            FFTW_FORWARD, FFTW_MEASURE);

        // perfom the FFT and measure the run time
        EvsHwLGPL::CTimers timer;
        timer.Start();
        fftwf_execute(fft_plan);
        timer.Stop();
        runtimes[i] = timer.GetTimeElapsed();

        // clean
        fftwf_destroy_plan(fft_plan);
        fftwf_free(fft_in);
        fftwf_free(fft_out);
    }
```

```
    // compute the average
    double avg = 0.0;
    for (int i = 0; i < RUNS; ++i) avg += static_cast<double>(runtimes[i]);
    avg = avg / static_cast<double>(RUNS);
    std::cout << "average run time:    " << avg << " ns" << std::endl;

    // compute the standard deviation
    double dev = 0.0;
    for (int i = 0; i < RUNS; ++i) dev += static_cast<double>(runtimes[i]) - avg;
    dev = dev * dev / static_cast<double>(RUNS);
    dev = sqrt(dev);
    std::cout << "standard deviation: " << dev << " ns" << std::endl;

    return 0;
}
```

J.2 Compilation

Commandes CMake utilisées pour générer l'exécutable permettant de mesurer le temps d'exécution de la FFT *float 32* de la librairie *FFTW3*.

```
# FFTW3 FFT performance (float32)
add_executable(run_fftw3_fft_f32 test/performance/run_fftw3_fft_f32.cpp src/Timers.cpp)
target_link_libraries(run_fftw3_fft_f32 fftw3f)
```

K Implémentation de la MDCT basée sur la FFT de *Ne10* en *fixed point*

K.1 Header

Header de la classe `mdct_ne10_i32_c` : MDCT basée sur la FFT de la librairie *Ne10* en *integer* (32 bits). La classe contient les structures de données `fft_in` en représentation Q1.15 et `fft_out` en Q9.15, le tableau de facteurs de `twiddle` utilisé pour le pre- et le post-processing et la configuration de la FFT (`cfg`). L'implémentation des fonctions de ce header est présentée dans les annexes suivantes.

```
#pragma once

#include "mdct_constants.h"
#include "NE10.h"

class ne10_mdct_i32_c
{
    private:
    ne10_fft_cfg_int32_t cfg;                                          // Ne10 configuration
    ne10_fft_cpx_int32_t fft_in[MDCT_M2] __attribute__((aligned(16))); // Ne10 FFT input buffer
                                                                       //   Q1.15
    ne10_fft_cpx_int32_t fft_out[MDCT_M2] __attribute__((aligned(16))); // Ne10 FFT output buffer
                                                                        //   Q9.15
    int16_t twiddle[MDCT_M] __attribute__((aligned(16)));              // MDCT twiddle factors

    public:
    ne10_mdct_i32_c();
    ~ne10_mdct_i32_c();
    void mdct(int16_t *time_signal, int32_t *spectrum);
};
```

K.2 Constructeur

Initialisation de la MDCT dans le constructeur de la classe `mdct_ne10_i32_c` :

- Le tableau de `twiddle` est initialisé en *double* puis converti en *integer* (représentation Q15) ;
- La configuration de la FFT de *Ne10* est initialisée en *complex to complex* en *integer 32* avec en paramètre la taille de la fenêtre de la FFT réduite à un quart de la taille de la fenêtre d'entrée.

```
ne10_mdct_i32_c::ne10_mdct_i32_c()
{
    // intialize the twiddling factors
    double alpha = M_PI / (8.0*MDCT_M);
    double omega = M_PI / MDCT_M;
    double scale = sqrt(sqrt(2.0 / static_cast<double>MDCT_M));
    for (int i = 0; i < MDCT_M2; ++i)
    {
        double x = omega * i + alpha;
        twiddle[2*i] = static_cast<int16_t>(cos(x)*scale*pow(2.0, 15.0));
        twiddle[2*i+1] = static_cast<int16_t>(sin(x)*scale*pow(2.0, 15.0));
    }

    // initialize the Ne10 FFT configuration
    cfg = ne10_fft_alloc_c2c_int32_c(MDCT_M2);
}
```

K.3 Destructeur

Destructeur de la classe `mdct_ne10_i32_c` qui permet de libérer la mémoire allouée à la configuration de la FFT avec la fonction appropriée de la librairie *Ne10*.

```
ne10_mdct_i32_c::~ne10_mdct_i32_c()
{
    ne10_fft_destroy_c2c_int32(cfg);
}
```

K.4 Fonction MDCT

Implémentation de l'algorithme de MDCT basé sur la FFT de la librairie *Ne10* en *integer 32* et en *plain C* :

- Initialisation du tableau d'entrée de la FFT : les opérations de *pre-twiddling* permettant de réduire la fenêtre d'entrée de la FFT sont faites en algorithmique *fixed point* ;
- Appel de la fonction FFT de *Ne10* ;
- Calcul du spectre de fréquences : les opérations de *post-twiddling* permettant de calculer le spectre à partir des données de sortie de la FFT et des facteurs de *twiddle* sont faites en algorithmique *fixed point*.

```
void ne10_mdct_i32_c::mdct(int16_t *time_signal, int32_t *spectrum)
{
    // pre-twiddling
    // fft_in = (Q1.15 + Q1.15) * Q1.15/4 + (Q1.15 + Q1.15) * Q1.15/4
    //          1/4 Q1.30 + 1/4 Q1.30 + 1/4 Q1.30 + 1/4 Q1.30 -> Q1.30
    //          >>7 -> Q1.23 + 8 bits reserved for the FFT
    int16_t *cos_tw = twiddle;
    int16_t *sin_tw = cos_tw + 1;
    for (int i = 0; i < MDCT_M2; i += 2)
    {
        int32_t r0 = static_cast<int32_t>(time_signal[MDCT_M32-1-i]) + time_signal[MDCT_M32+i];
        int32_t i0 = static_cast<int32_t>(time_signal[MDCT_M2+i]) - time_signal[MDCT_M2-1-i];

        int16_t c = cos_tw[i];
        int16_t s = sin_tw[i];

        fft_in[i/2].r = (((r0*c)+64)>>7) + (((i0*s)+64)>>7);
        fft_in[i/2].i = (((i0*c)+64)>>7) - (((r0*s)+64)>>7);
    }

    for (int i = MDCT_M2; i < MDCT_M; i += 2)
    {
        int32_t r0 = static_cast<int32_t>(time_signal[MDCT_M32-1-i]) - time_signal[-MDCT_M2+i];
        int32_t i0 = static_cast<int32_t>(time_signal[MDCT_M2+i]) + time_signal[MDCT_M52-1-i];

        int16_t c = cos_tw[i];
        int16_t s = sin_tw[i];

        fft_in[i/2].r = (((r0*c)+64)>>7) + (((i0*s)+64)>>7);
        fft_in[i/2].i = (((i0*c)+64)>>7) - (((r0*s)+64)>>7);
    }

    // perform the FFT
    ne10_fft_c2c_1d_int32_c(fft_out, fft_in, cfg, 0, 0);
```

```
    // post-twiddling
    // spectrum = Q9.23>>8 * Q1.15/4 + Q9.23>>8 * Q1.15/4
    //          = Q9.15 * Q1.15/4 + Q9.15 * Q1.15/4
    //          = Q10.30/4 + Q10.30/4
    //          = Q11.30/4
    //          = Q9.30 >> 15 = Q9.15
    for (int i = 0; i < MDCT_M; i += 2)
    {
        int32_t r0 = fft_out[i/2].r;
        int32_t i0 = fft_out[i/2].i;

        int16_t c = cos_tw[i];
        int16_t s = sin_tw[i];

        spectrum[i] = (((-(static_cast<int64_t>(r0)+128)>>8)*c+16384)>>15)
            - ((((static_cast<int64_t>(i0)+128)>>8)*s+16384)>>15);
        spectrum[MDCT_M-1-i] = ((((-static_cast<int64_t>(r0)+128)>>8)*s+16384)>>15)
            + ((((static_cast<int64_t>(i0)+128)>>8)*c+16384)>>15);
    }
}
```

L Implémentation de la MDCT basée sur la FFT de *Ne10* en *fixed point* avec optimisations NEON

Le code de cette annexe utilise les instructions intrinsèques NEON et doit être compilé avec l'option `-mfpu=neon`.

L.1 Header

Header de la classe `mdct_ne10_i32_neon` : MDCT basée sur la FFT de la librairie *Ne10* en *integer* (32 bits) optimisée par l'utilisation des opérations ARM NEON. La classe contient les structures de données `fft_in` en représentation Q1.15 et `fft_out` en Q9.15, les tableaux de facteurs de `twiddle` utilisés pour le *pre-* et le *post-processing* et la configuration de la FFT (`cfg`). Contrairement aux autres implémentations, les facteurs de twiddle ne sont pas rassemblés dans un seul tableau. Les tableaux de facteurs de *pre-twiddling* et de *post-twiddling* sont séparés car ils sont utilisés en 16 bits pour le *pre-twiddling* et en 32 bits pour le *post-twiddling*. Chacun de ces tableaux est séparé en deux afin que chaque moitié puisse être initialisée dans un ordre qui facilite l'utilisation des opérations SIMD. L'implémentation des fonctions de ce header est présentée dans les annexes suivantes.

```
#pragma once

#include <arm_neon.h>
#include "mdct_constants.h"
#include "NE10.h"

class ne10_mdct_i32_neon
{
private:
    ne10_fft_cfg_int32_t cfg;                                                   // Ne10 configuration
    ne10_fft_cpx_int32_t fft_in[MDCT_M2] __attribute__((aligned(16)));          // Ne10 FFT input buffer
    ne10_fft_cpx_int32_t fft_out[MDCT_M2] __attribute__((aligned(16)));         // Ne10 FFT output buffer
    int16_t pretwiddle_start[MDCT_M2] __attribute__((aligned(16)));             // pre-twiddle factors
    int16_t pretwiddle_end[MDCT_M2] __attribute__((aligned(16)));               // second half is stored
                                                                                //   in reversed order
    int32_t posttwiddle_start[MDCT_M2] __attribute__((aligned(16)));            // post-twiddle factors
    int32_t posttwiddle_end[MDCT_M2] __attribute__((aligned(16)));              // second half is stored
                                                                                //   in reversed order

public:
    ne10_mdct_i32_neon();
    ~ne10_mdct_i32_neon();
    void mdct(int16_t *time_signal, int32_t *spectrum);
};
```

L.2 Constructeur

Initialisation de la MDCT dans le constructeur de la classe `mdct_ne10_i32_neon` :

- Les tableaux de `twiddle` sont initialisés en *double* puis convertis en *integer* (représentation Q15 en 16 bits pour le *pre-twiddling* et en 32 bits pour le *post-twiddling*) : la première moitié des tableaux est rangée à l'endroit dans les tableaux `pretwiddle_start` et `posttwiddle_start` tandis que la seconde est rangée à l'envers dans les tableaux `pretwiddle_end` et `posttwiddle_end` ;
- La configuration de la FFT de *Ne10* est initialisée en *complex to complex* en *integer 32* avec en paramètre la taille de la fenêtre de la FFT réduite à un quart de la taille de la fenêtre d'entrée avec la fonction adaptée pour l'exécution d'une FFT optimisée avec les instructions ARM NEON.

```
ne10_mdct_i32_neon::ne10_mdct_i32_neon()
{
    double alpha = M_PI / (8.0*MDCT_M);
    double omega = M_PI / MDCT_M;
    double scale = sqrt(sqrt(2.0 / static_cast<double>MDCT_M));
    for (int i = 0; i < MDCT_M4; ++i)
    {
        double start = omega * (i) + alpha;
        double end = omega * (i+MDCT_M4) + alpha;
        double cos_start = cos(start);
        double sin_start = sin(start);
        double cos_end = cos(end);
        double sin_end = sin(end);
        pretwiddle_start[2*i] = static_cast<int16_t>(cos_start*scale*pow(2.0, 15.0));
        pretwiddle_start[2*i+1] = static_cast<int16_t>(sin_start*scale*pow(2.0, 15.0));
        pretwiddle_end[MDCT_M2-2*i-2] = static_cast<int16_t>(cos_end*scale*pow(2.0, 15.0));
        pretwiddle_end[MDCT_M2-2*i-1] = static_cast<int16_t>(sin_end*scale*pow(2.0, 15.0));
        posttwiddle_start[2*i] = static_cast<int32_t>(cos_start*scale*pow(2.0, 31.0));
        posttwiddle_start[2*i+1] = static_cast<int32_t>(sin_start*scale*pow(2.0, 31.0));
        posttwiddle_end[MDCT_M2-2*i-2] = static_cast<int32_t>(cos_end*scale*pow(2.0, 31.0));
        posttwiddle_end[MDCT_M2-2*i-1] = static_cast<int32_t>(sin_end*scale*pow(2.0, 31.0));
    }

    cfg = ne10_fft_alloc_c2c_int32_neon(MDCT_M2);
}
```

L.3 Destructeur

Destructeur de la classe `mdct_ne10_i32_c` qui permet de libérer la mémoire allouée à la configuration de la FFT avec la fonction appropriée de la librairie *Ne10*.

```
ne10_mdct_i32_neon::~ne10_mdct_i32_neon()
{
    ne10_fft_destroy_c2c_int32(cfg);
}
```

L.4 Fonction MDCT

Implémentation de l'algorithme de MDCT basé sur la FFT de la librairie *Ne10* en *integer 32* avec utilisation des instructions ARM NEON :

- Initialisation du tableau d'entrée de la FFT : les opérations de *pre-twiddling* permettant de réduire la fenêtre d'entrée de la FFT sont faites en algorithmique *fixed point*. L'utilisation des fonctions SIMD permet d'effectuer quatre opérations en parallèle afin de réduire le temps d'exécution. Les facteurs de *pre-twiddling* codés sur 16 bits transforment le signal d'entrée codé sur *16 bits* en un tableau d'entrée de la FFT codé sur 32 bits ;
- Appel de la fonction FFT de *Ne10* optimisée par l'utilisation des instructions ARM NEON ;
- Calcul du spectre de fréquences : les opérations de *post-twiddling* permettant de calculer le spectre à partir des données de sortie de la FFT et des facteurs de *twiddle* sont faites en algorithmique *fixed point*. Les fonctions SIMD permettent d'effectuer deux ou quatre opérations en parallèle afin de réduire le temps d'exécution.

```
void ne10_mdct_i32_neon::mdct(int16_t *time_signal, int32_t *spectrum)
{
    // see the twiddling_loops.nlsx file for more details

    // for i from 0 to 254, step 2

    // r[  0 -> 127, pas 1] =  time_signal[ 767 ->  513, pas 2] * c[  0 -> 127, pas 1]
    //                       +  time_signal[ 768 -> 1022, pas 2] * c[  0 -> 127, pas 1]
    //                       +  time_signal[ 256 ->  510, pas 2] * s[  0 -> 127, pas 1]
    //                       + -time_signal[ 255 ->    1, pas 2] * s[  0 -> 127, pas 1]

    // i[  0 -> 127, pas 1] =  time_signal[ 256 ->  510, pas 2] * c[  0 -> 127, pas 1]
    //                       + -time_signal[ 255 ->    1, pas 2] * c[  0 -> 127, pas 1]
    //                       + -time_signal[ 767 ->  513, pas 2] * s[  0 -> 127, pas 1]
    //                       + -time_signal[ 768 -> 1022, pas 2] * s[  0 -> 127, pas 1]

    // fft_in[i/2].r = time_signal[M32-1-i]    * cos_tw[i] + time_signal[M32+i]     * cos_tw[i]
    //               + time_signal[M2+i]       * sin_tw[i] + (-time_signal[M2-1-i]) * sin_tw[i]

    // fft_in[i/2].i = time_signal[M2+i]       * cos_tw[i] + (-time_signal[M2-1-i]) * cos_tw[i]
    //               + (-time_signal[M32-1-i]) * sin_tw[i] + (-time_signal[M32+i])  * sin_tw[i]


    // for i from 510 to 256, step 2

    // r[255 -> 128, pas 1] =  time_signal[ 257 ->  511, pas 2] * c[255 -> 128, pas 1]
    //                       + -time_signal[ 254 ->    0, pas 2] * c[255 -> 128, pas 1]
    //                       +  time_signal[ 766 ->  512, pas 2] * s[255 -> 128, pas 1]
    //                       +  time_signal[ 769 -> 1023, pas 2] * s[255 -> 128, pas 1]

    // i[255 -> 128, pas 1] =  time_signal[ 766 ->  512, pas 2] * c[255 -> 128, pas 1]
    //                       +  time_signal[ 769 -> 1023, pas 2] * c[255 -> 128, pas 1]
    //                       + -time_signal[ 257 ->  511, pas 2] * s[255 -> 128, pas 1]
    //                       +  time_signal[ 254 ->    0, pas 2] * s[255 -> 128, pas 1]

    // fft_in[i/2].r = time_signal[M32-1-i]    * cos_tw[i] + (-time_signal[-M2+i]) * cos_tw[i]
    //               + time_signal[M2+i]       * sin_tw[i] + time_signal[M52-1-i]  * sin_tw[i]

    // fft_in[i/2].i = time_signal[M2+i]       * cos_tw[i] + time_signal[M52-1-i] * cos_tw[i]
    //               + (-time_signal[M32-1-i]) * sin_tw[i] + time_signal[-M2+i]   * sin_tw[i]

    for (int i = 0; i < MDCT_M2; i += 8)
    {
        // tx.val[0] -> odd indexes
        // tx.val[1] -> even indexes
        int16x4x2_t t1 = vld2_s16(time_signal+MDCT_M2+i);
        int16x4x2_t t2 = vld2_s16(time_signal+MDCT_M2-8-i);
        int16x4x2_t t3 = vld2_s16(time_signal+MDCT_M32+i);
        int16x4x2_t t4 = vld2_s16(time_signal+MDCT_M32-8-i);

        t2.val[0] = (int16x4_t)vrev64_s32((int32x2_t)vrev32_s16(t2.val[0]));
                                            // reverse the t2 even values: 0 2 4 6 -> 6 4 2 0
        t2.val[1] = (int16x4_t)vrev64_s32((int32x2_t)vrev32_s16(t2.val[1]));
                                            // reverse the t2 odd values : 1 3 5 7 -> 7 5 3 1
        t4.val[0] = (int16x4_t)vrev64_s32((int32x2_t)vrev32_s16(t4.val[0]));
                                            // reverse the t4 even values: 0 2 4 6 -> 6 4 2 0
        t4.val[1] = (int16x4_t)vrev64_s32((int32x2_t)vrev32_s16(t4.val[1]));
                                            // reverse the t4 odd values : 1 3 5 7 -> 7 5 3 1
```

```
        // x_tw.val[0] -> cos twiddle
        // x_tw.val[1] -> sin twiddle
        int16x4x2_t start_tw = vld2_s16(pretwiddle_start+i);
        int16x4x2_t end_tw = vld2_s16(pretwiddle_end+i);

        // start.val[0] -> real part
        // start.val[1] -> imaginary part
        int32x4x2_t start;
        start.val[0] = vshrq_n_s32(
            vaddq_s32(
                vaddq_s32(
                    vmull_s16(t4.val[1], start_tw.val[0]),
                    vmull_s16(t3.val[0], start_tw.val[0])),
                vaddq_s32(
                    vmull_s16(t1.val[0], start_tw.val[1]),
                    vmull_s16(vneg_s16(t2.val[1]), start_tw.val[1])))
                , 7);
        start.val[1] = vshrq_n_s32(
            vaddq_s32(
                vaddq_s32(
                    vmull_s16(t1.val[0], start_tw.val[0]),
                    vmull_s16(vneg_s16(t2.val[1]), start_tw.val[0])),
                vaddq_s32(
                    vmull_s16(vneg_s16(t4.val[1]), start_tw.val[1]),
                    vmull_s16(vneg_s16(t3.val[0]), start_tw.val[1])))
                , 7);

        // end.val[0] -> real part
        // end.val[1] -> imaginary part
        int32x4x2_t end;
        end.val[0] = vshrq_n_s32(
            vaddq_s32(
                vaddq_s32(
                    vmull_s16(t1.val[1], end_tw.val[0]),
                    vmull_s16(vneg_s16(t2.val[0]), end_tw.val[0])),
            vaddq_s32(
                vmull_s16(t4.val[0], end_tw.val[1]),
                vmull_s16(t3.val[1], end_tw.val[1])))
            , 7);
        end.val[1] = vshrq_n_s32(
            vaddq_s32(
                vaddq_s32(
                    vmull_s16(t4.val[0], end_tw.val[0]),
                    vmull_s16(t3.val[1], end_tw.val[0])),
                vaddq_s32(
                    vmull_s16(vneg_s16(t1.val[1]), end_tw.val[1]),
                    vmull_s16(t2.val[0], end_tw.val[1])))
            , 7);

        // reverse the end part
        end.val[0] = (int32x4_t)vrev64q_s32(end.val[0]);
        end.val[0] = vcombine_s32(vget_high_s32(end.val[0]), vget_low_s32(end.val[0]));
        end.val[1] = (int32x4_t)vrev64q_s32(end.val[1]);
        end.val[1] = vcombine_s32(vget_high_s32(end.val[1]), vget_low_s32(end.val[1]));

        // store the result
        vst2q_s32((int32_t *)fft_in+i, start);
        vst2q_s32((int32_t *)(fft_in+MDCT_M2-4-i/2), end);
```

```
    }

    // perform the FFT
    ne10_fft_c2c_1d_int32_neon(fft_out, fft_in, cfg, 0, 0);

    // post-twiddling
    for (int i = 0; i < MDCT_M2; i += 8)
    {
        // load the fft ouput and reverse the end part
        // fft_out_x.val[0] -> real part
        // fft_out_x.val[1] -> imaginary part
        int32x4x2_t fft_out_start = vld2q_s32((int32_t *)fft_out+i);
        int32x4x2_t fft_out_end = vld2q_s32((int32_t *)fft_out+MDCT_M-8-i);
        fft_out_end.val[0] = (int32x4_t)vrev64q_s32(fft_out_end.val[0]);
        fft_out_end.val[0] = vcombine_s32(vget_high_s32(fft_out_end.val[0]),
                                          vget_low_s32(fft_out_end.val[0]));
        fft_out_end.val[1] = (int32x4_t)vrev64q_s32(fft_out_end.val[1]);
        fft_out_end.val[1] = vcombine_s32(vget_high_s32(fft_out_end.val[1]),
                                          vget_low_s32(fft_out_end.val[1]));

        // load the twiddle factors
        // x_tw.val[0] -> cos twiddle
        // x_tw.val[1] -> sin twiddle
        int32x4x2_t start_tw = vld2q_s32(posttwiddle_start+i);
        int32x4x2_t end_tw = vld2q_s32(posttwiddle_end+i);

        int32x4x2_t spectrum_start;
        spectrum_start.val[0] = vshrq_n_s32(vaddq_s32(
            vqrdmulhq_s32(vnegq_s32(fft_out_start.val[0]), start_tw.val[0]),
            vqrdmulhq_s32(vnegq_s32(fft_out_start.val[1]), start_tw.val[1])), 8);
        spectrum_start.val[1] = vshrq_n_s32(vaddq_s32(
            vqrdmulhq_s32(vnegq_s32(fft_out_end.val[0]), end_tw.val[1]),
            vqrdmulhq_s32(fft_out_end.val[1], end_tw.val[0])), 8);

        int32x4x2_t spectrum_end;
        spectrum_end.val[0] = vshrq_n_s32(vaddq_s32(
            vqrdmulhq_s32(vnegq_s32(fft_out_end.val[0]), end_tw.val[0]),
            vqrdmulhq_s32(vnegq_s32(fft_out_end.val[1]), end_tw.val[1])), 8);
        spectrum_end.val[1] = vshrq_n_s32(vaddq_s32(
            vqrdmulhq_s32(vnegq_s32(fft_out_start.val[0]), start_tw.val[1]),
            vqrdmulhq_s32(fft_out_start.val[1], start_tw.val[0])), 8);

        spectrum_end.val[0] = (int32x4_t)vrev64q_s32(spectrum_end.val[0]);
        spectrum_end.val[0] = vcombine_s32(vget_high_s32(spectrum_end.val[0]),
                                           vget_low_s32(spectrum_end.val[0]));
        spectrum_end.val[1] = (int32x4_t)vrev64q_s32(spectrum_end.val[1]);
        spectrum_end.val[1] = vcombine_s32(vget_high_s32(spectrum_end.val[1]),
                                           vget_low_s32(spectrum_end.val[1]));

        // store the result
        vst2q_s32((int32_t *)spectrum+i, spectrum_start);
        vst2q_s32((int32_t *)(spectrum+MDCT_M-8-i), spectrum_end);
    }
}
```

M Validation des spectres de fréquences produits par les MDCT

M.1 Code source

Code permettant de générer les spectres de fréquences par différentes MDCT avec différents signaux d'entrée. Les données sont sorties dans un fichier CSV afin de pouvoir être exmploitées sous forme de graphique.

```
#include <fstream>
#include <iomanip>
#include <iostream>
#include <limits>
#include <cmath>
#include <cstring>

#include "sin_wave.h"
#include "ref_mdct.h"
#include "fftw3_mdct_f32.h"
#include "ne10_mdct_f32_c.h"
#include "ne10_mdct_i32_c.h"
#include "ne10_mdct_i32_neon.h"

#define FREQUENCY 440.0

template<typename T, size_t L>
void convert_to_absolute(T* input, T* output)
{
    for (size_t i = 0; i < L; i+= 2)
        output[i/2] = std::hypot(input[i], input[i+1]);
}

/**
 * @brief one algorithm to compare them all, prints the test data in a CSV file
 */
int main(void)
{
    srand(static_cast<unsigned>(time(0)));

    // generate the time signal (double)
    double time_f64[MDCT_WINDOW_LEN] __attribute__((aligned(16)));
    sin_float(time_f64, MDCT_WINDOW_LEN, 0.9, FREQUENCY, 0.0, FS);
    // generate the time signal (float)
    float time_f32[MDCT_WINDOW_LEN] __attribute__((aligned(16)));
    sin_float(time_f32, MDCT_WINDOW_LEN, 0.9, FREQUENCY, 0.0, FS);
    // generate the time signal (integer)
    int16_t time_i16[MDCT_WINDOW_LEN] __attribute__((aligned(16)));
    sin_int(time_i16, MDCT_WINDOW_LEN, 0.9, FREQUENCY, 0.0, FS);
    // generate time signal with multiple amplitudes (integer)
    double time_temp0[MDCT_WINDOW_LEN] __attribute__((aligned(16)));
    sin_float(time_temp0, MDCT_WINDOW_LEN, 1.0, FREQUENCY, 0.0, FS);
    double time_temp1[MDCT_WINDOW_LEN] __attribute__((aligned(16)));
    sin_float(time_temp1, MDCT_WINDOW_LEN, 0.5, (FREQUENCY*2.0), 0.0, FS);
    int16_t time_i16_multi_ampl[MDCT_WINDOW_LEN] __attribute__((aligned(16)));
    for (int i = 0; i < MDCT_WINDOW_LEN; ++i)
        time_i16_multi_ampl[i] = static_cast<int16_t>(
            (time_temp0[i] + time_temp1[i]) * 0.45 * pow(2.0, 15.0));
    // generate time signal with multiple frequencies (integer)
    sin_float(time_temp0, MDCT_WINDOW_LEN, 1.0, FREQUENCY, 0.0, FS);
    sin_float(time_temp1, MDCT_WINDOW_LEN, 1.0, (FREQUENCY*2.0), 0.0, FS);
```

```
    double time_temp2[MDCT_WINDOW_LEN] __attribute__((aligned(16)));
    sin_float(time_temp2, MDCT_WINDOW_LEN, 1.0, (FREQUENCY*10.0), 0.0, FS);
    double time_temp3[MDCT_WINDOW_LEN] __attribute__((aligned(16)));
    sin_float(time_temp3, MDCT_WINDOW_LEN, 1.0, (FREQUENCY*20.0), 0.0, FS);
    double time_temp4[MDCT_WINDOW_LEN] __attribute__((aligned(16)));
    sin_float(time_temp4, MDCT_WINDOW_LEN, 1.0, (FREQUENCY*40.0), 0.0, FS);
    int16_t time_i16_multi_freq[MDCT_WINDOW_LEN] __attribute__((aligned(16)));
    for (int i = 0; i < MDCT_WINDOW_LEN; ++i)
        time_i16_multi_freq[i] = static_cast<int16_t>(
            (time_temp0[i] + time_temp1[i] + time_temp2[i] + time_temp3[i] + time_temp4[i])
            * 0.18 * pow(2.0, 15.0));
    // generate random time signal (integer)
    int16_t time_i16_random[MDCT_WINDOW_LEN] __attribute__((aligned(16)));
    for (int i = 0; i < MDCT_WINDOW_LEN; ++i)
        time_i16_random[i] = std::numeric_limits<int16_t>::max() * 2 * rand() / RAND_MAX
            - std::numeric_limits<int16_t>::max();
    // generate windowed (MLT) time signal
    sin_float(time_temp0, MDCT_WINDOW_LEN, 0.9, FREQUENCY, 0.0, FS);
    int16_t time_i16_mlt[MDCT_WINDOW_LEN] __attribute__((aligned(16)));
    for (int i = 0; i < MDCT_WINDOW_LEN; ++i)
        time_i16_mlt[i] = static_cast<int16_t>(
            time_temp0[i] * M_PI / MDCT_WINDOW_LEN * (i + 0.5)
            * 0.18 * pow(2.0, 15.0));

    // perform the ref MDCT (double precision)
    double ref_spectrum_f64[MDCT_M] __attribute__((aligned(16)));
    ref_float_mdct(time_f64, ref_spectrum_f64);
    double ref_f64_abs[MDCT_M2] __attribute__((aligned(16)));
    convert_to_absolute<double, MDCT_M>(ref_spectrum_f64, ref_f64_abs);

    // perform the ref MDCT (single precision)
    float ref_spectrum_f32[MDCT_M] __attribute__((aligned(16)));
    ref_float_mdct(time_f32, ref_spectrum_f32);
    float ref_f32_abs[MDCT_M2] __attribute__((aligned(16)));
    convert_to_absolute<float, MDCT_M>(ref_spectrum_f32, ref_f32_abs);

    // perform the ref MDCT (integer)
    int32_t ref_spectrum_i32[MDCT_M] __attribute__((aligned(16)));
    ref_int_mdct(time_i16, ref_spectrum_i32);
    int32_t ref_i32_abs[MDCT_M2] __attribute__((aligned(16)));
    convert_to_absolute<int32_t, MDCT_M>(ref_spectrum_i32, ref_i32_abs);

    int32_t ref_spectrum_i32_multi_ampl[MDCT_M] __attribute__((aligned(16)));
    ref_int_mdct(time_i16_multi_ampl, ref_spectrum_i32_multi_ampl);
    int32_t ref_i32_multi_ampl_abs[MDCT_M2] __attribute__((aligned(16)));
    convert_to_absolute<int32_t, MDCT_M>(ref_spectrum_i32_multi_ampl, ref_i32_multi_ampl_abs);

    int32_t ref_spectrum_i32_multi_freq[MDCT_M] __attribute__((aligned(16)));
    ref_int_mdct(time_i16_multi_freq, ref_spectrum_i32_multi_freq);
    int32_t ref_i32_multi_freq_abs[MDCT_M2] __attribute__((aligned(16)));
    convert_to_absolute<int32_t, MDCT_M>(ref_spectrum_i32_multi_freq, ref_i32_multi_freq_abs);

    int32_t ref_spectrum_i32_random[MDCT_M] __attribute__((aligned(16)));
    ref_int_mdct(time_i16_random, ref_spectrum_i32_random);
    int32_t ref_i32_random_abs[MDCT_M2] __attribute__((aligned(16)));
    convert_to_absolute<int32_t, MDCT_M>(ref_spectrum_i32_random, ref_i32_random_abs);

    int32_t ref_spectrum_i32_mlt[MDCT_M] __attribute__((aligned(16)));
```

```
ref_int_mdct(time_i16_mlt, ref_spectrum_i32_mlt);
int32_t ref_i32_mlt_abs[MDCT_M2] __attribute__((aligned(16)));
convert_to_absolute<int32_t, MDCT_M>(ref_spectrum_i32_mlt, ref_i32_mlt_abs);

// perform the FFTW3 MDCT
float fftw_spectrum_f32[MDCT_M] __attribute__((aligned(16)));
fftw3_mdct_f32 fftw3_mdct;
fftw3_mdct.mdct(time_f32, fftw_spectrum_f32);
float fftw_f32_abs[MDCT_M] __attribute__((aligned(16)));
convert_to_absolute<float, MDCT_M>(fftw_spectrum_f32, fftw_f32_abs);

// scale the FFTW3 spectrum to Q15
int32_t fftw_i32_abs[MDCT_M2] __attribute__((aligned(16)));
for (int i = 0; i < MDCT_M2; ++i)
    fftw_i32_abs[i] = static_cast<int32_t>(fftw_f32_abs[i]*pow(2.0, 15.0));

// perform the Ne10 MDCT f32 c
float ne10_spectrum_f32[MDCT_M] __attribute__((aligned(16)));
ne10_mdct_f32_c ne10_f32;
ne10_f32.mdct(time_f32, ne10_spectrum_f32);
float ne10_f32_abs[MDCT_M2] __attribute__((aligned(16)));
convert_to_absolute<float, MDCT_M>(ne10_spectrum_f32, ne10_f32_abs);

// perform the Ne10 MDCT i32 c
int32_t ne10_spectrum_i32_c[MDCT_M] __attribute__((aligned(16)));
ne10_mdct_i32_c ne10_i32_c;
ne10_i32_c.mdct(time_i16, ne10_spectrum_i32_c);
int32_t ne10_i32_c_abs[MDCT_M2] __attribute__((aligned(16)));
convert_to_absolute<int32_t, MDCT_M>(ne10_spectrum_i32_c, ne10_i32_c_abs);

// perform the Ne10 MDCT i32 neon
ne10_mdct_i32_neon ne10_i32_neon;

int32_t ne10_spectrum_i32_neon[MDCT_M] __attribute__((aligned(16)));
ne10_i32_neon.mdct(time_i16, ne10_spectrum_i32_neon);
int32_t ne10_i32_neon_abs[MDCT_M2] __attribute__((aligned(16)));
convert_to_absolute<int32_t, MDCT_M>(ne10_spectrum_i32_neon, ne10_i32_neon_abs);

int32_t ne10_spectrum_i32_neon_multi_ampl[MDCT_M] __attribute__((aligned(16)));
ne10_i32_neon.mdct(time_i16_multi_ampl, ne10_spectrum_i32_neon_multi_ampl);
int32_t ne10_i32_neon_multi_ampl_abs[MDCT_M2] __attribute__((aligned(16)));
convert_to_absolute<int32_t, MDCT_M>(ne10_spectrum_i32_neon_multi_ampl,
    ne10_i32_neon_multi_ampl_abs);

int32_t ne10_spectrum_i32_neon_multi_freq[MDCT_M] __attribute__((aligned(16)));
ne10_i32_neon.mdct(time_i16_multi_freq, ne10_spectrum_i32_neon_multi_freq);
int32_t ne10_i32_neon_multi_freq_abs[MDCT_M2] __attribute__((aligned(16)));
convert_to_absolute<int32_t, MDCT_M>(ne10_spectrum_i32_neon_multi_freq,
    ne10_i32_neon_multi_freq_abs);

int32_t ne10_spectrum_i32_neon_random[MDCT_M] __attribute__((aligned(16)));
ne10_i32_neon.mdct(time_i16_random, ne10_spectrum_i32_neon_random);
int32_t ne10_i32_neon_random_abs[MDCT_M2] __attribute__((aligned(16)));
convert_to_absolute<int32_t, MDCT_M>(ne10_spectrum_i32_neon_random, ne10_i32_neon_random_abs);

int32_t ne10_spectrum_i32_neon_mlt[MDCT_M] __attribute__((aligned(16)));
ne10_i32_neon.mdct(time_i16_mlt, ne10_spectrum_i32_neon_mlt);
int32_t ne10_i32_neon_mlt_abs[MDCT_M2] __attribute__((aligned(16)));
```

```
    convert_to_absolute<int32_t, MDCT_M>(ne10_spectrum_i32_neon_mlt, ne10_i32_neon_mlt_abs);

    // print the results in CSV file
    std::ofstream csv_file;
    csv_file.open("mdct.csv");
    csv_file
        << "time (ms),signal (f64),signal (f32),signal (i16),"
        << "multi amplitudes signal,multi frequencies signal,"
        << "frequency (Hz),ref spectrum f64,ref spectrum f32,ref spectrum i32,"
        << "FFTW3 spectrum f32,FFTW3 spectrum i32 (scaled to q15),"
        << "Ne10 spectrum f32,Ne10 spectrum i32 c,Ne10 spectrum i32 neon,"
        << "ref spectrum multi amplitudes,Ne10 spectrum multi amplitudes,"
        << "ref spectrum multi frequencies,Ne10 spectrum multi frequencies,"
        << "ref spectrum random,Ne10 spectrum random,"
        << "ref spectrum with MLT,Ne10 spectrum with MLT,"
        << std::endl;

    for (int i = 0; i < MDCT_M2; ++i)
        csv_file
            << i*1000.0/FS << ","
            << time_f64[i] << ","
            << time_f32[i] << ","
            << time_i16[i] << ","
            << time_i16_multi_ampl[i] << ","
            << time_i16_multi_freq[i] << ","
            << (i+1.0)*FS/MDCT_M << ","
            << ref_f64_abs[i] << ","
            << ref_f32_abs[i] << ","
            << ref_i32_abs[i] << ","
            << fftw_f32_abs[i] << ","
            << fftw_i32_abs[i] << ","
            << ne10_f32_abs[i] << ","
            << ne10_i32_c_abs[i] << ","
            << ne10_i32_neon_abs[i] << ","
            << ref_i32_multi_ampl_abs[i] << ","
            << ne10_i32_neon_multi_ampl_abs[i] << ","
            << ref_i32_multi_freq_abs[i] << ","
            << ne10_i32_neon_multi_freq_abs[i] << ","
            << ref_i32_random_abs[i] << ","
            << ne10_i32_neon_random_abs[i] << ","
            << ref_i32_mlt_abs[i] << ","
            << ne10_i32_neon_mlt_abs[i] << ","
            << std::endl;

    for (int i = MDCT_M2; i < MDCT_WINDOW_LEN; ++i)
        csv_file
            << i*1000.0/FS << ","
            << time_f64[i] << ","
            << time_f32[i] << ","
            << time_i16[i] << ","
            << time_i16_multi_ampl[i] << ","
            << time_i16_multi_freq[i] << ","
            << std::endl;

    csv_file.close();

    return 0;
}
```

M.2 Compilation

Commandes CMake utilisées pour générer l'exécutable de test des MDCT.

```
# MDCTs to CSV file
add_executable(test_mdct test/validation/test_mdct.cpp
    src/sin_wave.cpp src/ref_mdct.cpp src/fftw3_mdct_f32.cpp
    src/ne10_mdct_f32_c.cpp src/ne10_mdct_i32_c.cpp src/ne10_mdct_i32_neon.cpp)
target_link_libraries(test_mdct NE10 fftw3f)
```

N Mesure des performances des MDCT *Ne10* et MDCT de référence

N.1 Code source

Code permettant de mesurer les performances des MDCT de *Ne10* en *float 32 plain C*, *integer 32 plain C*, *integer 32* avec optimisation NEON et de la MDCT de référence en *double* (en fonction de la variable de préprocesseur définie à la compilation). Le code mesure et affiche le temps d'exécution moyen et l'écart type. Ces informations sont calculées sur un nombre d'exécutions donné en paramètre à l'exécutable. Les MDCT sont testées avec le même signal sinusoïdal en entrée dont la valeur par défaut est de 200 Hz.

```
#include <iostream>
#include <cstring>

#include "args_parser.h"
#include "mdct_constants.h"
#include "sin_wave.h"
#include "Timers.h"


#ifdef FIXED_POINT_C     // fixed point arithmetic

#include "ne10_mdct_i32_c.h"

#define INPUT_DATA               int16_t
#define OUTPUT_DATA              int32_t
#define GENERATE_SIN             sin_int
#define MDCT                     ne10_mdct_i32_c

#elif FIXED_POINT_NEON   // fixed point arithmetic

#include "ne10_mdct_i32_neon.h"

#define INPUT_DATA               int16_t
#define OUTPUT_DATA              int32_t
#define GENERATE_SIN             sin_int
#define MDCT                     ne10_mdct_i32_neon

#elif FLOATING_POINT     // floating point arithmetic

#include "ne10_mdct_f32_c.h"

#define INPUT_DATA               float
#define OUTPUT_DATA              float
#define GENERATE_SIN             sin_float
#define MDCT                     ne10_mdct_f32_c

#else                    // reference algorithm in floating point arithmetic

#include "ref_mdct.h"

#define INPUT_DATA               double
#define OUTPUT_DATA              double
#define GENERATE_SIN             sin_float

#endif
```

```
/**
 * @brief Run the MDCT on a single frame x times
 * the signal is a single tone configurable via the --sin parameter (200Hz by default)
 * the number of runs is setted by the --run parameter (1 by default)
 */
int main(int argc, char **argv)
{
    // initialize the parameters
    params p;
    try
    {
        p = parse_args(argc, argv);
    }
    catch(const std::runtime_error &err)
    {
        std::cerr << err.what() << std::endl;
        usage();
        return 1;
    }

    // print which MDCT will be tested
#ifdef FIXED_POINT_C
    std::cout << "MDCT Ne10 i32 plain C: "
        << p.runs << " runs with a single tone signal (" << p.frequency << "Hz)" << std::endl;
#elif FIXED_POINT_NEON
    std::cout << "MDCT Ne10 i32 Neon: "
        << p.runs << " runs with a single tone signal (" << p.frequency << "Hz)" << std::endl;
#elif FLOATING_POINT
    std::cout << "MDCT Ne10 f32 plain C: "
        << p.runs << " runs with a single tone signal (" << p.frequency << "Hz)" << std::endl;
#else
    std::cout << "Reference MDCT: "
        << p.runs << " runs with a single tone signal (" << p.frequency << "Hz)" << std::endl;
#endif

    // generate the time signal
    INPUT_DATA time_signal[MDCT_WINDOW_LEN] __attribute__((aligned(16)));
    memset(&time_signal, 0, MDCT_WINDOW_LEN*sizeof(INPUT_DATA));
    GENERATE_SIN(time_signal, MDCT_WINDOW_LEN, 0.9, p.frequency, 0.0, FS);

    // initialize an empty spectrum
    OUTPUT_DATA mdct_spectrum[MDCT_M] __attribute__((aligned(16)));
    memset(&mdct_spectrum, 0, MDCT_M*sizeof(OUTPUT_DATA));

    // perform the MDCT x times
    EvsHwLGPL::CTimers timer;
    int64_t *runtimes = static_cast<int64_t *>(malloc(p.runs * sizeof(int64_t)));

#ifdef REF
    for (unsigned i = 0; i < p.runs; ++i) {
        timer.Start();
        ref_float_mdct<INPUT_DATA>(time_signal, mdct_spectrum);
        timer.Stop();
        runtimes[i] = timer.GetTimeElapsed();
    }
```

```
#else
    MDCT ne10_mdct;
    for (unsigned i = 0; i < p.runs; ++i) {
        timer.Start();
        ne10_mdct.mdct(time_signal, mdct_spectrum);
        timer.Stop();
        runtimes[i] = timer.GetTimeElapsed();
    }
#endif

    // compute the average run time
    double avg = 0.0;
    for (unsigned i = 0; i < p.runs; ++i) avg += static_cast<double>(runtimes[i]);
    avg = avg / static_cast<double>(p.runs);
    std::cout << "average run time: " << avg << " ns" << std::endl;

    // compute the standard deviation
    double dev = 0.0;
    for (unsigned i = 0; i < p.runs; ++i) dev += static_cast<double>(runtimes[i]) - avg;
    dev = dev * dev / static_cast<double>(p.runs);
    dev = sqrt(dev);
    std::cout << "standard deviation: " << dev << std::endl;

    // clean
    free(runtimes);

    return 0;
}
```

N.2 Compilation

Commandes CMake pour la compilation des différents exécutables de tests de performance des MDCT :

- la variable `FLOATING_POINT` permet de compiler l'exécutable `run_mdct_f32_c` pour tester la MDCT basée sur la FFT de *Ne10* en *float 32 plain C* ;
- la variable `FIXED_POINT_C` permet de compiler l'exécutable `run_mdct_i32_c` pour tester la MDCT basée sur la FFT de *Ne10* en *integer 32 plain C* ;
- la variable `FIXED_POINT_NEON` permet de compiler l'exécutable `run_mdct_i32_neon` pour tester la MDCT basée sur la FFT de *Ne10* en *integer 32* avec optimisations NEON ;
- la variable `REF` permet de compiler l'exécutable `run_mdct_ref` pour tester la MDCT de référence en *double* ;

```
# Ne10 f32 x times on a single frame
add_executable(run_ne10_mdct_f32_c test/performance/run_ne10_mdct.cpp
    src/args_parser src/sin_wave.cpp src/Timers.cpp
    src/ne10_mdct_f32_c.cpp)
target_compile_definitions(run_ne10_mdct_f32_c PUBLIC -DFLOATING_POINT)
target_link_libraries(run_ne10_mdct_f32_c NE10)

# Ne10 i32 plain C x times on a single frame
add_executable(run_ne10_mdct_i32_c test/performance/run_ne10_mdct.cpp
    src/args_parser src/sin_wave.cpp src/Timers.cpp
    src/ne10_mdct_i32_c.cpp)
target_compile_definitions(run_ne10_mdct_i32_c PUBLIC -DFIXED_POINT_C)
target_link_libraries(run_ne10_mdct_i32_c NE10)
```

```
# Ne10 i32 neon x times on a single frame
add_executable(run_ne10_mdct_i32_neon test/performance/run_ne10_mdct.cpp
    src/args_parser src/sin_wave.cpp src/Timers.cpp
    src/ne10_mdct_i32_neon.cpp)
target_compile_definitions(run_ne10_mdct_i32_neon PUBLIC -DFIXED_POINT_NEON)
target_link_libraries(run_ne10_mdct_i32_neon NE10)

# Ref MDCT x times on a single frame
add_executable(run_ref_mdct test/performance/run_ne10_mdct.cpp
    src/args_parser src/sin_wave.cpp src/Timers.cpp
    src/ref_mdct.cpp)
target_compile_definitions(run_ref_mdct PUBLIC -DREF)
target_link_libraries(run_ref_mdct NE10)
```

O Mesure des performances de la MDCT *FFTW3* en *float 32*

O.1 Code source

Code permettant de mesurer les performances de la MDCT basée sur la FFT de *FFTW3* en *float 32*. Le code mesure et affiche le temps d'exécution moyen et l'écart type. Ces informations sont calculées sur 10 000 000 d'exécutions. La MDCT est testée avec le même signal sinusoïdal en entrée dont la valeur est définie à 440 Hz.

```
#include <iostream>
#include <cstring>

#include "fftw3_mdct_f32.h"
#include "sin_wave.h"
#include "Timers.h"

#define RUNS         10000000
#define FREQUENCY    440.0

int main()
{
    // print which MDCT will be tested
    std::cout << "MDCT FFTW3 f32 plain C: "
        << RUNS << " runs with a single tone signal (" << FREQUENCY << "Hz)" << std::endl;

    // generate the time signal
    float time_signal[MDCT_WINDOW_LEN];
    sin_float(time_signal, MDCT_WINDOW_LEN, 0.9, FREQUENCY, 0.0, FS);

    // initialize an empty spectrum
    float spectrum[MDCT_M];
    memset(spectrum, 0, MDCT_M*sizeof(float));

    // initialize the configuration
    fftw3_mdct_f32 fftw3_mdct;

    // perform the MDCT
    EvsHwLGPL::CTimers timer;
    int64_t *runtimes = static_cast<int64_t *>(malloc(RUNS * sizeof(int64_t)));
    for (int i = 0; i < RUNS; ++i)
    {
        timer.Start();
        fftw3_mdct.mdct(time_signal, spectrum);
        timer.Stop();
        runtimes[i] = timer.GetTimeElapsed();
    }

    // compute the average run time
    double avg = 0.0;
    for (int i = 0; i < RUNS; ++i) avg += static_cast<double>(runtimes[i]);
    avg = avg / static_cast<double>(RUNS);
    std::cout << "average run time: " << avg << " ns" << std::endl;

    // compute the standard deviation
    double dev = 0.0;
    for (int i = 0; i < RUNS; ++i) dev += static_cast<double>(runtimes[i]) - avg;
    dev = dev * dev / static_cast<double>(RUNS);
    dev = sqrt(dev);
    std::cout << "standard deviation: " << dev << std::endl;
```

```
    // clean
    free(runtimes);

    return 0;
}
```

O.2 Compilation

Commandes CMake pour la compilation de l'exécutable de tests de performance de la MDCT basée sur la FFT de *FFTW3* en *float 32*.

```
# FFTW3 f32
add_executable(run_fftw3_mdct_f32 test/performance/run_fftw3_mdct_f32.cpp
    src/fftw3_mdct_f32.cpp src/sin_wave.cpp src/Timers.cpp)
target_link_libraries(run_fftw3_mdct_f32 fftw3f)
```