Optimisation de chaînes de traitement de signaux numériques radiofréquences et application à une cascade de filtres

Arthur Hugeat

UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ

UNIVERSITÉ DE FRANCHE-COMTÉ

THÈSE DE DOCTORAT DE L'ÉTABLISSEMENT UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

PRÉPARÉE À L'UNIVERSITÉ DE FRANCHE-COMTÉ

École doctorale n°37

Sciences Pour l'Ingénieur et Microtechniques

Doctorat d'Informatique

par

ARTHUR HUGEAT

Optimisation de chaînes de traitement de signaux numériques radiofréquences et application à une cascade de filtres

Thèse présentée et soutenue à Besançon, le 9 décembre 2019

Composition du Jury :

PÉTROT FRÉDÉRIC	Professeur, Institut Polytechnique de Grenoble TIMA	Examinateur
RISSET TANGUY	Professeur, Insa-Lyon Département Télécommunications, Services et Usages	Rapporteur
CASSEAU EMMANUEL	Professeur, Université de Rennes INRIA	Rapporteur
FRIEDT JEAN-MICHEL	Maître de Conférence HDR, Université de Franche-Comté FEMTO-ST Temps-Fréquence	Directeur de thèse
BERNARD JULIEN	Maître de Conférence, Université de Franche-Comté FEMTO-ST DISC	Co-encadrant de thèse
BOURGEOIS PIERRE-YVES	Chargé de Recherche CNRS FEMTO-ST Temps-Fréquence	Co-encadrant de thèse

N° | X | X | X |

Remerciements

Je tiens à remercier tout d'abord Emmanuel Casseau et Tanguy Risset d'avoir accepté d'être mes rapporteurs de thèse. Je veux remercier également Frédéric Pétrot pour avoir accepté d'être examinateur et président du jury. Enfin, je remercie tous les membres du jury d'avoir bravé les grèves SNCF et d'avoir fait le déplacement jusqu'à Besançon pour assister à la soutenance.

Ensuite, j'aimerais remercier Jean-Michel Friedt pour m'avoir, dès la licence, pris en stage et fait découvrir les problématiques de recherche liées au temps-fréquence. Tout au long des cinq années où j'ai eu le plaisir de travailler avec lui, il a été patient et pédagogue ce qui m'a permis à l'issu de ma thèse d'avoir des bases en traitement du signal. Je remercie également Gwen Goavec-Mérou et Pierre-Yves Bourgeois qui m'ont été d'un grand secours pour toute la partie développement sur FPGA et sur le traitement numérique.

Je voudrais aussi remercier Julien Bernard que je connais depuis la licence. Il m'a soutenu tout au long de mes études : il a su me conseiller quant à mon projet professionnel et il m'a été d'un grand secours lors de ma thèse en m'apportant son expertise. Il m'a également soutenu dans mes engagements associatifs que ce soit pour organiser et participer à la Nuit de l'Info ou à la Global Game Jam mais aussi lors des séances de Dead Pixel Society. C'est aussi lui qui m'a formé à ce merveilleux langage qu'est le C++.

Je souhaite aussi remercier l'ensemble des mes collègues doctorants sans qui ma thèse n'aurait pas été aussi agréable. Je remercie tout tout d'abord Florent Marie pour les longues heures qu'on a passé à jouer et à faire des pauses au lieu de travailler. Je remercie aussi Lisa Corbat pour nos échanges constructifs sur les chats et les jeux de société. Je remercie également Henri de Bouteray et Jean-Philippe Gros pour les débats et les fou-rires qu'on a pu avoir au RU ou aux croissants du mercredi. Enfin, je remercie Jessy Colonval et Paul Breugno pour m'avoir supporté comme collègue de bureau pendant mon ATER.

Je remercie également ma famille et mes proches qui m'ont toujours encouragé à continuer et plus particulièrement ma grand-mère qui a eu le courage de relire mon manuscrit pour m'éviter de me ridiculiser vis à vis de la grammaire et de l'orthographe.

Je remercie également le conseil régional de Franche-Comté pour avoir financé ma thèse.

Introduction

Le traitement du signal est un domaine incontournable de l'électronique qui a pour but d'étudier les techniques de traitement, d'analyse et d'interprétation des signaux. Le but de ces traitements peut être de transmettre des informations (signaux radiofréquences), de connaître avec précision toutes les composantes physiques du signal (horloge atomique), ou de caractériser des appareils (métrologie des signaux). Afin d'illustrer un domaine d'application de cette discipline, prenons le cas du traitement de signaux radiofréquences, définis dans notre cas comme les signaux entre quelques dizaines de kilohertz à quelques centaines de gigahertz.

Les signaux radiofréquences sont couramment utilisés : stations de radio FM, GPS, téléphonie mobile, Wifi... Le problème est que tous ces signaux partagent le même medium lors de leur transmission. Il faut donc pouvoir partager l'ensemble de la bande spectrale pour que tous ces moyens de communication puissent être utilisés en même temps. Il existe quatre méthodes de multiplexage : temporel, spatial, par codage et fréquentiel. Le dernier consiste à transposer l'information sur une plage de fréquences non utilisée.

Pour illustrer ce fonctionnement, prenons le cas de la station radio. Son objectif est de transmettre un flux audio à ses auditeurs : ce flux est caractérisé par sa bande passante (la largeur de l'étalement spectral). Pour y arriver, elle code ce flux avec une fréquence porteuse créant ainsi une bande radiofréquence. L'auditeur quant à lui, reçoit cette bande qui contient le message et il doit donc décoder le flux audio avant de pouvoir l'entendre. Cette étape consiste à ramener la bande radiofréquence en bande de base (la bande passante est centrée autour de 0 Hz).

L'étage de réception est donc conçu pour supprimer la fréquence porteuse pour ne garder que l'information transmise. La première étape des traitements radiofréquences est la suppression, en analogique, de cette fréquence. Toutefois, les oscillateurs locaux de l'émetteur et du récepteur n'étant pas localisés au même endroit et présentant donc des fréquences légèrement différentes, la suppression n'est pas parfaite. Le mélangeur chargé de transposer le signal reçu en bande de base n'est lui non plus pas parfait et des composantes spectrales du signal porteur viennent parasiter l'information reçue par fuite de l'oscillateur local. L'architecture superhétérodyne fut développée pour pallier ce problème. Dans cette architecture, un filtre de fréquence intermédiaire a été introduite afin d'éliminer les éventuels artefacts dus aux fuites de l'oscillateur local lors du mélange avec le signal reçu.

La principale limitation de cette architecture est due à ce filtre supplémentaire. En

effet, la bande passante analysable sera directement liée à la bande passante que le filtre est capable de traiter. Dans le cadre de la réception d'un signal FM, cette architecture va bien fonctionner mais avec les techniques plus modernes de transmission de l'information (comme par exemple le saut de phases) plus gourmandes en ressources spectrales (saut de fréquence), elle ne sera pas capable de traiter toute la bande passante.

Avec l'avènement du numérique est apparu le traitement numérique du signal et plus particulièrement la radio logicielle. Cette nouvelle discipline garde le même objectif que son pendant analogique mais les traitements sont effectués numériquement. Cela permet donc d'avoir des comportements prédictibles et stables dans le temps, alors qu'en analogique les composants s'usent ou voient leurs propriétés varier avec l'environnement, ce qui dégrade leurs performances. Toutefois, les traitements numériques sont limités par leur capacité de traitement. En effet, si on reprend l'exemple précédent, pour analyser une bande radio, il faudrait être capable de traiter l'intégralité du spectre fréquentiel ce qui serait trop coûteux en ressources de calcul. L'approche classique est donc de garder la première partie du traitement en analogique (typiquement la suppression de la fréquence porteuse) puis de traiter la suite en radio logicielle (en numérique). Dans cette thèse, nous nous intéresserons au cas des bandes de fréquences plus modestes (sub-50 MHz) permettant un échantillonnage direct du signal radiofréquence en vue d'une chaîne de traitement totalement numérique.

Dans le contexte de la métrologie du signal, c'est à dire l'étude des signaux ultra-stables (issus d'horloges atomiques par exemple), une des caractéristiques analysées est le bruit du signal. Le bruit peut provenir de nombreuses sources : du signal en lui même, des traitements analogiques, des traitements numériques... Et il aura un impact certain sur le signal et notamment sur la phase, c'est ce qu'on appelle le bruit de phase. Pour pouvoir le quantifier, on doit faire un traitement particulier qu'on appelle banc de bruit de phase.

Le principe de ce traitement est d'analyser l'ensemble des composantes spectrales du signal. Pour y arriver, il est impératif de réduire le débit du flux en le décimant (en ne gardant qu'un point à intervalles réguliers). Le problème d'une décimation est que cela va induire un phénomène appelé le repliement spectral. Il s'agit d'un principe physique qui montre que lors d'une décimation, les composantes hautes fréquences vont être ramenées en bande de base. Pour éviter cela il est impératif d'utiliser des filtres qui vont couper les composantes hautes fréquences du signal (des filtres passe-bas). Encore une fois la conception du filtre sera le facteur limitant du traitement car c'est sa capacité de traitement qui déterminera la bande passante du traitement.

Pour étudier le bruit de phase, il est également impératif de pouvoir traiter en temps réel tous les échantillons du signal. Pour ce faire, il faut des composants capables de traiter des flux de données de l'ordre du million voir du milliard d'échantillons à la seconde. Un ordinateur équipé de processeur générique (CPU) n'est pas capable de traiter un tel débit de données. C'est pourquoi, il faut avoir recours à des outils massivement parallèles qui sont capables de traiter toutes les données qui arrivent au fur et à mesure en effectuant les différents traitements en même temps.

Il existe plusieurs outils répondant à ces spécifications mais dans le cadre de nos travaux nous nous sommes sommes tout particulièrement intéressés aux *Field-Programmable Gate Array* (FPGA). Il s'agit d'une puce électronique qui est programmable à l'échelle de la porte logique, ce qui signifie qu'il est possible de les configurer pour qu'ils puissent travailler comme le ferait un circuit intégré dédié (*Application-Specific Integrated Circuit*, ASIC). Cependant un FPGA, contrairement aux ASIC, est reconfigurable. Il est donc possible de les configurer pour un traitement puis de le réutiliser plus tard pour faire un autre traitement. Les FPGA sont donc couramment utilisés pour faire des prototypages. De plus certains FPGA disposent d'un CPU exécutant un système d'exploitation (OS) avec lequel il peut communiquer. Il est donc tout à fait possible de faire des calculs dans le FPGA puis d'utiliser le CPU pour post-traiter les données ou bien, si le traitement est trop lourd, d'envoyer les données sur d'autres machines qui s'en chargeront.

Le temps de développement pour un design sur FPGA peut être long car on est très proche du comportement matériel et il est donc compliqué de pouvoir trouver les erreurs d'une implémentation ou de prendre en compte tous les effets de bord possibles. Afin d'accélérer le temps de conception, de nombreuses techniques ont été mises au point. Notre attention fut particulièrement attirée par la méthodologie des Skeletons. Cette approche consiste à décomposer les chaînes de traitement en une succession de tâches élémentaires et paramétrables. Lors de la création d'un design pour FPGA, on crée toutes les tâches nécessaires puis on les ajoute dans un catalogue. Ainsi, lors d'un prochain design on pourra réutiliser au besoin des blocs déjà prêts à l'emploi et les adapter à de nouveaux besoins.

L'autre avantage d'une telle méthode est de pouvoir automatiser la création d'une chaîne. En effet, certains chercheurs ont rajouté des descriptions pour chacun des blocs et grâce à un logiciel spécialisé, ils sont capables d'obtenir une chaîne de traitement optimisée. Dans la majorité des travaux, ces optimisations concernent des aspects matériels (par exemple, un placement optimal des composants dans le FPGA pour éviter d'avoir des chemins trop longs entre les tâches).

Le but de cette thèse est de proposer une méthodologie permettant d'optimiser des chaînes de traitements dans leur ensemble, spécifique aux FPGA, en se basant sur des Skeletons. On souhaite toutefois avoir une approche beaucoup plus haut niveau nous permettant d'utiliser des contributions du monde informatique. Afin de démontrer que notre méthodologie est efficace, nous allons l'appliquer au problème classique de conception de filtre et plus particulièrement à la conception de cascade de filtres.

Le principe de la méthodologie est la suivante. La première étape est de modéliser le problème par une abstraction complète, nous détachant complètement des préoccupations matérielles. Cette abstraction est nécessaire pour avoir suffisamment de degrés de liberté dans le choix des réponses mais il faut également veiller à ne pas être trop éloigné de la réalité car l'objectif est d'obtenir un résultat exploitable matériellement.

La deuxième étape est d'utiliser des outils issus de la recherche opérationnelle pour trouver des solutions optimales à partir de la modélisation. La recherche opérationnelle est une discipline informatique qui se consacre à l'étude et au développement d'outils et

de techniques capables de trouver des solutions optimales dans des ensembles de solutions possibles (espace de recherche). On a recours à ces méthodes quand les problèmes sont complexes à résoudre par des programmes plus classiques (les problèmes sont dit NP, NP-difficile ou NP-complet). Dans notre cas, nous utiliserons un programme quadratique pour traduire notre modèle en un ensemble de contraintes (taille limitée, réjection minimale, taille des données...) et en définissant une fonction objectif qui nous permettra d'exprimer ce qu'on cherche à maximiser ou minimiser. Puis en utilisant un logiciel spécialisé dans la résolution d'un tel programme, nous obtiendrons une solution optimale.

La troisième étape consiste à traduire la solution optimale précédemment obtenue en un design de FPGA fonctionnel. Ici nous nous baserons sur l'écosystème OscImpDigital qui nous fournit le catalogue de skeletons utilisés pour créer la chaîne de traitement. Nous travaillerons avec une carte RedPitaya qui embarque un FPGA Zynq-7010 SoC de Xilinx et un processeur ARM Cortex A9 ce qui nous permet d'avoir un OS Linux embarqué sur la carte. Cette carte est l'exemple parfait car la puce FPGA ne dispose pas de beaucoup de ressources, donc nous devrons optimiser suffisamment les designs pour qu'ils marchent malgré des ressources limitées.

Pour finir la quatrième et dernière étape consiste à générer des données expérimentales puis de les post-traiter afin de nous assurer que les résultats obtenus sont bien ceux attendus (donnée par la solution optimale de la deuxième étape). Si les résultats sont incohérents, on repart à la première étape et on adapte le modèle pour éviter que les mauvais résultats ne se reproduisent. Et si au contraire les résultats coïncident avec ceux attendus, alors notre modèle est correct et les résultats expérimentaux obtenus sont donc validés.

Ce manuscrit de thèse s'articule autour de quatre parties. La partie I sera constituée du chapitre 1 dans lequel nous reviendrons plus en détails sur les aspects traitement du signal et recherche opérationnelle. De plus dans ce chapitre nous dresserons l'état de l'art de l'optimisation au sens large d'un traitement dans les FPGA.

La partie II sera composée de deux chapitres. Le chapitre 2 revient sur la présentation de notre méthodologie et détaillera la première étape de celle-ci en décrivant pas à pas la création du modèle abstrait. Le chapitre 3 se focalisera sur la deuxième étape de notre méthodologie et il présentera la création du programme quadratique utilisé pour trouver la solution optimale du modèle abstrait.

La partie III sera composée du chapitre 4. Dans ce chapitre, nous présenterons les deux dernière étapes de notre méthodologie : la génération d'un chaîne de traitement et l'analyse des résultats. Nous donnerons également deux adaptations possibles du modèle et leurs conséquences sur les résultats.

Pour finir la partie IV contiendra le chapitre 5 dans lequel nous donnerons les conclusions de nos résultats ainsi que les perspectives possibles à l'issue de cette thèse.

Table des matières

Première partie

Contexte et Problématiques

Chapitre 1

Contexte

Cette thèse s'inscrit dans la thématique du traitement du signal. Cette science est un domaine vaste qui nécessite de nombreuses connaissances physiques pour bien comprendre nos travaux et leurs impacts. De plus, nous allons travailler dans le domaine numérique, il sera également indispensable de présenter les notions utilisées dans le traitement de l'information.

Ce premier chapitre a donc pour but d'essayer de présenter et d'expliquer le plus clairement possible tous les prérequis nécessaires à notre réflexion.

Dans la section 1.1, nous abordons la thématique du traitement du signal numérique et plus spécifiquement le cas de la radio logicielle. Nous verrons également ce que sont les *Field-Programmable Gate Arrays* (FPGA) et pourquoi ils sont tant prisés dans les traitements numériques. Pour finir nous détaillerons des méthodologies de travail, issues de la littérature scientifique, qui sont utilisées pour manipuler les FPGA.

Dans la section suivante 1.2 nous définirons ce qu'est la métrologie des signaux. Nous en profiterons également pour définir tous les termes physiques utilisés dans la suite de cette thèse.

La section 1.3, quant à elle, sera consacrée à la présentation d'une étape clé dans le traitement du signal : le filtrage des données. Nos travaux sont principalement concentrés sur ce point, nous présenterons donc en quoi consiste un filtrage et pourquoi cette étape est cruciale. Nous parlerons également des nombreuses contributions scientifiques consacrées à l'amélioration d'algorithmes, aux méthodes de conception des filtres ou aux techniques d'utilisation de ces filtres.

La dernière section 1.4 présente le domaine de la recherche opérationnelle. Comme nous le verrons, la question de l'optimisation d'un filtre est complexe. Nous allons devoir recourir à des outils performants pour trouver des solutions optimales, nous ferons de la recherche opérationnelle. Dans cette partie, nous présenterons donc différentes techniques et leurs applications dans d'autres contextes que le nôtre.

1.1 Traitement de signaux numériques à haut débit

Pour traiter un signal, nous pouvons utiliser des composants analogiques ou des composants numériques. Les composants analogiques ont l'avantage d'être au plus proche de la physique des signaux mais ils n'ont pas une stabilité inconditionnelle dans le temps et ils imposent donc de concevoir un nouveau circuit dédié à chaque application. À l'inverse, les composants numériques présentent des différences avec le comportement d'un signal continu qu'ils discrétisent mais ils resteront stables et fiables dans le temps. De plus, les différences observables sont connues et elles peuvent donc être éventuellement compensées ou à défaut prises en compte lors de l'interprétation des résultats.

Afin de gérer la numérisation au plus près du signal radiofréquence en éliminant un maximum d'étages analogiques, sources de bruit et limitant la flexibilité du matériel, un enjeu majeur dans le traitement de signaux numériques est la capacité à traiter en temps réel des signaux à haut débit. En effet, traiter en temps réel un signal à plusieurs MHz voir à plusieurs GHz nécessite des composants spécifiques capables d'absorber des dizaines de millions d'échantillons à la seconde.

Dans la section 1.1.1 nous verrons comment est exploité le traitement numérique dans le cadre de la radio logicielle et dans la section 1.1.2 nous parlerons plus en détail de composants capables de traiter des signaux à haut débit.

1.1.1 Architectures de radio logicielle

Depuis l'avènement du télégraphe, la multiplication des informations transmises sur un medium de transfert donné (câble électrique, fibre optique, spectre radiofréquence...) impose de multiplexer la ressource. Quatre modes de multiplexage sont identifiés : temporel, spatial, par codage et fréquentiel. Le dernier impose une transposition de l'information vers une bande spectrale inoccupée : il s'agit de coder l'information en faisant varier la fréquence (modulation) de la porteuse. À l'autre bout de la chaîne de communication, le récepteur désire restituer l'information en éliminant la transposition intermédiaire de fréquence : il s'agit de la démodulation. La transposition de fréquence passe nécessairement par un composant non-linéaire, puisque seul $\exp(j\omega_1 t) \times \exp(j\omega_2 t)$ permet de produire la somme des fréquences $\exp(j(\omega_1 + \omega_2)t)$: il s'agit du mélangeur. Ce mélangeur est imparfait est donne lieu à des fuites de l'oscillateur local du récepteur qui handicape la capacité à extraire l'information lors de la démodulation. Ce constat a poussé, suite aux liaisons radiofréquences initiées depuis la fin du XIX^{e} siècle, à l'avènement de récepteurs selon l'architecture superhétérodyne . La figure 1.1 présente cette architecture.

La figure 1.2 donne la définition des différents termes utilisés pour décrire le fonctionnement de cette architecture et, plus généralement, lors d'un traitement d'un signal radio-fréquence.

Dans cette architecture de récepteur, le signal modulé est transposé en fréquence de la bande radiofréquence vers la bande de base par mélange avec l'oscillateur local

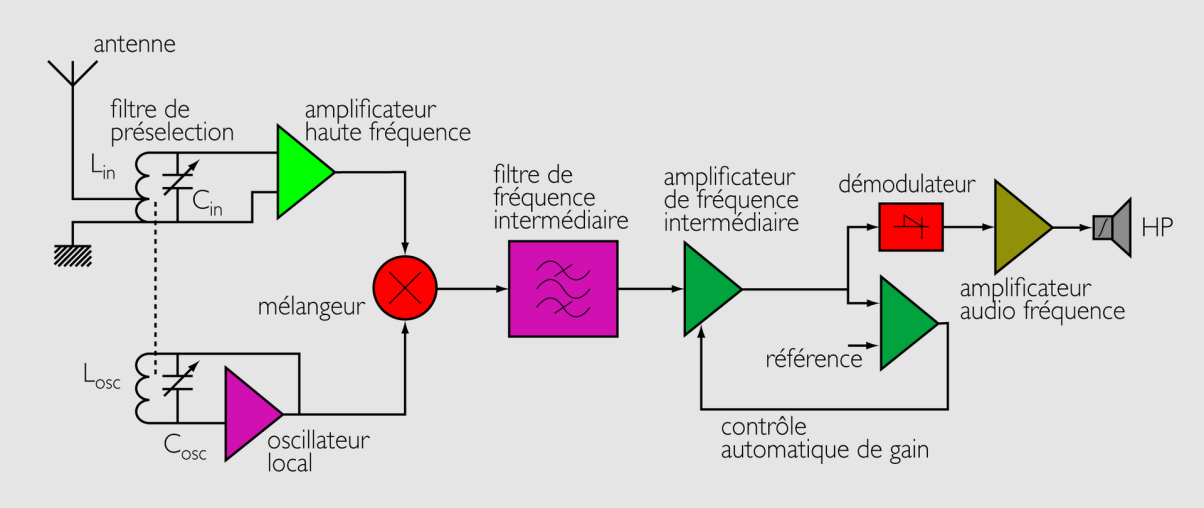


FIGURE 1.1 – Schéma de l'architecture superhétérodyne (source : Gerben Hoeksma [46])

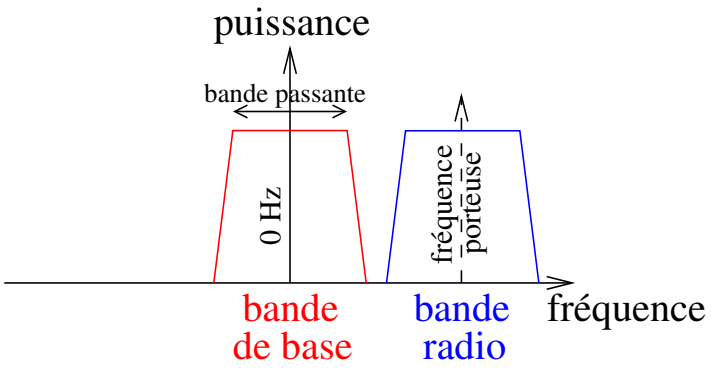


FIGURE 1.2 – Définition des termes utilisés pour décrire les divers signaux d'un spectre radiofréquence.

qui détermine la bande de fréquence analysée. La bande passante détermine la quantité d'information transmise. Afin de pallier les imperfection du mélangeur, la transposition de fréquence se fait en passant par une fréquence intermédiaire beaucoup plus basse que la porteuse et pour laquelle les composants seront plus performants. Par ailleurs, le filtre intermédiaire élimine les composantes spectrales induites par l'imperfection du mélangeur radiofréquence, mais limite aussi la flexibilité du système : la bande passante analysable par le démodulateur est limitée par la bande passante du filtre de fréquence intermédiaire. En effet, quand la largeur de la bande passante (encombrement spectral) est faible, le filtre est capable de traiter l'intégralité de l'information. Cependant, dans des méthodes de transmission plus modernes (le saut de phases par exemple), il est nécessaire de traiter des bandes passantes plus larges et par conséquent le filtre de fréquence intermédiaire devient limitant.

La flexibilité ultime de la radio logicielle consisterait à échantillonner la bande radiofréquence et à effectuer tous ces traitements analogiques en numérique. Les oscilloscopes radiofréquences à plusieurs GHz de bande passante permettent d'appréhender une telle approche, mais le débit de communication entre les convertisseurs analogique numériques

et l'unité de traitement numérique ne permet pas une communication continue. Acceptable pour des applications de traitement de signaux discontinus tels que les radars mais une telle limitation ne l'est pas pour un traitement en continu tel que nécessaire pour une communication analogique ou numérique.

L'évolution des composants et l'amélioration de l'isolation des mélangeurs permettent aujourd'hui de proposer une solution intermédiaire, dite sans fréquence intermédiaire (0-IF) dans laquelle le signal d'antenne amplifié est transposé de la bande radiofréquence vers la bande de base et échantillonné. De ce fait, la fréquence d'échantillonnage n'est plus imposée comme le double de la porteuse mais comme la bande passante du signal. De tels débits peuvent rester significatifs : à titre d'exemple, les 80 MHz de la bande Industrielle, Scientifique et Médicale (ISM) autour de 2440 MHz induisent un débit de 640 MB/s pour transmettre des données complexes codées par deux nombres flottants simple précision. Nous devons donc appréhender le traitement de tels débits de données numériquement en flux tendu pour répondre aux attentes de la radio logicielle.

Il est donc indispensable d'avoir des composants qui peuvent garantir la contrainte de temps réel (latence bornée) et qui soient capables de traiter toutes les données à temps. Une première famille de composants répondant à ces besoin sont les *Application-Specific Integrated Circuit* (ASIC). Il s'agit de circuits intégrés créés et paramétrés pour effectuer une action particulière. L'inconvénient majeur de ce type de circuits est qu'ils ne sont pas reconfigurables une fois créés et qu'ils sont coûteux à produire en phase de prototypage. Une alternative aux ASIC est d'utiliser des circuits reprogrammables tels que les FPGA : ces matrices de portes logiques reconfigurables sont en effet l'étape préliminaire à tout développement d'ASIC numérique, en fournissant la flexibilité de reconfigurer par logiciel l'agencement des liaisons entre blocs de traitements numériques.

1.1.2 Utilisation de FPGA

La technologie des *Field-Programmable Gate Arrays* (FPGA) est née dans les années 1980 et n'a cessé de se développer. Le principe fondamental d'un FPGA est de travailler à l'échelle de la porte logique.

Les FPGA sont reprogrammables, ils sont donc réutilisables dans d'autres situations et s'adaptent plus facilement aux besoins de nouveaux projets, contrairement aux ASIC qui ne peuvent être modifiés et obligent à concevoir un nouveau circuit à chaque changement.

En 1984, Altera conçoit son premier modèle de circuits reprogrammables (l'EP300) suivi l'année suivante par Xilinx avec son premier modèle de FPGA (le XC2064). Les années qui suivent voient la puissance des FPGA grandement augmenter. C'est dans les années 1990 que les FPGA commencent à se répandre dans les télécommunications et dans les réseaux informatiques, puis, vers la fin de la décennie, dans le monde industriel.

C'est au XXI^{e} siècle que les FPGA gagnent en notoriété avec l'arrivée des premiers modèles de FPGA ayant un CPU en parallèle. Xilinx propose, par exemple, en haut de gamme la série de Virtex *System on Chip* (SoC) qui inclut des processeurs PowerPC. En

combinant sur la même puce un FPGA et un CPU, la communication entre les deux est simplifiée et le débit est plus important. Plus tard, sera produite la gamme des Zynq-7000 all Programmable SoC, qui inclut un double-cœur ARM. La puissance des CPU varie en fonction du modèle, il s'agit souvent d'un processeur ARM Cortex-A9 double-cœur cadencé à 1 GHz. Une architecture alternative nous est présentée par les puces Nios II, MicroBlaze ou encore Micro32 (respectivement d'Altera, Xilinx et Lattice Semiconductor) qui, quant à elles, simulent le CPU directement dans le FPGA ce qui réduit la quantité de ressources disponibles de ces FPGA.

Chaque constructeur de FPGA a sa propre façon de concevoir le fonctionnement matériel de ses puces. Dans le cadre de cette thèse, nous avons surtout travaillé avec la série des Zynq-7000.

Chez Xilinx, ces puces sont constituées d'un premier ensemble de ressources : les *Configurable Logic Blocks* (CLB). Selon les modèles elles peuvent être constituées de registres, de *Look-Up Tables* (LUT) et de *Flip-Flops* (FF). Ce sont ces CLB qui constituent essentiellement le cœur de la partie reconfigurable du FPGA. Autre caractéristique, chaque CLB est cadencé grâce à une horloge interne à la puce : toutes les opérations des CLB sont donc effectuées à chaque changement d'état de l'horloge (ou coup d'horloge) selon le principe que tout circuit numérique doit être synchrone pour être déterministe.

Outre les CLB, dans le cadre du traitement de signaux numériques, on a recours à des *Digital Signal Processors* (DSP, à ne pas confondre avec Digital Signal Processing). Ces DSP permettent de faire des opérations spécifiques plus simplement et plus efficacement que si elles avaient été programmées avec des CLB. On peut citer l'exemple des multiplieurs qui permettent de faire une multiplication entre deux nombres entiers codés sur 18 bits en consommant moins de ressources qu'une implémentation équivalente avec des CLB.

Une autre ressource intéressante pour notre étude sont les Block RAMs (BRAM). Ce sont des RAMs accessibles très rapidement pour respecter la contrainte de lecture ou d'écriture d'une donnée en un seul coup d'horloge. Bien souvent ces blocs sont répartis dans la puce de manière à être facilement accessibles au besoin.

De par leur nature et leurs propriétés intrinsèques, les FPGA sont des solutions idéales lorsqu'on a des exigences en termes de débit et/ou de bande passante. En effet, bien que la fréquence de l'horloge d'un FPGA ne s'étale que de 125 MHz à 250 MHz, ce sont ses propriétés sur le parallélisme qui lui permettent de gérer un flux important de données. On pourrait penser qu'un FPGA cadencé à 250 MHz est quatre fois moins rapide qu'un CPU cadencé à 1 GHz, mais la réalité est un peu plus complexe. Pour chaque opération, le CPU va faire appel à une unité arithmétique et logique (en anglais *Arithmetic–Logic Unit*, ALU) qui va aller récupérer l'instruction, la décoder puis l'exécuter. Au contraire, un FPGA est massivement parallèle, il est donc capable de faire toutes ces opérations en même temps.

Par conséquent, même si l'opération est faite moins rapidement, il est possible de faire des pipelines pour chaque donnée entrante et ainsi de pouvoir absorber un flux de données important. De plus, la contrainte que chaque donnée soit traitée en un coup

d'horloge nous garantit que le traitement est fait en temps réel, contrairement à un CPU qui peut-être dépendant de l'ordonnancement du système d'exploitation qui lui, n'est pas toujours en temps réel.

Cependant, malgré leur reconfigurabilité et leur rapidité d'exécution, il est nécessaire d'avoir des bonnes connaissances en électronique pour exploiter pleinement les capacités d'un FPGA. En effet pour créer un traitement sous FPGA, il faut écrire le traitement en Verilog ou en *VHSIC Hardware Description Language* (VHDL). Ces deux langages sont très proches du matériel et il est donc nécessaire d'avoir une connaissance approfondie des circuits logiques pour pouvoir obtenir un résultat satisfaisant. De plus, le debugage du design est plus long car il y a des contraintes matérielles qui ne sont pas forcément prises en compte par les outils de synthèse et de simulation fournis par les fabricants des puces.

À l'heure actuelle, les FPGA sont couramment utilisés dans de nombreux domaines :
— Traitement audio / vidéo (reconnaissance de voix ou de formes par exemple) [44, 58, 82, 83, 66, 92, 97, 36, 5, 2, 56].
— Imagerie médicale [48, 51, 47]
— Résonance magnétique nucléaire [88, 79, 98, 17]
— Tomographie par rayons X [23]
— Métrologie des oscillateurs [84, 87, 24]

Dès 2010, Microsoft dans son projet de *Catapult* [18] et *Brainwave* [12] a intégré des FPGA dans ses serveurs afin d'accélérer leur réactivité et de permettre à leur cloud Azure de passer à l'échelle plus facilement et de mieux s'adapter à l'extension du cloud. De son coté en 2016, Amazon a mis en preview des machines virtuelles équipées de FPGA Xilinx Ultrascale Plus pour permettre à ses clients d'utiliser la puissance des cartes sans être obligé d'acheter le matériel.

Dans le cadre de cette recherche, nous utilisons les FPGA pour pré-traiter les données afin que les contraintes (débits, bande-passante...) soient suffisamment faibles pour pouvoir faire le reste du traitement en radio-logicielle (*Software-Defined Radio*, SDR).

1.1.3 Méthodologie de conception

Les FPGA sont appréciés pour leur flexibilité et leur performance. Afin d'en tirer le meilleur profit, de nombreux scientifiques ont proposé des méthodologies pour travailler efficacement avec les FPGA.

Dans [19], Cole présente le concept de *Algorithmic Skeleton*. L'idée est de séparer la structure et les opérations. Autrement dit, il sépare la partie algorithmique de la partie de communication d'une tâche. Initialement appliquée au domaine des machines parallèles, cette logique peut-être adaptée au domaine des FPGA.

En effet, un FPGA peut être considéré comme une machine parallèle puisqu'il est possible de faire les traitements simultanément. Chacun de ces traitements peut donc être considérée comme un nœud d'une machine parallèle.

Dans [29], Dittmann montre comment il a adapté ce concept aux FPGA. Il explique qu'en séparant l'algorithme de la communication, on est capable d'avoir une abstraction supérieure lors de la création d'un design de FPGA. En effet, il est possible d'écrire un code optimisé sans avoir à se soucier des détails matériels de la carte sur laquelle on travaille. De plus, il est possible de *templatiser* le code pour le rendre générique et ainsi plus facilement réutilisable dans d'autres designs.

Ce gain d'abstraction permet, selon Dittmann, de tirer pleinement profit de la reconfiguration partielle des FPGA sans complexifier l'écriture du code. De plus le code étant indépendant du matériel, on peut aussi l'adapter plus facilement à une autre puce FPGA. On peut même facilement changer de constructeur sans être obligé revoir complètement la tâche : il suffit d'adapter la partie communication pour que le composant reste valable.

Ainsi en gardant une bibliothèque de tous les composants génériques, il devient beaucoup plus facile de concevoir rapidement des nouveaux designs et le niveau d'abstraction supplémentaire permet également de faire des analyses poussées de la chaîne de traitement.

Benkrid *et al.* proposent eux aussi une adaptation des skeletons dans le cadre du traitement d'images et de vidéos. Ils démontrent dans [10] qu'un traitement d'images peut être abstrait sous la forme d'un DAG *(Directed Acyclic Graph)*. La définition d'un DAG est donnée plus tard dans la section 1.4. Les tâches sans prédécesseur seraient les entrées et celles sans successeurs les sorties. Chacun des nœuds représente une tâche élémentaire et au fur et à mesure du parcours du graphe, on enchaîne les traitements jusqu'à avoir l'image traitée complètement.

Les auteurs mettent en avant cette abstraction en précisant qu'ainsi il est plus simple de pouvoir analyser finement le traitement effectué sans se soucier des détails matériels de l'implémentation. Ils ont également développé toute une infrastructure, nommée HIDE, pour permettre l'analyse et la génération du design pour le FPGA.

De plus ils émettent l'hypothèse que cette analyse puisse être automatisée mais qu'ils doivent pour l'instant la faire manuellement.

Ce n'est que dans [8, 9] que Benkrid *et al.* proposent une évolution de leur infrastructure HIDE. En effet, dans cet article, les auteurs décrivent comment HIDE fonctionne et comment l'utiliser pour optimiser une chaîne de traitement. Ce nouveau langage leur permet de décrire finement la structure des tâches qu'ils ont implémentées : comment sont organisées les données traitées, où sont situées les entrées/sorties... Avec ces informations, leur écosystème est capable de placer et de synthétiser automatiquement des designs FPGA.

Avec cette logique, Goavec-Merou propose un outil, COGEN [40], capable lui aussi de placer et d'optimiser les différentes tâches de la chaîne de traitement afin d'obtenir une consommation minimale ou d'obtenir la meilleur latence. Cet outil se focalise sur le choix de la meilleure configuration possible en tenant compte des contraintes de traitement des blocs. En effet, il faut s'assurer que les tâches puissent traiter toutes les données à temps : si on doit traiter un échantillon par coup d'horloge alors que la tâche ne peux

traiter qu'une donnée tous les deux cycles le traitement ne fonctionnera pas.

COGEN est capable également de gérer des traitements faits en parallèle qui se rejoignent plus tard (par exemple pour calculer une corrélation). L'intérêt de l'outil est donc de pouvoir donner la chaîne de traitement et de le laisser choisir les paramètres pour que toutes les contraintes soient respectées. L'autre avantage de COGEN est qu'il s'appuie sur POD, un utilitaire en ligne de commande fourni par Armadeus pour automatiser la création de design pour leur FPGA. Il donc possible d'instrumentaliser la création de traitement pour ces cartes.

Toujours en restant à un haut niveau d'abstraction, une autre méthodologie est largement étudiée : High-Level Synthesis (HLS). Il s'agit ici de laisser la possibilité aux développeurs d'écrire leur programme avec un langage de programmation classique (tel que le C par exemple) et d'utiliser un logiciel pour traduire ce code source en un design FPGA.

Nane *et al.* ont publié en 2015 un état de l'art sur le sujet [69]. Ils nous apprennent qu'il existe de nombreux outils pour faire du HLS. Ils peuvent être rangés dans deux catégories :

1. Domain Specific Language (DSL) : ces outils seront basés sur de nouveaux langages ou sur des langages existants adaptés à des besoins spécifiques
2. General-purpose Programmable Language (GPL) : ces outils seront eux basés sur des langages existants (C, C++, Java, SystemC...)

Dans cet article, l'accent est mis notamment sur trois outils issus de la recherche académique :

— DWARV [68] est un compilateur HLS développé par Delft University of Technology. Il est basé sur l'architecture du compilateur commercial CoSy [32].
— Bambu [76] est un compilateur HLS développé à Politecnico di Milano. Il se base sur le compilateur open-source GCC [77] et profite des nombreuses fonctionnalités proposées par ce dernier.
— LegUp [13] est un compilateur développé à l'Université de Toronto.

Ces trois compilateurs ont été comparés grâce à un benchmark mais les auteurs soulignent le fait qu'il est très difficile d'être parfaitement objectif compte-tenu du fait que les différents outils n'ont pas les mêmes objectifs ni ne visent les mêmes architectures. Cependant la conclusion donnée dans l'article nous apprend que les compilateurs sont globalement aussi performants et aucun d'entre eux n'est meilleur dans tous les domaines. Tous ont leurs spécialités et leurs faiblesses.

Il existe également des catégories de langages à mi-chemin des langages bas niveau tel que le Verilog ou le HDL et des langages HLS haut-niveau présenté précédemment. On peut notamment citer Chisel [4] ou encore SpinalHDL [1] qui est un fork de Chisel. Ces langages sont basés sur le langage Scala [72] qui est une surcouche au Java et ils visent à décrire la chaîne de traitement tout en laissant la possibilité soit de générer du codes C++ pour faire des simulations, soit de générer VHDL pour synthétiser le circuit. L'intérêt d'utiliser le Scala comme base est de pouvoir tirer profit des avantages d'un langage de programmation haut niveau tel que le polymorphisme et le paradigme orienté objet. Un autre avantage considérable des langages Chisel et SpinalHDL est de cacher

à l'utilisateur des considérations techniques complexes (gestion des signaux d'horloge et de reset, changement du domaine d'horloge...). L'utilisateur n'a donc qu'à écrire la suite d'opérations qu'il veut et ces langages vont se charger de traduire cela en VHDL ou en simulations C++.

Chisel est notamment utilisé dans la suite open-source rocketChip [3] réalisée par l'université de Berkeley, qui est l'implémentation de référence de l'*Instruction Set Architecture* (ISA) RISC-V [95].

Une autre approche similaire est présentée par Ettus Research avec RFNoC. En s'appuyant sur la carte USRP (qui embarque un frontend RF et un FPGA), il est possible d'écrire un script qui interroge directement des composants dans le FPGA. On se rapproche de l'idée de skeleton car on place des blocs de traitement génériques paramétrables et on les fait communiquer entre eux. La seul différence avec les outils précédents c'est que le design du FPGA n'est pas automatique, c'est à l'utilisateur de mettre dans son design tous les composants qu'il va utiliser pour que RFNoC puisse les configurer et les utiliser. L'autre limite de ce framework c'est qu'il est spécifique aux produits d'Ettus Research et qu'il n'est pas possible de l'utiliser avec d'autres FPGA.

RFNoC permet également de s'interfacer avec un logiciel couramment utilisé pour faire de la SDR : GNU Radio [35]. Il s'agit d'un logiciel qui représente un flux de traitement sous forme de blocs de traitement qu'on dispose selon les besoins et la chaîne ainsi créée sera exécutée par GNU Radio. Cet outil permet aussi l'intégration de tâches personnalisées écrites en Python ou C++. RFNoC tire profit de cette possibilité en fournissant à l'utilisateur des blocs de traitement qui sont capables de communiquer avec le FPGA.

1.2 Métrologie

Avant de détailler ce qu'est la métrologie d'un signal, il est primordial de bien définir ce qu'est un signal. Un signal issu d'un oscillateur parfait est une sinusoïde qui présente une période et une amplitude fixes comme le montre la figure 1.3. Le signal bleu a une période exacte de 2π et une amplitude de 1 en unité arbitraire, et le signal en rouge est lui aussi parfait mais il est légèrement décalé par rapport au bleu. L'écart entre la courbe bleu et la courbe rouge s'appelle la phase.

Cependant, les signaux ne sont jamais parfaits, ils sont toujours affectés de fluctuations plus ou moins fortes, c'est ce qu'on appelle du bruit. Ces variations sont présentées par la figure 1.4 qui a été emprunté dans le livre de Rubiola [81]. Le signal en bleu est encore un signal parfait mais le signal rouge est une version bruitée du signal parfait. On peut voir sur la figure qu'il y a deux types de bruit :

— le bruit de phase : il s'agit d'un écart temporel entre le signal réel et la sinusoïde
— le bruit en amplitude : il s'agit d'un écart de tension entre le signal parfait et le signal réel.

Le bruit d'un signal peut être lié à des paramètres intrinsèques au système (qualité

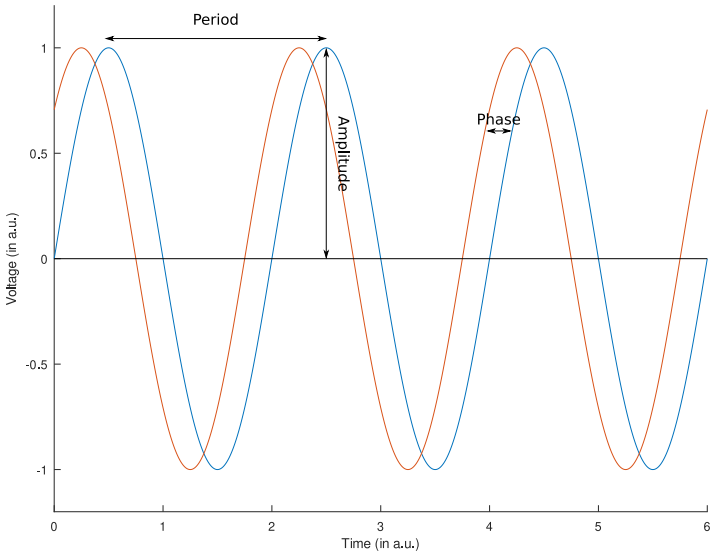


FIGURE 1.3 – Schéma de signaux parfaits

de l'oscillateur, parasitage de l'environnement...). Dans le cas du traitement numérique du signal, la première étape de tout traitement est d'échantillonner en temps discret et de quantifier le signal. Pour cela, on utilise un *Analog to Digital Converter* (ADC). Cependant cette étape est la source de quatre types de bruit [14] :

1. le bruit additif introduit par l'étage de mise en forme en entrée du convertisseur analogique-numérique *(input stage circuit)*
2. la gigue sur la date d'échantillonnage du signal analogique par le convertisseur *(aperture jitter)*
3. le bruit sur la tension de référence par rapport à laquelle la tension mesurée est comparée lors de la conversion analogique-numérique *(voltage reference)*
4. la quantification des données sur un nombre fini de bits.

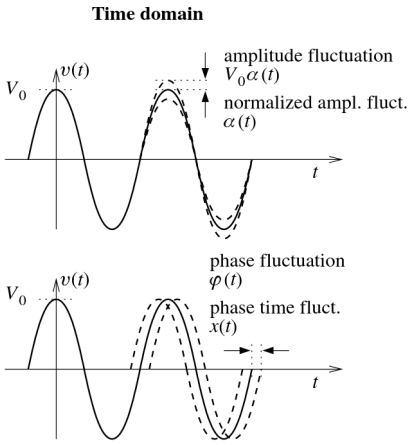


FIGURE 1.4 – Schémas explicatifs des bruits d'un signal (source : Enrico Rubiola [81])

Le problème rencontré lors de cette conversion est qu'un signal analogique est une variable continue en temps et en tension tandis que l'ADC est quantifié par le nombre de bits sur lesquels il peut coder le signal et échantillonné en temps discret. C'est pourquoi le choix et l'étude des ADC est primordial pour maîtriser et quantifier l'impact de ces discrétisations.

La quantification est la valeur physique mesurée lorsqu'on observe le signal et la discrétisation est l'intervalle de temps entre deux acquisitions. La figure 1.5 illustre ces deux notions.

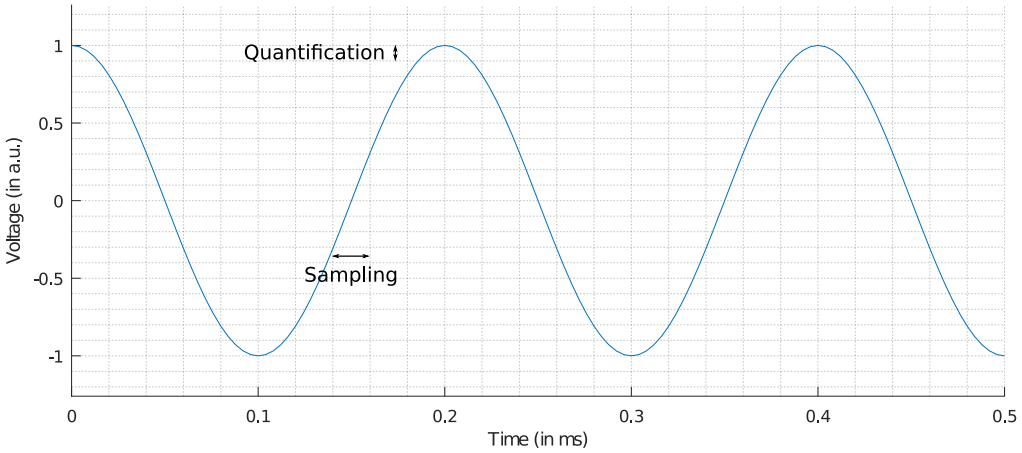
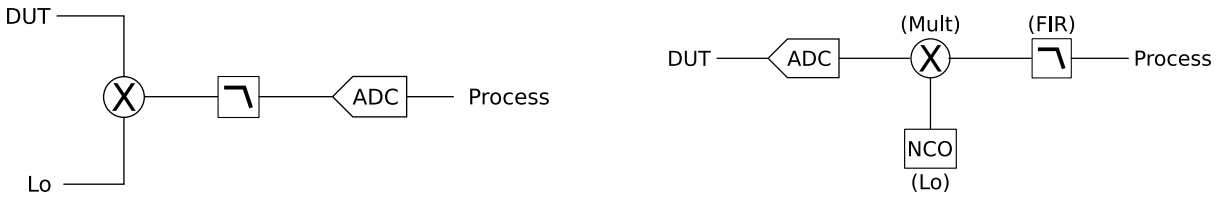


FIGURE 1.5 – Schéma explicatif de la quantification et de la discrétisation

Un autre exemple de bruit de quantification que l'on peut présenter ici se produit lors de la démodulation d'un signal. Lors de la mesure des fluctuations de phase d'un signal périodique, il faut supprimer la porteuse afin de n'observer que les fluctuations (bruit de phase). Pour ce faire, on transpose en fréquence le signal d'entrée en le multipliant par un autre signal à la même fréquence. On ramène ainsi le signal radiofréquence en bande de base (centrée sur 0 Hz) en vue de l'analyse de ses fluctuations. Dans sa version analogique, la transposition de fréquence par mélange sera affectée de deux sources de bruit, les fluctuations de l'oscillateur de référence et le bruit introduit par le mélangeur. Le pendant numérique de cette opération après échantillonnage du signal à analyser est l'imperfection de l'implémentation numérique de l'oscillateur local (*Numerically Controlled Oscillator*, NCO) et le produit entre le nombre fini de bits représentant le signal discrétisé et le nombre fini de bits sur lesquels est représenté le NCO. La maîtrise du plancher de bruit introduit par ces paramètres est critique pour déterminer la plus petite fluctuation de phase détectable par un tel traitement. Afin de rejeter les bruits haute fréquence et ne conserver que les composantes spectrales proches de la porteuse, un filtre passe-bas est placé après le mélangeur.

La figure 1.6 présente une démodulation en analogique (schéma 1.6a) et une démodulation en numérique (schéma 1.6b). Dans les deux cas, on cherche a récupérer l'information du signal (*Device Under Test*, DUT). On transpose le signal avec un oscillateur local (*Local Oscillator*, LO). Dans le cas du numérique, c'est le NCO qui simule le LO. L'origine des bruits n'est pas la même dans les deux cas. Dans le cas analogique, le LO n'étant



(a) Schéma d'une démodulation en analogique

(b) Schéma d'une démodulation en numérique

FIGURE 1.6 – Schéma comparatif entre la démodulation d'un signal en analogique et en numérique

pas parfait, il ajoute du bruit. De même pour le mixer, il n'est pas parfait et introduit du bruit. Dans le cas numérique, en revanche, les bruits sont dus aux approximations de calcul faites par le NCO, le multiplieur ou par le FIR.

Avant de chercher à évaluer des filtres, il est nécessaire d'expliquer ce qu'est un signal numérique et comment l'analyser. Un signal est une grandeur continue dans le temps et dans son unité (valeur). Il peut s'exprimer de façon bijective par ses composantes spectrales, elles aussi continues par transformée de Fourier. Le traitement numérique du signal impose de discrétiser l'axe des abscisses (échantillonnage) et des ordonnées (quantification). Une fois le signal discrétisé, sa décomposition spectrale devient elle aussi somme de composantes discrètes, chacune centrée sur une fréquence f_p exprimée en Hz avec une pondération propre à chaque composante telle que déterminée par l'opération de transformée de Fourier discrète. Quand on numérise un signal, on passe par un composant chargé de convertir le signal analogique en un signal numérique, c'est le rôle de l'ADC. L'une des principales caractéristiques d'un ADC est sa fréquence d'échantillonnage f_s : il s'agit du nombre de mesures générées par seconde. Selon le théorème de Shannon [73], on ne peut reconstruire le signal original de l'observation de ses composantes spectrales aux fréquences f que si $f < f_s/2$. On appelle cette limite la fréquence de Nyquist F_N.

Pour savoir quelles sont les composantes spectrales contenues dans un signal il faut calculer son spectre en passant dans la base orthonormale des fonctions trigonométriques par transformée de Fourier :

$$S(f) = \int_{-\infty}^{+\infty} S(t) e^{j2\pi ft} dt \tag{1.1}$$

Nous utilisons la transformée de Fourier quand nous travaillons dans le cas continu. Dans le cas numérique, nous devons utiliser sa version discrétisée :

$$S(n) = \sum_{k=0}^{N-1} S_k e^{j2\pi \frac{n}{N} k} \tag{1.2}$$

Dans la suite de nos travaux, nous serons toujours dans le cas discret et par abus de langage, nous appellerons *transformée de Fourier* la transformée de Fourier discrtète et son implémentation rapide, la transformée de Fourier rapide (FFT).

La figure 1.7 donne l'exemple d'un spectre d'un signal à 10 kHz et échantillonné à

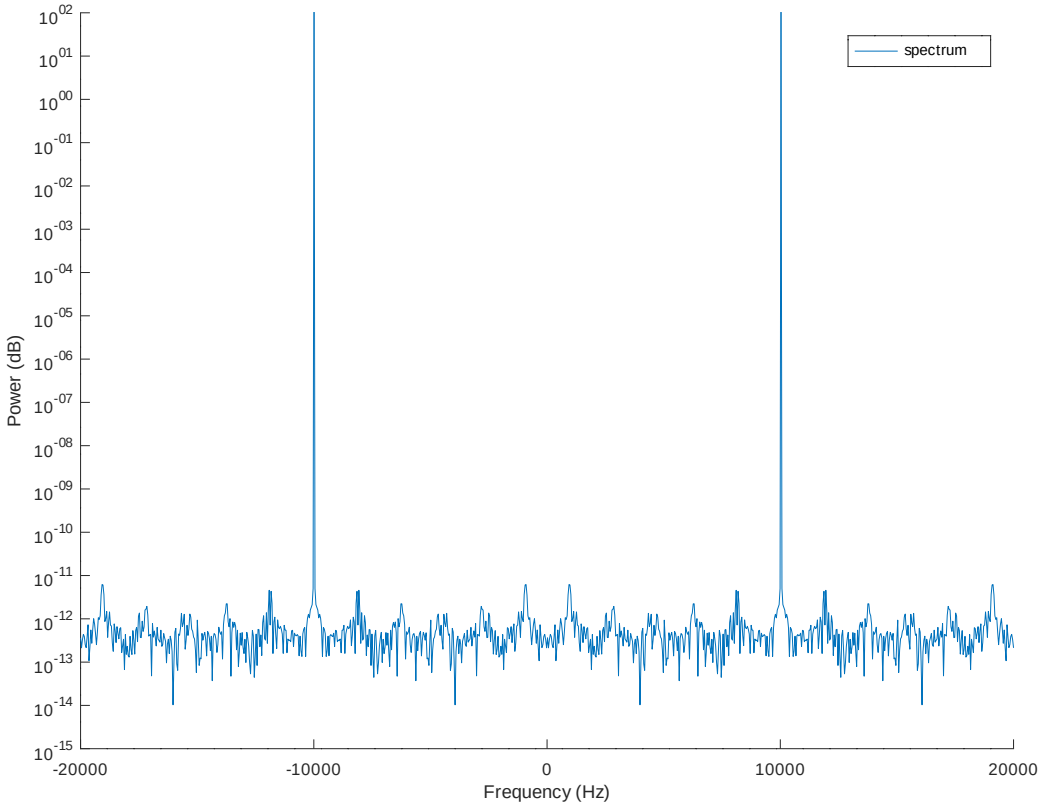


FIGURE 1.7 – Exemple du spectre d'un signal à 10 kHz et échantillonné à 40 kHz

40 kHz. Dans cet exemple on a : $F_N = 40/2 = 20$ kHz. La fréquence de Nyquist est donc suffisante pour observer la composante spectrale à 10 kHz (le pic bleu).

Un spectre discret s'étale de $-F_N$ à F_N. Dans le cas particulier d'un signal réel, la transformée de Fourier est hermitienne et son module est donc pair : dans ce cas, on n'affichera que la partie des fréquences positives du spectre pour éviter la redondance. Cependant, il n'est pas rare de travailler avec des signaux complexes dans le cadre de la SDR et donc de ne pas pouvoir faire cette simplification.

L'analyse du bruit de phase d'un signal périodique consiste à caractériser la distribution spectrale des fluctuations de phase et donc à prendre la transformée de Fourier de l'évolution de la phase du signal. Pour s'affranchir de la composante spectrale puissance de la porteuse, il faut transposer le signal afin de ramener ses fluctuations autour de 0 Hz. Une densité spectrale de bruit de phase est définie comme la puissance du bruit par intervalle de fréquence. Conserver un pas de fréquence constant tel que proposé par une transformée de Fourier sur n points pour obtenir une largeur de chaque composante spectrale $f_b = f_N/n$ induira une résolution inacceptable aux basses fréquences (initialement proches de la porteuse) et, en échelle logarithmique, une résolution variable par décade. Afin d'obtenir une résolution constante par décade, nous devrons décimer le signal afin que les transformées de Fourier successives sur n points dans chaque décade fournissent

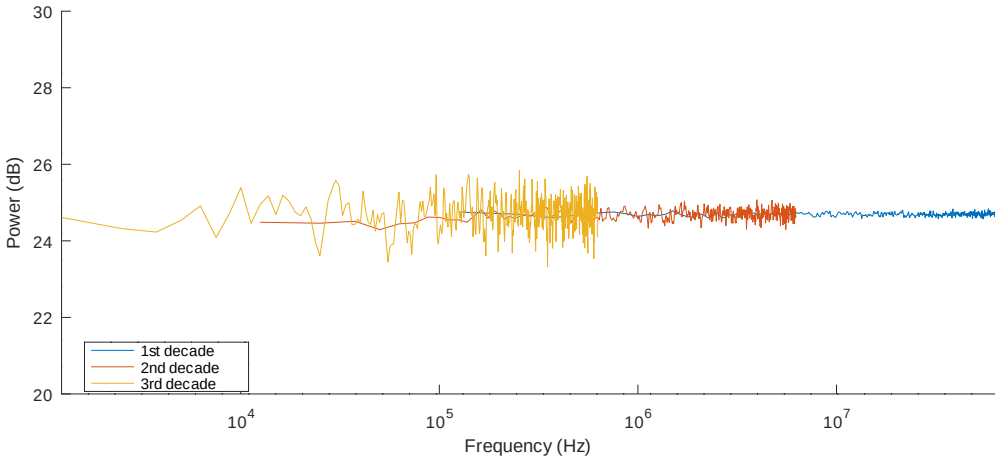


FIGURE 1.8 – Densité spectrale d'un bruit blanc échantillonné à 125 MHz

un pas de résolution spectrale constant. Décimer revient à prendre un point tous les P dans le domaine temporel, ce qui revient à passer d'une fréquence d'échantillonnage de f_N à f_N/P. De ce fait, la résolution dans la nouvelle décade sera $f_N/(P \times n)$. Par définition, les décades sont obtenues en sélectionnant $P = 10$.

La figure 1.8 illustre l'analyse d'un bruit blanc, i.e. signal caractérisé par une distribution constante des composantes spectrales. La fréquence f_s est de 125 MHz, la courbe bleue représente la première décade puis nous l'avons décimé par dix, f_s a donc aussi été divisé par dix c'est pourquoi la deuxième décade (la courbe rouge) ne s'étale que jusqu'à 6,25 MHz. On répète l'opération encore une fois pour obtenir le dernière décade (la courbe orange). On voit qu'à chaque nouvelle décade, on améliore la résolution de la densité de bruit pour des fréquences de plus en plus faibles.

Cependant, la décimation va induire du repliement spectral. En effet, puisque la fréquence f_s est divisée par le nombre de points perdus, il se peut que des composantes spectrales plus grandes que f_N soient ramenées dans la bande de base par effet de repliement spectral comme l'illustre la figure 1.9. Dans cette figure, nous avons à gauche la vue du signal dans le domaine temporel et à gauche la vue dans le domaine spectral. Pour illustrer le principe du repliement, nous avons pris un même signal (à 10 kHz) et nous l'échantillonnons une fois à 250 kHz (la courbe bleu), puis une autre fois à 12,5 kHz (la courbe rouge). Le signal en rouge peut être considéré comme le résultat du signal bleu décimé par 20 ($250/20 = 12,5kHz$). Échantillonner à 12,5 kHz signifie qu'on ne peut mesurer un point que toutes les 0,8 ms ($1/12,5 \cdot 10^3 = 0,8ms$), ces mesures sont données par les ronds rouges. La figure de droite montre ce qu'il se passe dans le domaine spectral. Le signal qu'on observe est décalé de 3,75 kHz de plus que la fréquence de Nyquist (f_N). Il sera donc ramené par symétrie par rapport à f_N à 2,5 kHz. C'est pourquoi le signal initial à 10 kHz peut être confondu avec un signal à 2,5 kHz si la fréquence d'échantillonnage n'est pas assez élevée.

Pour éviter ce phénomène, il est possible d'appliquer un filtre passe-bas (un filtre qui ne garde que les composantes basses fréquence dans le spectre du signal) avant de faire la

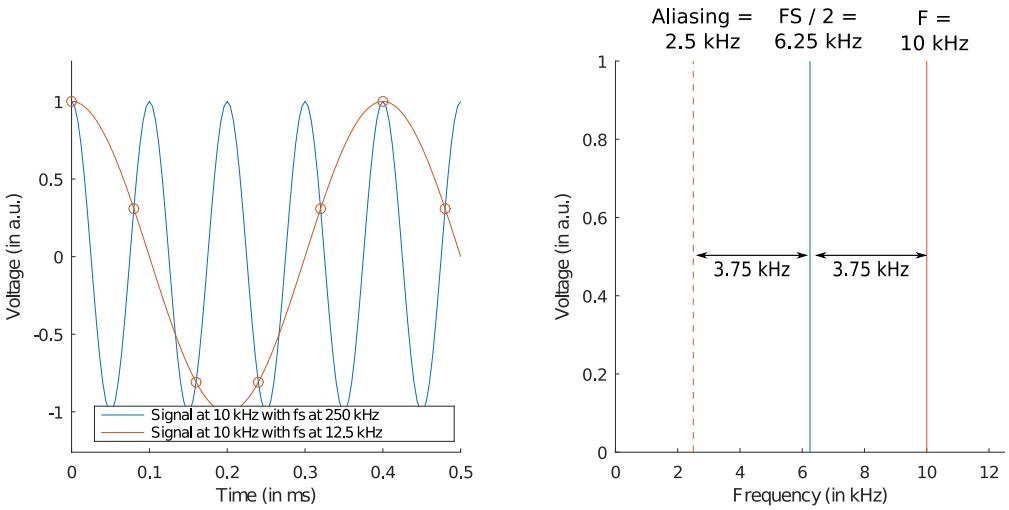


FIGURE 1.9 – Illustration de l'effet du repliement spectral

décimation. En effet, le but d'un filtre passe-bas est de couper les composantes spectrales à haute fréquence donc, en choisissant le bon gabarit de filtre (quelles sont fréquences gardés et les fréquences rejetées), on peut rejeter les hautes fréquence pour qu'elles ne soient pas ramenées en bande base lors de la décimation. Si on reprend l'exemple de la figure 1.9, il faudrait que le filtre ne garde que les premiers 5% du spectre, alors que l'exemple arbitraire présenté dans la figure 1.10a rejette 50% des hautes-fréquence, on pourrait donc décimer par 2 le signal.

Lorsqu'on veut traiter numériquement un signal représenté par des entiers signés, on connaît le nombre de bits effectifs (en anglais Effective *Number Of Bits* : ENOB) des données en entrée. Cette valeur nous indique seulement le nombre pertinent de bits sur lequel le signal sera codé. Plus ce nombre est grand, meilleure sera la résolution du signal. Cet ENOB joue également un rôle sur le niveau de bruit du signal étudié. Si on n'a pas beaucoup d'ENOB, on est capable d'échantillonner un signal rapidement mais nous allons introduire beaucoup de bruit car la résolution du signal sera faible. À l'inverse avec un ENOB important on a une grande résolution sur le signal mais on est contraint à étudier des signaux plus lents. Cette valeur de ENOB nous indique également le rapport signal à bruit (en anglais, *Signal-to-Noise Ratio* : SNR). Soit N la valeur de ENOB, alors le SNR est défini de la façon suivante :

$$SNR \approx 6,02 \times N + 1,76 \text{ dB} \tag{1.3}$$

Cette valeur indique le niveau maximal de bruit que le signal peut supporter sans être dégradé.

À partir de l'équation 1.3 on peut également considérer le problème sous un autre angle. Dans le cas présent, nous cherchons à réduire, à la sortie de chaque filtre, le nombre de bits sur lesquels les données sont codées. Ce nombre est, dans notre cas, le ENOB du signal : après chaque filtrage nous augmentons donc la résolution du signal. Le problème est que le niveau de bruit de ce signal n'est pas assez bas pour que le ENOB soit pleinement utile, nous avons donc des bits significatifs superflus. Nous pouvons calculer le nombre de bits superflus car nous verrons que, par conception, nous connaîtrons la quantité de

bruit que le filtre va retirer du signal. Nous connaissons ainsi le SNR à la sortie du filtre. Nous déduisons donc la nouvelle valeur de ENOB de la façon suivante :

$$N' = \lceil \frac{1}{6,02} \times SNR - 1,76 \rceil \text{ bits} \tag{1.4}$$

Grâce à cette nouvelle valeur de ENOB, on sait quel est le nombre de bits minimal pour coder les données sortantes sans rajouter du bruit au signal. Si on décale trop vers la droite les bits représentant nos données, nous rajouterons $20 \cdot \log_{10}(2) \simeq 6$ dB de bruit par bit utile perdu.

Dans la suite de nos travaux, nous allons nous concentrer essentiellement sur cette étape clé de filtrage. Nous avons vu qu'elle était indispensable pour nous permettre d'analyser le bruit de phase d'un signal. Dans la section suivante nous allons voir en détail comme filtrer un signal.

1.3 Filtrage

L'étape de filtrage est le cœur de nombreux traitements. En effet, il est important de bien contrôler quelles sont les composantes spectrales qu'on observe. De plus dans le cadre de signaux à haut débit, il est impératif de pouvoir traiter à temps tous les échantillons. Les filtres doivent donc être performants pour éviter de perdre des données.

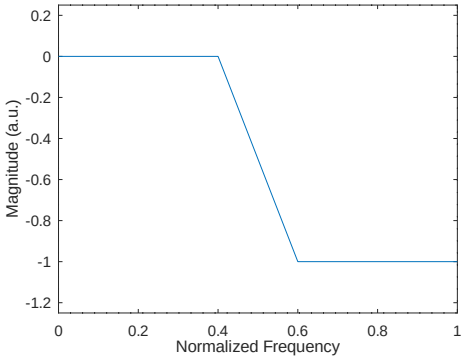
Dans cette section nous allons détailler ce qu'est un filtre et comment on les conçoit. Puis nous ferons un état de l'art sur les différentes méthodes d'optimisation des filtres.

1.3.1 Définitions

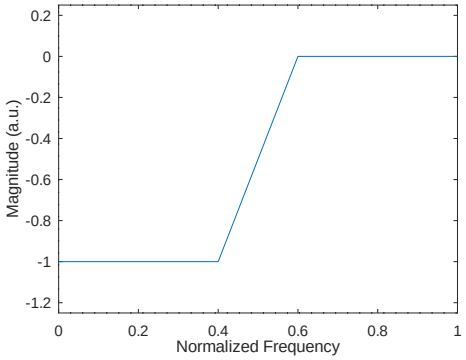
Dans le cadre de cette thèse, nous allons appliquer une méthode d'optimisation de blocs de traitement au cas particulier des filtrages numériques de signaux échantillonnés en temps discret, représentés par des entiers naturels. Il existe plusieurs types de filtrage : filtre passe-bas (low-pass filter), filtre passe-haut (high-pass filter), filtre passe-bande (band-pass filter) ou filtre coupe-bande (notch). Ces quatre types de filtres ont respectivement pour tâche de couper les hautes-fréquences, couper les basses-fréquences, de ne laisser passer qu'une bande de fréquences ou de couper une bande de fréquences (généralement étroite si la source de bruit est à bande étroite (raie parasite)).

Les figures 1.10a, 1.10b, 1.10c et 1.10d schématisent respectivement un filtre passe-bas, passe-haut, passe-bande, coupe-bande.

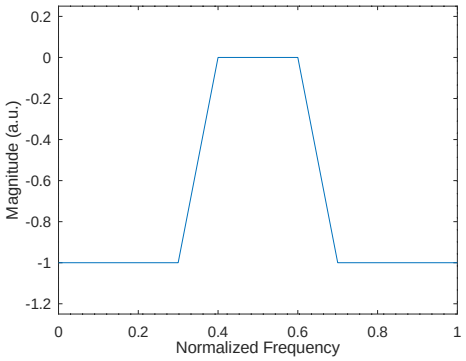
Chaque filtre est créé pour répondre à des besoins : rejeter telle bande de fréquences, éliminer les hautes ou les basses fréquences... Définir ces caractéristiques revient à donner le gabarit d'un filtre.



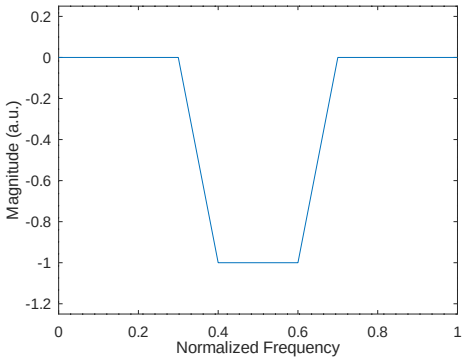
(a) Schéma d'un filtre passe-bas



(b) Schéma d'un filtre passe-haut



(c) Schéma d'un filtre passe-bande



(d) Schéma d'un filtre coupe-bande

FIGURE 1.10 – Schémas des différents types de filtre

1.3.2 Filtre à réponse impulsionnelle

Il existe plusieurs méthodes de filtrage d'un signal. Dans les filtres linéaires, deux catégories de filtres existent : les filtres à réponse impulsionnelle finie (Finite Impulse Response filter ou FIR filter) et les filtres à réponse impulsionnelle infinie (Infinite Impulse Response filter ou IIR filter). La principale différence entre ces deux types de filtres est que la sortie du premier ne dépend pas des sorties qu'il a générées précédemment, alors que la sortie du second dépend des sorties précédentes. En effet, les filtres IIR prennent comme données d'entrée des échantillons à traiter et des échantillons déjà traités. Cette particularité permet aux filtres IIR d'être très efficaces avec très peu de coefficients mais étant donné que le système fonctionne comme une boucle de rétroaction, il faut s'assurer de sa stabilité en fonction des jeux de coefficients utilisés. En revanche, les filtres FIR n'utilisant pas de données déjà traitées, ils sont inconditionnellement stables quel que soit le jeu de coefficients utilisé et quelles que soient les données en entrée. Cependant ces filtres nécessitent plus de coefficients pour obtenir les mêmes performances qu'un filtre IIR. Dans notre cas, puisque nous voulons générer plusieurs jeux de coefficients automatiquement, il serait beaucoup trop long de s'assurer que toutes ces combinaisons sont stables, nous avons donc restreint notre choix aux FIR. Dans la suite du document, lorsque nous parlerons de filtres, nous ferons implicitement référence à des FIR.

Mathématiquement parlant, un filtre FIR est défini comme un produit de convolution dont la forme classique est :

$$y_k = \sum_{i=0}^{|b|-1} b_i \times x_{k-i} \tag{1.5}$$

x représente les données échantillonnées du signal à filtrer, b représente le jeu de coefficients utilisé lors du filtrage du signal et enfin y est le résultat du filtrage. Dans tous les cas, les indices $_i$ et $_k$ représentent le temps discret. Dans ce contexte la notation $|b|$ représente la cardinalité de notre ensemble.

1.3.3 Contributions scientifiques

Les filtres ont été largement étudiés dans le monde scientifique. Ils sont en effet primordiaux dans de nombreux domaines et notamment en métrologie du signal. Il est donc logique que de nombreuses études se soient focalisées sur l'optimisation des filtres et sur leur application dans les FPGA.

Nous pouvons classer ces recherches en deux catégories :

1. Optimisation algorithmique et/ou matérielle
2. Génération de coefficients optimaux

1.3.3.1 Optimisation algorithmique et/ou matérielle

Dans cette première catégorie on peut citer les travaux de Parker et Parhi. Dans [75], ils proposent une nouvelle façon d'implémenter l'algorithme classique du FIR et tirant parti du parallélisme des opérations. En effet, les auteurs expliquent que pour optimiser l'algorithme d'un filtre, ils faut suivre plusieurs étapes :

1. Définir les coefficients en fonction des besoins
2. Décomposer le filtre initial en plusieurs filtres parallèles
3. Quantifier les coefficients
4. Appliquer de différentes techniques d'optimisation matérielle

La première étape dépend uniquement du contexte et des besoins de l'utilisateur, il est donc difficile d'optimiser cette étape. La deuxième étape consiste à tirer parti de la linéarité et de la symétrie des FIR pour adapter l'algorithme et ainsi réduire la complexité des opérations. La troisième étape nécessite de convertir les coefficients du filtre en entiers signés. Enfin la dernière étape consiste à réduire les coûts matériels. La conclusion du travail de Parker et Parhi est que pour bien optimiser un filtre, il faut avoir une démarche d'ensemble et non se concentrer uniquement sur certains aspects.

Un architecture possible pour optimiser les filtres FIR est *Residue Number System (RNS)* [86, 85, 52]. RSN est une méthode basée sur le théorème des restes chinois afin de diminuer le nombre de multiplieurs et de pouvoir paralléliser une partie du traitement. Dans [70], Nannarelli *et al.* comparent les performances de cette architecture, en terme

temps de calculs (latence), de consommation d'énergie et de ressources, avec l'architecture basique. Ils arrivent à la conclusion que cette nouvelle architecture est plus avantageuse en tous points que l'implémentation de référence.

En terme d'opérations, le filtre ne compte que des additions et des multiplications. Dans un contexte de calcul en temps réel, les multiplications peuvent être lentes. Cependant, il existe une architecture permettant de remplacer ces opérations coûteuses par des approximations : *Radix-2 Signed Digit (SD)*. Dans [54], Kawahito *et al.* présentent cette architecture.

Dans [61], Lindahl et Bengtsson proposent une architecture combinant un filtre FIR RSN et les SD. Il s'agit de tirer parti de deux méthodes d'implémentation d'un filtre pour réduire le temps de calcul (la latence), diminuer la consommation électrique (puissance) et pour minimiser les coûts matériels en diminuant le nombre de multiplieurs dans la chaîne de traitement. Les auteurs démontrent grâce à cette architecture hybride, qu'ils obtiennent de meilleurs résultats que l'implémentation classique. Ici encore, cette méthode se focalise sur l'optimisation algorithmique et matérielle.

Une autre façon d'optimiser un FIR filter est d'adapter les coefficients pour qu'ils aient des propriétés mathématiques plus pratiques lors des calculs. On peut citer par exemple les travaux de Dam *et al.* [26] ou encore Lim *et al.* [25]. En effet, ils n'expriment les coefficients que sous forme de puissance de deux ou d'une somme de puissances de deux car la multiplication peut être remplacée de manière exacte par un décalage logique. Ce faisant, le design gagne en rapidité et on économise des multiplieurs. Dans [90], Tsao et Choi utilisent des jeux de coefficients symétriques afin d'accélérer et de réduire le consommation de ressources.

1.3.3.2 Génération de coefficients optimaux

Dans cette section, nous allons présenter différentes techniques existantes pour choisir les coefficients les plus optimisés pour obtenir ls meilleur filtre possible.

En 2016, Chandra et Chattopadhyay dressent un état de l'art [16] sur les façons de concevoir un FIR. Ils présentent les différentes méthodes que nous avons présentées dans la partie précédente mais ils vont plus loin en confirmant que le choix du jeu de coefficients d'un filtre est un problème d'optimisation. À ce titre, ils présentent de nombreuses techniques de recherche opérationnelle qui répondent à ce problème.

La première catégorie de techniques que présentent les auteurs est basée sur la résolution de programme linéaire ou quadratique [80, 91] :
- *Mixed Integer Linear Programming (MILP)* [43]
- *Integer semi-infinite linear programming* [50]
- *Semi Definite Programming (SDP)* [62]
- *Discrete Semi-Infinite Linear Programming (DSILP)* [49]
- *Mixed Integer Programming (MIP)* [34]

À cela s'ajoutent des algorithmes de *branch and bound* pour faciliter la recherche d'une

solution en nombres entiers dans le cas des MILP[50, 71].

La seconde catégorie d'outils issus de la recherche opérationnelle sont les méthodes utilisant des graphes [27, 42, 93]. Les graphes sont souvent utilisés pour diminuer la complexité des opérations (par exemple, pour réduire le nombre de multiplications). Pour ce faire, les graphes sont construits itérativement de bas en haut *(bottom-up)*. À chaque itération, une heuristique est utilisée pour définir le prochain nœud ajouté au graphe.

La dernière catégorie présentée est l'utilisation de métaheuristiques :
— Recherche tabou [89]
— Algorithme génétique [15, 94, 37]
— Essaim particulaire [63]
— Colonie artificielle d'abeilles [65, 64]
D'après Chandra et Chattopadhyay, les méthodes basées sur cette dernière métaheuristique semblent s'être imposées par leur efficacité.

Il est également possible d'obtenir de bonnes performances en cascadant plusieurs filtres. Dans [59] Lim *et al.* cherchent à diminuer les variations *(ripples)* dans la bande passante. En effet, des fluctuations trop fortes dans la bande passante peuvent fausser l'interprétation des résultats car cela peut amplifier ou diminuer le gain de certaines composantes spectrales dans la bande passante. Les auteurs constatent que lorsqu'on utilise un filtre avec des valeurs en nombres réels, les *ripples* sont moins forts que lorsqu'on utilise des coefficients entiers. Ils expliquent cette différence par l'impact de la troncature des nombres réels en nombre entiers. Ils expliquent également que ces fluctuations peuvent être limitées si on utilise deux filtres successifs avec des coefficients spécifiques pour compenser les déviations dans la bande passante.

Dans [60], Lim *et al.* approfondissent leurs travaux. Ils choisissent un premier filtre qui répond à leurs besoins puis ils utilisent un *MILP* pour trouver les coefficients optimaux du second filtre afin de compenser au mieux les déviations dans la bande passante. L'article a été publié en 1996 et à cette époque les auteurs étaient limités par la puissance de calcul insuffisante des machines de l'époque pour pouvoir traiter des cascades plus grandes ou des filtres ayant plus de coefficients.

Young et Jones reprennent aussi l'idée de cascade de filtres dans [96] mais pour réduire la consommation de ressources. En effet, ils proposent une méthode pour définir conjointement les coefficients de deux filtres FIR. Dans leur conclusion, les auteurs disent que pour obtenir une meilleure réjection dans la coupe-bande, il est préférable d'utiliser des filtres en cascade.

Dans le cadre de cette thèse, nous cherchons à optimiser le filtrage. Il existe beaucoup de coefficients possibles pour répondre à un gabarit donné et on peut faire varier la précision de ces coefficients. Tester exhaustivement toutes les combinaisons possibles entre les jeux de coefficients et leur précision est impossible car il y a beaucoup trop de possibilités. Nous allons avoir besoin de techniques et d'outils plus performants pour trouver une combinaison optimale. Nous allons faire de la recherche opérationnelle. Dans la section suivante, nous allons présenter plusieurs façon de procéder pour chercher une solution dans ce genre de situation.

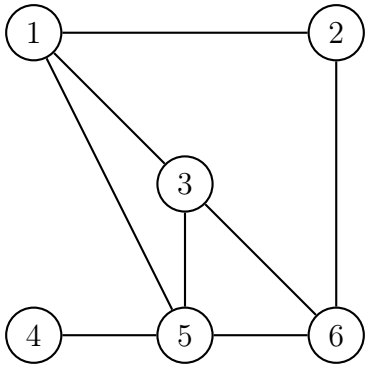


FIGURE 1.11 – Exemple d'un graphe quelconque

1.4 Recherche opérationnelle

La recherche opérationnelle est un domaine de l'informatique qui regroupe toutes les méthodes de recherches de solutions optimales pour des problèmes spécifiques. Ces méthodes doivent parcourir efficacement l'espace de recherche (l'ensemble des solutions) pour trouver une solution optimale ou sous-optimale selon les contextes.

Il existe beaucoup des méthodes différentes et chacune d'elles a ses spécificités et ses caractéristiques. Ces méthodes sont couramment utilisées pour résoudre des problèmes de type :

- Combinatoire [74] : recherche d'une solution ou d'un ensemble de solutions optimales, qualifiées par une fonction objective, dans un ensemble discret et fini donné. On peut citer le problème du sac à dos comme exemple d'un problème combinatoire. En effet, dans ce problème il s'agit de remplir un sac, dont la contenance est limitée, avec des objets ayant une valeur et un poids différents de manière à maximiser la valeur totale du sac.
- Ordonnancement [78, 21] : recherche d'une planification de tâches et/ou d'attributions de ressources permettant de garantir un temps d'exécution minimal. C'est le cas quand on cherche à répartir la charge (*load balancing)* des requêtes d'un service entres plusieurs serveurs ou encore lorsque l'ordonnanceur d'un ordinateur affecte des tâches à ses processeurs.

Les espaces de recherche peuvent être représentés par des graphes. En informatique, un graphe est défini comme un ensemble de nœuds reliés entre eux par des arrêtes. La figure 1.11 représente un graphe quelconque mais il existe d'autres types de graphes qui ont leur propres caractéristiques.

Nous allons tout d'abord, nous intéresser au graphe orienté acyclique (*directed acyclic graph*, DAG). Dans un graphe classique les arêtes sont bidirectionnelles, on peut donc suivre l'arrête pour aller du nœud A au nœud B et inversement de B à A. Dans un graphe orienté, les arrêtes sont directionnelles, on ne peut donc aller que dans un sens. De plus un DAG est acyclique c'est à dire qu'en partant de n'importe quel nœd il est impossible de retomber sur un nœud déjà visité. La figure 1.12 donne un exemple de DAG.

Un dernier type de graphe que nous citerons est l'arbre. Il s'agit d'une forme un peu

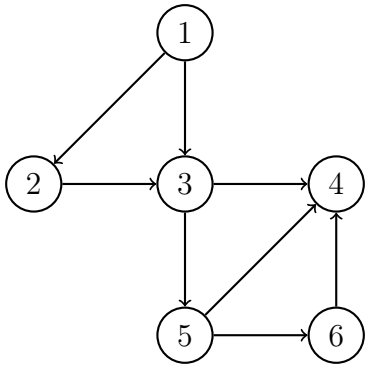


FIGURE 1.12 – Exemple d'un graphe orienté acyclique (*directed acyclic graph*, DAG)

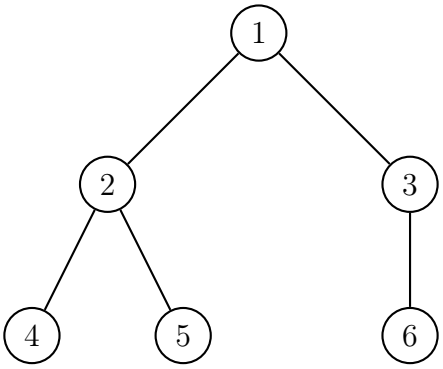


FIGURE 1.13 – Exemple d'un arbre

particulière d'un DAG qui possède une racine unique (un seul nœud qui n'a aucune arrête qui arrive sur lui) et dont tous les autres nœuds n'ont qu'un seul parent (une seule arrête arrive sur le nœud). La figure 1.13 donne un exemple d'arbre.

Il existe deux catégories de méthodes utilisées dans la recherche opérationnelle : les algorithmes qui produisent des solutions exactes et les métaheuristiques ou les algorithmes d'approximation qui produisent des résultats suboptimaux mais satisfaisants.

1.4.1 Algorithmes optimaux

Ces méthodes produisent des solutions optimales mais peuvent être lentes lorsque les instances des problèmes sont trop grandes. Parmi tous les outils utilisés, on peut citer :

1.4.1.0.1 La programmation linéaire [6, 11, 67] Il s'agit d'exprimer le problème à traiter par un ensemble d'inéquations linéaires et de chercher à maximiser ou minimiser une fonction objective. Quand les variables d'un tel système sont des nombres réels, on utilise généralement l'algorithme du Simplexe pour trouver une réponse optimale. Cependant, si des variables sont entières la résolution devient plus complexe. Habituellement, on considère, dans un premier temps que toutes les variables sont des nombres réels et on applique le simplexe, puis on regarde dans un espace proche de la solution pour trouver

une solution optimale.

1.4.1.0.2 La programmation dynamique [7] Le principe de cette méthode est de décomposer un problème en plusieurs sous-problèmes et de résoudre séparément ces sous-problèmes avant de converger vers une solution globale.

1.4.1.0.3 Les méthodes arborescentes Il s'agit ici de représenter l'espace de recherche sous la forme d'un arbre puis d'utiliser des algorithmes du type A* [45] ou encore des techniques de séparation et évaluation [57] *(branch and bound)*.

1.4.2 Heuristiques, métaheuristiques et algorithmes d'approximation

Lorsque trouver une solution optimale est trop long ou que cette solution est inexploitable, on a recours à des méthodes non-optimales mais suffisamment efficaces pour produire une solution proche de l'optimal. La définition de proche dans ce contexte dépendra de la méthode. En effet, une heuristique consiste à utiliser un algorithme qui produit une solution. On peut imaginer une heuristique qui consiste à prendre une solution possible sans aller en tester d'autres. Le temps de calcul sera bien plus court mais la solution a de fortes chances d'être très éloignée de la solution optimale. Dans les faits, les heuristiques sont mieux pensées et elles sont plus proches de l'optimal mais rien ne prouve que toutes les solutions sont proches de l'optimal.

Si on prouve qu'une heuristique est toujours proche de l'optimal, alors il s'agit d'un algorithme d'approximation. Avec un tel algorithme, on peut faire confiance à tous les résultats car il existe une preuve formelle qui démontre que la différence avec l'optimal est connue. Bien entendu, trouver une telle preuve est ardu c'est pourquoi il n'existe pas toujours des algorithmes d'approximation mais seulement des heuristiques.

Enfin, il reste toute une catégorie d'outils appelés métaheuristique. Il s'agit d'un ensemble de méthodes génériques qui peuvent s'appliquer à la plupart des situations. Elles sont souvent basées sur des principes stochastiques : nous ne sommes donc pas sûrs d'atteindre une solution très proche de l'optimale du premier coup. Il est donc nécessaire de répéter l'opération de nombreuses fois pour se rapprocher d'une solution optimale.

Il existe beaucoup de métaheuristiques différentes, mais nous allons n'en présenter que quelques unes.

1.4.2.0.1 Algorithme génétique [41] Le principe de cette métaheuristique est de faire *évoluer* la solution et de la rendre meilleure à chaque génération. Pour ce faire, il faut représenter la solution sous forme de gènes (la plupart du temps un nombre dont on prend les bits comme gène). Ensuite on choisit au hasard un certain nombre de gènes différents et on évalue leurs performances. Vient ensuite l'étape de croisement qui change

selon les cas et les besoins. On peut ne choisir de croiser les gènes que des meilleurs solutions, on peut aussi prendre des gènes des autres solutions afin d'avoir un brassage génétique. Le pourcentage de gènes changés de génération en génération varie lui aussi. Il est aussi possible d'intégrer des mutations spontanées dans les gènes pour éviter de stagner dans des optimaux locaux. Une fois que la nouvelle génération est créée, on recommence l'évolution. Le cœur d'un algorithme est de bien représenter les solutions sous forme de gènes et de bien choisir les différents paramètres pour ne pas rater de solution optimale et pour ne pas être bloqué dans des optimaux locaux.

1.4.2.0.2 Colonie de fourmis [20] Il s'agit ici de mimer le comportement réel d'une colonie de fourmis. Pour trouver leur nourriture, les fourmis envoient des éclaireuses et quand celles-ci ont trouvé de la nourriture, elles reviennent à leur colonie en déposant des phéromones. Les autres fourmis suivront le chemin de phéromones et plus un chemin a de passage, plus les phéromones seront concentrés. Si un chemin plus court est trouvé, certaines fourmis vont choisir ce chemin et comme il est plus court, la concentration en phéromones augmentera plus vite attirant d'autant plus d'autres fourmis. Ainsi, le premier chemin sera peu à peu déserté par les fourmis au profit du chemin le plus court. Une première application d'une telle idée nous est proposée dans [31]. Par la suite, le principe des colonies de fourmis fut adapté à d'autres espèces comme par exemple les colonies artificielles d'abeilles [53].

1.4.2.0.3 Essaim particulaire [55] Cette métaheuristique se rapproche de l'idée des colonies des fourmis. En effet il s'agit de faire collaborer un ensemble d'éléments aux capacités limitées afin de chercher des minima dans l'ensemble de l'espace de recherche. Le principe de base est de faire déplacer une population appelée *essaim* parmi les solutions possibles appelées *particules*. Chaque individu connaît la particule optimale déjà découverte et leur mouvement dépendra de cette information.

1.4.2.0.4 Recherche par Tabou [38, 39] Cette méthode consiste à explorer des minimums locaux jusqu'à trouver une solution satisfaisante (qui minimise suffisamment une fonction objective) ou après un nombre limité d'itérations. Afin de maximiser l'efficacité de la recherche dès qu'on a exploré une solution locale, on mémorise les différentes solutions explorées pour ne pas les ré-explorer aux itérations suivantes.

1.4.3 Application à des problèmes de gestion de matériel

Toutes ces méthodes (optimales ou approchées) peuvent nous aider à trouver le meilleur filtrage possible. Il faudra être capable de reformuler ou d'adapter notre recherche au formalisme spécifique des différentes méthodes mais cela nous permettra de chercher plus efficacement une solution optimale (ou sous-optimale) que si on devait faire un parcours exhaustif.

Comme nous avons pu le constater, la conception d'un filtre peut être assimilée à un problème d'optimisation. À ce titre, il est donc possible d'utiliser des outils classiques de la recherche opérationnelle pour trouver une solution optimale ou proche de l'optimale. Cependant, on peut constater qu'il existe d'autres situations où il est possible de faire appel à de tels outils pour résoudre des problèmes matériels.

On peut citer le cas des micro-usines [22, 30]. Il s'agit d'optimiser la chaîne de production d'une micro-usine tout en tenant compte des pannes qui peuvent survenir. L'enjeu majeur est donc d'être capable d'adapter une solution optimale (la chaîne de production au départ) quand un certain nombre d'étapes de cette chaîne sont dysfonctionnels. On peut donc faire une analyse abstraite de la chaîne de traitement et la représenter sous la forme d'un graphe. On connaît toutes les connexions possibles entre les différents nœuds (les tâches), il est donc possible de trouver un nouveau chemin quand l'un des nœuds devient inaccessible (tombe en panne).

Dans cette thèse, nous allons aussi résoudre un problème d'optimisation matérielle avec des outils informatiques plus abstraits. Nous voulons également développer une méthodologie qui permettrait d'optimiser n'importe quelle chaîne de traitement numérique.

Pour créer et valider notre approche, nous allons ré-étudier le problème des filtres en cascade. En effet, dans [59, 96, 60] les auteurs avaient déjà conclu qu'il était préférable de concevoir des filtres discrets en cascade pour obtenir de meilleures performances et diminuer la consommation de ressources. Ainsi, si avec notre méthode nous arrivons aux mêmes conclusions, nous prouverons que cette approche est juste et qu'elle pourra être adaptée à d'autres traitements et/ou à d'autres contextes. Dans le chapitre 2 nous détaillerons notre méthode, dans le chapitre 3 nous l'appliquerons à un cas concret, les cascades de filtres. Pour finir dans le chapitre 4 nous générons et analyserons des données expérimentales issues de nos résultats abstraits.

Deuxième partie

Méthodologie

Chapitre 2

Méthodologie générique pour l'optimisation de flux de traitement

Dans le chapitre précédent, nous avons défini les notions et le vocabulaire nécessaires pour bien comprendre ces travaux de recherche. Nous avons également passé en revue les travaux scientifiques relatifs à notre sujet pluridisciplinaire. Nous avons abordé le coté électronique, que cela soit l'aspect métrologie du signal numérique ou l'aspect traitement sur FPGA, ainsi que le coté informatique sur l'optimisation de flux de traitement et de recherche de solutions optimales.

Ce chapitre sera consacré à la présentation et à l'explication de notre méthodologie. Elle consiste à créer un modèle abstrait d'un traitement sur FPGA concret. À partir de cette abstraction, on peut utiliser des outils issus de la recherche opérationnelle pour trouver une solution optimale. Nous détaillerons cette approche dans la section 2.1.

Afin de pouvoir construire notre modèle abstrait, nous expliquerons comment on peut définir un bloc de traitement dans la section 2.2. Dans la section 2.3, nous verrons comment représenter toute une chaîne de traitement et les différentes relations entre les blocs. Dans la section 2.4, nous détaillerons la modélisation générique d'un filtre FIR.

2.1 Description du principe général

Le but de cette thèse est de transposer des outils informatiques d'optimisation dans un contexte électronique. Comme nous l'avons vu dans le chapitre précédent dans la section 1.3, de nombreux travaux s'inscrivent dans cette optique.

Dans notre cas, nous proposons une approche nouvelle visant à optimiser n'importe quel traitement numérique. Pour créer cette méthode générique il nous faut impérativement une abstraction très forte. Ainsi, nous visons à ne pas être restreint par une technologie, un constructeur de FPGA ou un logiciel de synthèse propriétaire spécifique. Néanmoins, nous nous efforcerons de démontrer la pertinence de l'approche sur des solu-

tions concrètes.

Notre méthodologie se décompose en quatre étapes :

1. définir une modélisation d'un problème de traitement numérique concret
2. trouver une solution optimale à ce modèle
3. transformer et appliquer la solution abstraite dans un cas concret
4. comparer les résultats concrets avec les résultats abstraits pour s'assurer que le modèle est cohérent.

La première étape est spécifique à chaque problème et le niveau d'abstraction est un paramètre crucial.

Avoir une abstraction trop forte risque de nous éloigner grandement de la réalité et cela nous empêcherait d'obtenir des résultats concrets. En effet, avec une modélisation trop abstraite du problème, nous pourrons toujours trouver des solutions abstraites mais qui ne seraient pas forcément réalisables dans la pratique.

En revanche, si nous optons pour une abstraction trop faible, nous risquons de perdre des degrés de liberté lors de la recherche optimale, car nous aurons à tenir compte de trop de contraintes liées au cas traité. Par conséquent, les solutions optimales trouvées seraient trop prévisibles car les contraintes seraient trop nombreuses et / ou trop fortes.

Il est donc important de définir un niveau d'abstraction suffisant pour avoir des solutions intéressantes mais en restant proche du cas réel pour que la solution optimale puisse être exploitée dans une application concrète.

Une fois que le modèle est conçu, la deuxième étape consiste à définir un critère d'optimisation (ou plusieurs si on veut faire de l'optimisation multi-objectif) et de trouver la solution optimale qui va maximiser ou minimiser l'objectif souhaité.

Selon la complexité du modèle et les contraintes, on va recourir à différents moyens. Si le problème est simple (peu de contraintes et peu de paramètres), il est probablement possible d'écrire un algorithme pour donner la solution optimale. En revanche si le problème est trop compliqué, on va avoir recours à des outils de la recherche opérationnelle pour trouver une solution optimale (nous avons présenté une partie des techniques possibles dans la section 1.4 du chapitre précédent). Cette étape peut prendre beaucoup de temps (de l'ordre du mois ou plus) pour fournir une solution, c'est pourquoi dans certains cas il peut être utile d'avoir recours à des méthodes sous-optimales (heuristiques et méta-heuristiques).

Une fois la solution trouvée, la troisième étape consiste, dans un premier temps, à la transformer en application réelle. Autrement dit, on doit générer une application matérielle (un design FPGA par exemple) à partir de la solution optimale. Dans un second temps, il faut faire l'expérimentation et à récupérer des données concrètes.

Pour finir, la quatrième et dernière étape consiste à analyser les résultats expérimentaux pour voir s'ils sont cohérents avec ceux attendus (d'après la solution abstraite). Si c'est le cas, la modélisation est suffisamment cohérente et on peut décliner le problème

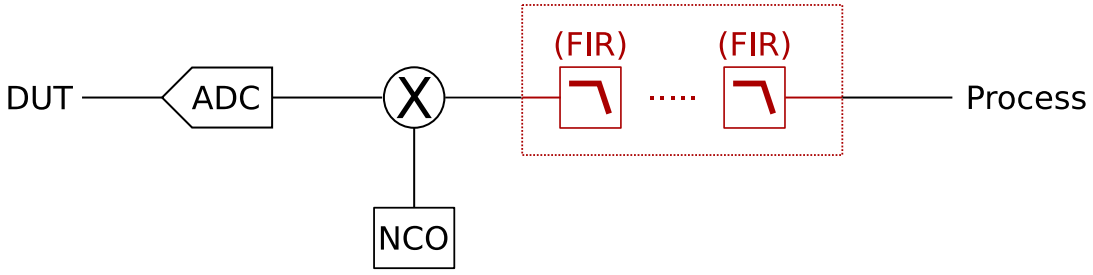


FIGURE 2.1 – Schéma de la chaîne de traitement type pour rejeter du bruit

initial en changeant les paramètres du système pour étudier plus en détail le traitement. Si en revanche, les résultats sont trop éloignés de nos attentes, cela veut dire que notre modèle n'est pas bon. Il faut alors l'adapter, et recommencer à l'étape 1.

Dans la suite de ce chapitre, nous allons nous attacher à expliquer par l'exemple le déroulement de la première étape. Les autres étapes quant à elle seront détaillées dans le chapitre suivant.

2.2 Modélisation d'un bloc de traitement

Comme nous venons de le voir, la première étape de l'analyse est de définir un modèle du traitement. Nous illustrons notre méthode en regardant comment maximiser la réjection (la quantité de bruit éliminée lors du traitement) dans un traitement fait sur FPGA. Le schéma présenté dans la figure 2.1 montre la chaîne de traitement avec laquelle nous travaillerons. Notre objectif sera d'optimiser les filtres (le cadre en rouge).

La question est donc savoir s'il est préférable de une cascade de filtres plutôt qu'un filtre monolithique plus gros. Ce problème a déjà été traité dans le passé et nous savons qu'il est préférable d'avoir une cascade de filtres [59, 96, 60]. Si on parvient aux mêmes conclusions avec notre approche, nous aurons démontré qu'elle est valable dans cette situation. Il s'agit là d'une étape indispensable avant de pouvoir appliquer cette méthodologie dans une autre application.

Nous aurons besoin plus tard de créer un design de ce traitement (étape 3 de notre méthode) : nous avons décidé de séparer le flux de traitement en tâches élémentaires afin d'utiliser la philosophie des skeleton et de faciliter ainsi la création d'applications matérielles.

Dans cette section, nous montrerons comment on a modélisé un bloc de traitement et quelles sont les paramètres ajustables pour trouver une solution optimale.

Les ressources dans les FPGA ne sont pas illimitées comme nous l'avons mentionné dans la section 1.1.2 du chapitre précédent. Un FPGA dispose de différentes ressources (CLB, DSP, RAM...) qui pourraient être modélisées individuellement, mais ce faisant notre niveau d'abstraction serait déjà trop bas et notre modèle pourrait ne pas être suffisamment abstrait. Nous décidons qu'une tâche i occupera une surface a_i de silicium

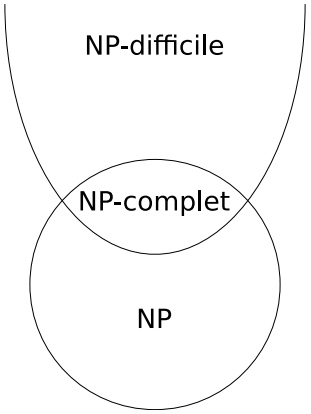


FIGURE 2.2 – Représentation graphique des différentes classes de problème

dans le FPGA. Cette surface sera exprimée en unité arbitraire (u.a.) afin de ne pas nous focaliser sur des considérations trop techniques. Cette surface dépendra de la complexité de la tâche : plus elle sera complexe, plus elle consommera de ressources.

À cela, nous définissons également la variable r_i qui va donner pour la tâche i, la quantité de bruit rejeté en dB. Si cette valeur est positive, on rejette du bruit (dans le cas d'un filtre par exemple), sinon on en introduira (dans le cas d'une décimation par exemple). Le système 2.1 représente notre première modélisation.

$$\begin{cases} a_i \\ r_i \end{cases} \tag{2.1}$$

Grâce à elle, nous pouvons faire le rapprochement avec le problème classique du sac à dos *(Knapsack problem)*. Pour rappel, il s'agit d'une personne qui doit remplir son sac avec des objets qui ont une valeur et une masse données. Cependant le sac à dos a une capacité limitée : la question est donc de savoir quels objets prendre afin d'avoir la plus grande valeur possible.

Le problème du sac à dos est un problème NP-complet, c'est à dire qu'il est dans la catégorie des problèmes NP et dans la catégorie de problème NP-difficile. Un problème est dit NP quand il est possible de le résoudre en temps polynomial par une machine de Turing non déterministe. Un problème est NP-difficile quand il est au moins aussi difficile que les autres problèmes de la classe NP (plus formellement que tous les problèmes de la classe NP se ramènent à celui-ci via une réduction polynomiale). Enfin, un problème NP-complet appartient à NP et est NP-difficile. La figure 2.2 donne une représentation graphiques des différentes catégories de problèmes.

Il n'est donc pas possible de résoudre en un temps raisonnable une grande instance de ce problème avec un algorithme déterministe. Si on arrive à associer notre problème avec le problème du sac à dos, nous pourrons affirmer que la solution optimale sera complexe à trouver et qu'il sera préférable d'avoir une bonne heuristique ou mieux, d'utiliser un algorithme d'approximation.

Avec ce premier modèle, la relation avec le problème du sac à dos est assez facile puisqu'on peut considérer que la capacité du sac est notre surface disponible dans un FPGA, que les objets sont les tâches de traitement avec comme valeur r_i et comme masse a_i : nous pouvons utiliser les travaux faits sur le problème du sac à dos.

Malheureusement, avec cette modélisation les résultats obtenus n'étaient pas applicables matériellement. Ils étaient cohérents avec le modèle mais pas du tout avec la réalité physique du traitement numérique du signal. Notre modèle était donc trop simple et trop abstrait.

C'est pourquoi nous avons dû modifier le modèle. Ces solutions incohérentes provenaient du fait que nous ne tenions pas compte de la taille des données en entrée et en sortie de chaque traitement. Comme nous travaillons avec des FPGAs, la taille des données n'est pas nécessairement alignée sur 8, 16, 32 ou 64 bits comme on pourrait le penser dans un contexte de processeur généraliste (CPU), mais les données peuvent être de taille arbitraire. Or cette taille influe sur la place qu'occupe le traitement.

Considérons une moyenne glissante $y = \frac{1}{N}\sum_{k=1}^{N} x_k$ sur N éléments x_k codés chacun sur p bits. Alors y est codé sur $p + \log_2(N)$ bits puisque nous faisons N accumulations. Si nous faisons l'hypothèse que le nombre de cellules logiques est proportionnel au nombre de bits représentant les données traitées (cas par exemple de l'additionneur avec autant d'étages de *Full Adder* que de bits à traiter), alors la consommation de ressources dans le FPGA augmentera en fonction de y.

Nous avons donc fait évoluer notre modèle pour tenir compte de cette contrainte. En plus des paramètres présentés dans l'équation 2.1, nous avons rajouté π_i^+ et π_i^- qui représentent respectivement la taille des données en sortie et en entrée de la tâche i ce qui nous donne :

$$\begin{cases} a_i \\ r_i \\ \pi_i^+ \\ \pi_i^- \end{cases} \tag{2.2}$$

À partir de la seconde version de notre modèle, nous avons essayé de définir ce que pouvaient être tous les paramètres et toutes les contraintes génériques applicables à toutes nos tâches de traitement. Nous allons donc, dans un premier temps, avoir le niveau d'abstraction le plus élevé possible afin d'avoir le plus de liberté pour répondre à toutes les contraintes critiques de notre système. C'est seulement après avoir obtenu ce modèle que nous serons en mesure de savoir si celui-ci semble fiable en prenant quelques exemples simples et en les confrontant à la réalité après synthèse.

Dans notre second modèle 2.2, une tâche générique est au moins composée de a_i, r_i, π_i^+ et π_i^- qui correspondent respectivement à la surface occupée par le bloc, la réjection du traitement, la taille des données en sortie et la taille des données en entrée. À ces différents paramètres s'ajoute un ensemble $\mathcal{P}_i$ qui représente l'ensemble des paramètres spécifiques à un traitement. Pour reprendre notre exemple de moyenne glissante, un paramètre spécifique à ce traitement serait le nombre de valeurs à accumuler avant de

faire la moyenne. Ce paramètre n'a de sens que dans ce traitement en particulier et n'a pas lieu d'être généralisé à tous les traitements, au contraire de a_i par exemple qui lui est commun à tous les traitements.

Nous constatons que nous ne pouvons pas exprimer formellement les contraintes sur a_i, r_i et π_i^+ car elles sont spécifiques à chaque traitement, on peut juste affirmer qu'elles dépendront de $\mathcal{P}_i$ et de π_i^-. Néanmoins, avec ces considérations, nous sommes capables de modéliser des traitements simples.

Si nous prenons le cas de la décimation, notre ensemble $\mathcal{P}_i$ contiendrait uniquement un paramètre D_i qui serait le facteur de décimation et on obtiendrait donc le modèle suivant :

$$\begin{cases} \mathcal{P}_i & = \{D_i\} \\ a_i & = \log_2(D_i) \\ r_i & = -10\log_{10}(D_i) \\ \pi_i^+ & = \pi_i^- \end{cases} \tag{2.3}$$

La tâche n'occupe que la place nécessaire pour compter le nombre de données, l'opération va aussi induire du repliement spectral, donc du bruit, qui dépendra de D_i. r_i a donc une valeur négative. La taille des données reste inchangée durant l'opération.

Un autre exemple simple que nous pouvons prendre est le cas de l'oscillateur numérique NCO. Les seuls paramètres qui nous intéressent dans notre cas sont la taille des données en sortie π_i^{out} et N le nombre de données mémorisées pour générer le signal. Ces deux paramètres appartiennent donc à $(P)_i$ et avec eux on obtient le modèle suivant :

$$\begin{cases} \mathcal{P}_i & = \{\pi_i^{out}, N\} \\ a_i & = \pi_i^{out} \times N \\ r_i & = 0 \\ \pi_i^+ & = \pi_i^{out} \end{cases} \tag{2.4}$$

La taille qu'occupe le NCO ne compte que la place nécessaire pour stocker le signal précalculé, il n'ajoute ni n'enlève de bruit car le signal généré est, par définition, parfait et enfin, la taille des données en sortie ne dépend que du paramètre que l'utilisateur définit selon ses besoins.

Comme nous avons pu le voir, ce modèle semble bien fonctionner pour des petits opérations. Afin de pouvoir donner le modèle d'une chaîne de traitement, nous allons devoir décrire comment ces différentes tâches interagissent entre elles.

2.3 Modélisation d'une chaîne de traitement

Une chaîne de traitement est une suite de tâches élémentaires. Dans la section précédente, nous avons défini la modélisation de l'une de ces tâches. Ici, nous donnerons la modélisation d'un ensemble de tâches.

Dans la section 1.1.3 du chapitre, nous citions Benkrid *et al.* dans [10] qui disaient qu'on pouvait associer un flux de traitement à un DAG. Cependant, dans un DAG, il est possible qu'une tâche (un nœud) soit précédée de plus d'une autre tâche. En l'état, la modélisation des traitements n'est pas adaptée à ce genre de situation car nous serions incapables de savoir quels sont les traitements précédant ou suivant une tâche donnée. Toutefois, la partie que l'on souhaite optimiser (le cadre rouge de la figure 2.1) ne contient aucune tâche avec plus d'un prédécesseur : notre modèle reste donc valable.

Pour que le système reste cohérent, il faut définir un certain nombre de contraintes. Tout d'abord, il faut s'assurer que la taille des données soit cohérente. Nous définissons donc la contrainte suivante :

$$\pi_i^+ = \pi_j^- \tag{2.5}$$

Soit i une tâche et j la tâche suivante, alors l'équation 2.5 nous garantit que la taille des données en sortie d'une tâche est bien la même que celle de la tâche précédente. Nous n'aurons pas d'incohérences sur la taille des données au milieu du traitement.

La prochaine contrainte à définir concerne la place qu'occupe la chaîne de traitement. Si celle-ci est constituée de n tâches alors :

$$\sum_{i=1}^{n} a_i = A \tag{2.6}$$

L'équation 2.6 définit la taille totale A de la chaîne de traitement comme étant la somme de toute les surfaces des tâches qui la composent. Pour rappel, dans notre modèle la surface occupée est exprimée en u.a. pour nous donner plus de liberté. Nous faisons ici l'hypothèse que l'outil de synthèse est incapable d'avoir une vision globale du traitement et de factoriser des éléments de blocs différents : c'est peut être la principale différence entre les outils de synthèse pour FPGA actuels et les compilateurs pour processeurs généralistes.

La dernière contrainte que nous allons évoquer concerne la définition de la réjection globale. En reprenant n comme le nombre total de tâches différentes alors :

$$\sum_{i=1}^{n} r_i = R \tag{2.7}$$

Cette contrainte définit donc la réjection totale du traitement comme étant la somme individuelle de chacune des tâches qui la composent. Rappelons que, r_i peut être négatif dans le cas où le traitement introduit du bruit.

Avec ces trois contraintes, nous sommes capables d'exprimer notre chaîne de traitement par une succession de modèles de tâches élémentaires. Néanmoins, il faut encore donner le modèle d'un filtre FIR, nous serons capables alors d'exprimer l'intégralité de la chaîne de traitement.

2.4 Modélisation d'un filtre FIR

Comme nous l'avons dit dans la section 2.2, nous cherchons à optimiser une opération de filtrage. Plus spécifiquement, nous allons chercher à mettre plusieurs filtres les uns après les autres. Nous avons déjà vu comment modéliser une telle chaîne, il ne reste qu'à définir complètement le modèle d'un filtre.

Un filtre i dispose de deux paramètres intrinsèques C_i et π_i^C qui représentent respectivement le nombre de coefficients utilisés et le nombre de bits sur lesquels ils sont codés. Nous aurions pu être plus précis en donnant non pas seulement le nombre de coefficients mais l'ensemble des coefficients eux-mêmes. Ce faisant, nous pourrions connaître plus finement le comportement du filtre mais cela complexifierait énormément le modèle et rajouterait un nombre considérable de paramètres. Pour rester le plus simple possible nous avons préféré ne considérer que le nombre de coefficients et leur nombre de bits.

Un filtre est aussi caractérisé par l'espace a_i qu'il occupe. Comme nous l'avons dit précédemment, nous faisons l'hypothèse que notre filtre n'occupe qu'une surface arbitraire suffisante pour stocker ses coefficients et opérations. Ainsi, nous gardons un niveau d'abstraction plus élevé et on a plus de degrés de liberté pour le choix d'une solution optimale. De plus, prendre en considération tous les composants réellement consommés dans le FPGA alourdirait le modèle et par conséquent demanderait plus de temps de calcul lors de la recherche de solutions optimales. Pour toutes ces raisons, nous avons décidé qu'il était préférable de considérer uniquement la surface de silicium utilisée et de ne garder qu'une représentation arbitraire. Une telle abstraction nous permet de changer la plateforme matérielle facilement, les solutions optimales trouvées seront donc applicables à n'importe quel FPGA. Dans la section 4.2 nous appliquerons cette modélisation et nous vérifierons que notre approche est juste.

Nous définissons également la taille des données en sortie π_i^+ comme étant la somme de la taille en entrée π_i^- et de la taille des coefficients. En toute rigueur, nous devrions définir $\pi_i^+ = \pi_i^- + \pi_i^C + \lceil \log_2(C_i) \rceil$ puisque lors du produit de convolution nous faisons une accumulation. Toutefois, cette estimation n'est vraie que dans le pire des cas, à savoir quand tous les coefficients et les données en entrée sont à leur valeur maximum. La figure 2.3 représente un jeu de coefficients quelconque. On peut remarquer que seul semble se démarquer le maximum des coefficients. C'est pourquoi, notre estimation ne tient compte que de ce maximum et non de la taille théorique possible. Ce faisant, nous évitons d'avoir des données trop grandes en gardant à l'esprit que le filtre va ajouter des bits tout au long du traitement.

Pour finir, nous définissons la réjection comme étant une fonction $\mathcal{F}$ qui prend en paramètre C_i et π_i^C. Il est compliqué de donner cette fonction puisque que la réjection va dépendre des coefficients et qu'il existe de nombreuses méthodes pour les générer comme nous l'avons dit dans la section 1.3.3 du chapitre précédent. Nous verrons dans le chapitre suivant, section 3.1, comment pallier ce problème.

Au final, le système 2.8 donne le modèle abstrait que nous allons utiliser pour optimiser la chaîne de traitement.

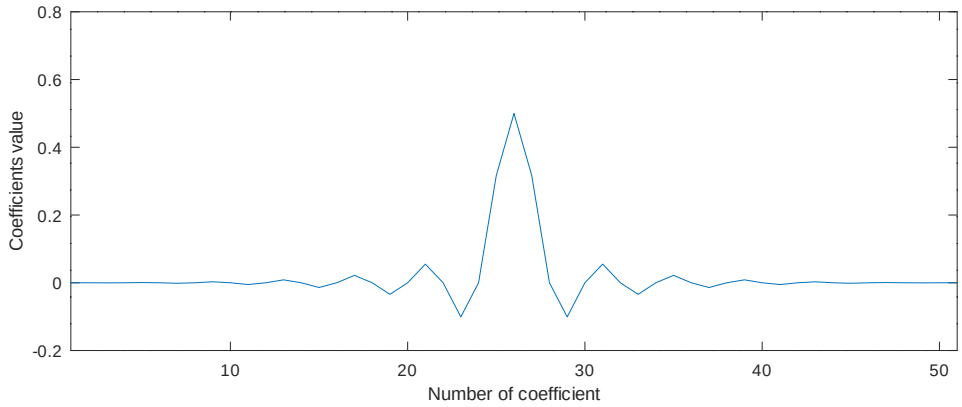


FIGURE 2.3 – Exemple d'un jeu de coefficients quelconque

$$\begin{cases} \mathcal{P}_i & = \{C_i, \pi_i^C\} \\ a_i & = C_i \times (\pi_i^C + \pi_i^-) \\ \pi_i^+ & = \pi_i^- + \pi_i^C \\ r_i & = \mathcal{F}(C_i, \pi_i^C) \end{cases} \tag{2.8}$$

Ce nouveau modèle respecte également l'architecture matérielle des filtres que nous allons utiliser dans le FPGA. Nous souhaitons traiter les données en temps-réel, nous utilisons donc un filtre pipeliné pour les traiter. La figure 2.4 schématise le fonctionnement du bloc dans le FPGA et la figure 2.5 donne le chronogramme associé. On peut voir que les données en entrée sont de taille π_i^- et les coefficients sont de taille π_i^C. Bien que les calculs intermédiaires soient stockés sur le nombre de bits théorique maximum, à la sortie du filtre nous ne gardons que π_i^+ bits. Le fonctionnement général du filtre est de traiter la données courante, de l'accumuler et de retarder sa propagation à l'étage suivant d'un coup d'horloge. Ainsi, si on a suffisamment d'étage (C_i étages), nous sommes capables de traiter tout le flux.

Grâce à cette nouvelle modélisation d'un filtre FIR nous sommes capables de l'intégrer au modèle de chaîne de traitement présenté dans la section précédente. Cependant, cela ne sera pas suffisant pour définir une cascade de filtres dans le contexte particulier des FPGA. En l'état, la modélisation n'est pas contrainte et rien ne garanti que les résultats optimaux obtenus soient cohérents avec une implémentation réelle sur FPGA. Pour nous le garantir, il faut renforcer notre modélisation en apportant un ensemble de contraintes liées au contexte.

Dans le prochain chapitre, nous allons passer à la deuxième étape de notre modèle : la recherche d'une solution optimale. Nous verrons comment nous instancions ce modèle et les contraintes que cela soulève. Puis nous détaillerons la méthode que nous avons choisie pour trouver la solution optimale.

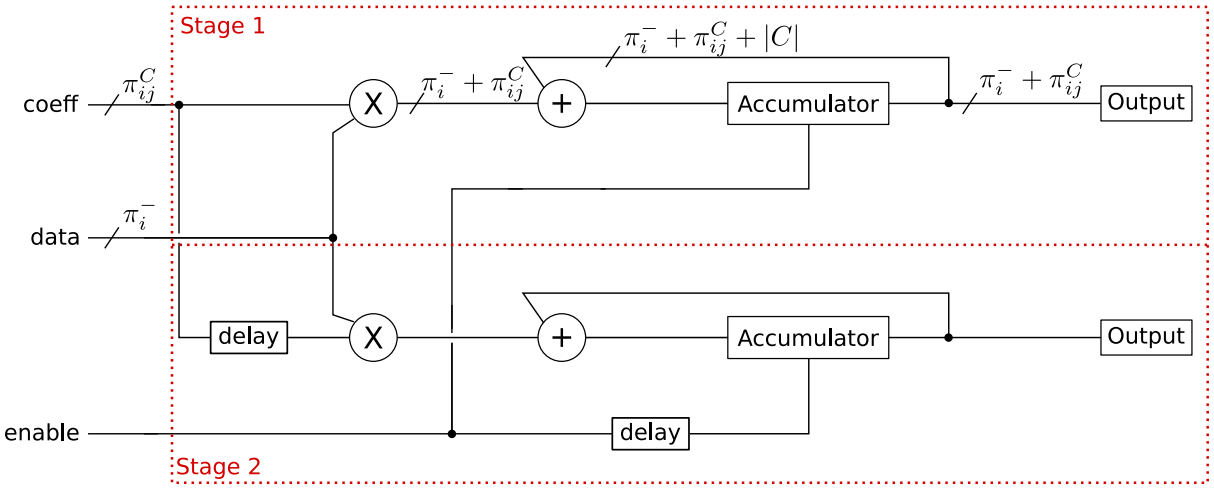


FIGURE 2.4 – Schéma de l'architecture matériel d'un filtre pipeliné

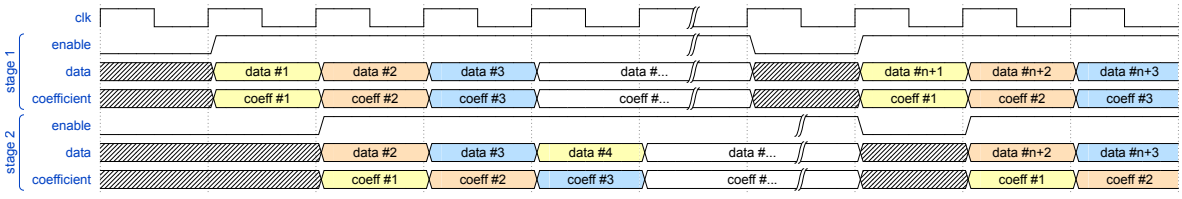


FIGURE 2.5 – Chronogramme du fonctionnement d'un filtre pipeliné

Chapitre 3

Analyse de la cascade de FIR

Dans le chapitre précédent, nous avons présenté notre méthodologie et nous l'avons appliquée pour optimiser le filtrage de données dans une application exécutée sur FPGA. La première étape a consisté à définir un modèle abstrait du problème. Dans les sections 2.2 et 2.3, nous présentions et détaillions le modèle qui va permettre de résoudre ce problème d'optimisation.

Ce chapitre sera consacré à la deuxième étape de notre méthodologie : la recherche d'une solution optimale à partir du modèle. Néanmoins, pour rechercher une solution optimale, nous avons besoin d'instancier notre modèle et cela nécessite d'être détaillé.

Intuitivement, nous pourrions arriver à la conclusion que l'idéal serait de mettre le plus possible de filtres avec beaucoup de coefficients codés sur beaucoup de bits. Toutefois, nous verrons que ce n'est pas forcément la meilleure solution et qu'il existe une configuration optimale qui sélectionne un nombre de coefficients et une nombre de bits idéal. à cela nous ajouterons la contrainte que la place dans le FPGA n'est pas illimitée et qu'il faudra choisir judicieusement les filtres.

Nous traiterons un problème complémentaire au nôtre : comment minimiser la consommation de ressources avec un objectif de réjection minimal. Ainsi, nous pourrons voir quel est le meilleur compromis entre la réjection et la place occupée dans le FPGA.

Dans la section 3.1, nous reviendrons sur l'expression de la réjection d'un filtre et les contraintes de son implémentation dans les FPGA. Puis dans la section 3.2, nous définirons les contraintes ajoutées à notre modèle pour tenir compte de son instanciation et des contraintes liées au FPGA. Enfin, dans la section 3.3, nous détaillerons la technique que nous utiliserons pour trouver des solutions optimales.

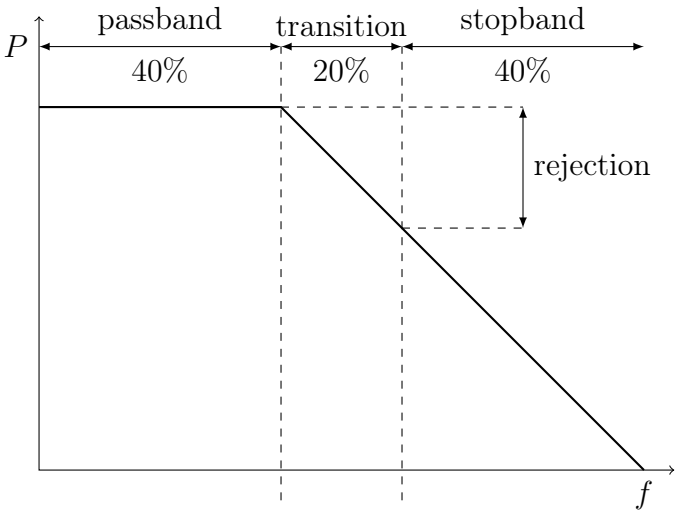


FIGURE 3.1 – Gabarit des filtres que nous imposons arbitrairement

3.1 Évaluation de la performance d'un filtre

Dans la section 2.4, nous donnions le modèle général d'un filtre FIR. Nous avions expliqué qu'il était compliqué d'exprimer la relation entre la réjection et le jeu de coefficients car cela dépendait, entre autres, de la façon dont ils étaient générés. Nous allons ici, exposer comment nous évaluons la performance d'un filtre donné.

Concentrons nous dans un premier temps sur l'évaluation du FIR. Différents critères sont classiquement considérés comme, par exemple, les fluctuations dans la bande passante, la vitesse de coupure et la réjection. Compte tenu de notre objectif de synthétiser une chaîne de filtrage visant à empêcher le repliement spectral du bruit avant décimation, nous imposerons la bande de coupure, qui n'est donc pas un paramètre libre.

La figure 3.1 montre le gabarit visé pour les filtres passe-bas afin d'éliminer les hautes fréquences du signal. Nous définissons la largeur de la bande passante comme étant les 40% des composantes spectrales aux fréquences les plus basses, et la bande de réjection comme étant les 40% des composantes spectrales aux fréquences les plus hautes. La bande de transition sera sur les 20% de fréquences centrales restantes.

Cependant cela ne suffit pas à qualifier la réjection d'un filtre. En effet, la réponse impulsionnelle du filtre va impacter les composantes spectrales à différentes fréquences. La question est donc de trouver comment calculer une unique valeur permettant d'estimer la performance globale du filtre.

Une première idée naïve pour estimer la qualité d'un filtre a été de regarder la quantité moyenne de bruit rejetée dans la bande de coupure *(stopband)*. La figure 3.2 illustre deux filtres : l'un est très mauvais et l'autre bien meilleur. Pour qualifier la réjection d'un filtre, nous considérons qu'il s'agit de la moyenne de réjection dans la bande de coupure. Cette évaluation est représenter par les droites pointillées. La première conclusion qu'on peut tirer de cette figure est qu'on surestime la qualité du filtre. En effet, la courbe bleue, qui

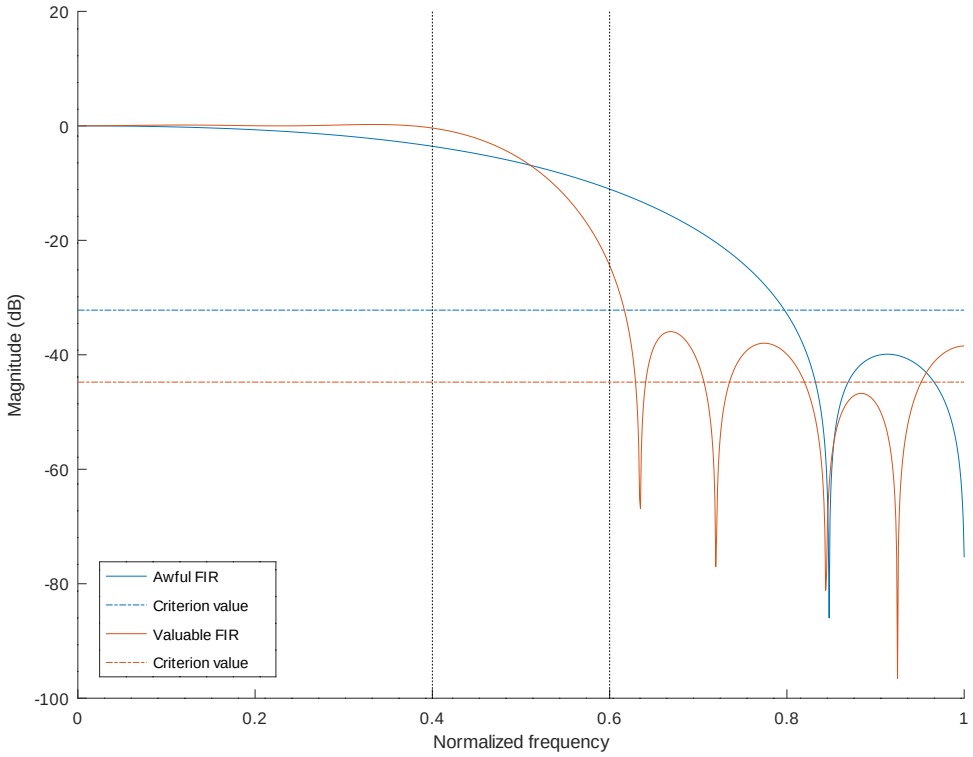


FIGURE 3.2 – Qualification de deux filtres avec la moyenne comme critère

représente le plus mauvais filtre, indique qu'il est capable de rejeter plus de 30 dB de bruit. Certes le rebond le plus bas se situe à environ -40 dB mais la majeure partie de la courbe est au-dessus de -30 dB. De la même façon, pour la courbe rouge, le critère nous indique que ce filtre est capable de rejeter près de 45 dB de bruit. Cet effet s'explique simplement par la présence de puits (*notch* en anglais) qui tendent vers $-\infty$. La moyenne sera donc très fortement affectée par ces valeurs extrêmes.

Constatant ce problème, nous avons décidé de changer de critère de réjection en prenant, cette fois, la médiane de la bande de coupure. La figure 3.3 représente les mêmes filtres que précédemment mais en considérant la médiane comme valeur de réjection. Nous constatons que calculer la médiane n'est pas la bonne solution. En effet, bien que l'évaluation de la réjection du bon FIR se soit un peu rapprochée de la réalité, la réjection du mauvais FIR s'est encore plus éloignée de la réalité qu'avec la moyenne comme critère. Cette fois encore, l'impact s'explique par la présence des *notch* qui alourdissent la médiane.

Notre tentative suivante fut de ne conserver que la réjection minimale en bande de coupure. Ce faisant, nous nous assurons que dans le pire des cas la réjection sera au moins de cette valeur, même si la majeure partie des composantes spectrales en bande de coupure est mieux rejetée que la composante spectrale avec la plus mauvaise réjection.

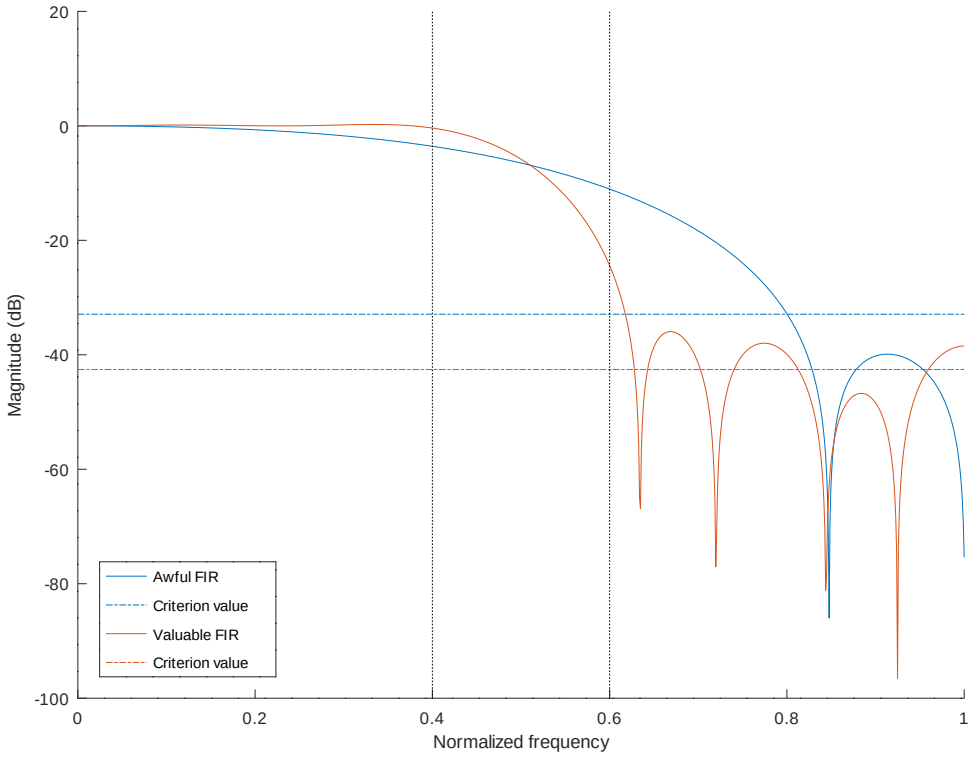


FIGURE 3.3 – Qualification de deux filtres avec la médiane comme critère

La figure 3.4 représente l'application de notre nouveau critère avec les mêmes filtres que les deux figures précédentes. Cette fois-ci, on peut constater que le choix de ce critère semble donne une bien meilleure estimation de la réjection. En effet, le plus mauvais filtre ne rejette qu'environ 10 dB alors que le meilleur des filtres rejette environ 25 dB. Dans le cas du meilleur filtre, on constate également que la très grande majorité des composantes spectrales dans la bande de coupure est bien en dessous de du critère. En raisonnant ainsi, nous considérons seulement le pire des cas et lorsque nous utiliserons ce critère, nous sommes assurés d'avoir au pire cette valeur de réjection même s'il se peut que la réjection soit meilleure dans les faits.

Malgré cette nette amélioration, il reste un paramètre à prendre en compte : le comportement dans la bande passante *(passband)*. Dans les critères précédents, nous n'en tenions pas compte. Pourtant, il est important de préserver au mieux cette partie du spectre. En effet, c'est ici que sont situées les informations que nous cherchons à analyser. Si nous rejetons une partie de la bande passante, nous perdons alors des informations utiles et au contraire si nous l'amplifions, nous ajoutons des gains superflus.

Pour prendre en compte les déviations dans la bande passante, nous pénalisons la réjection globale en soustrayant l'amplitude des fluctuations des composantes spectrales (*ripples)* dans cette bande. Ce faisant, nous diminuons la valeur de réjection en tenant

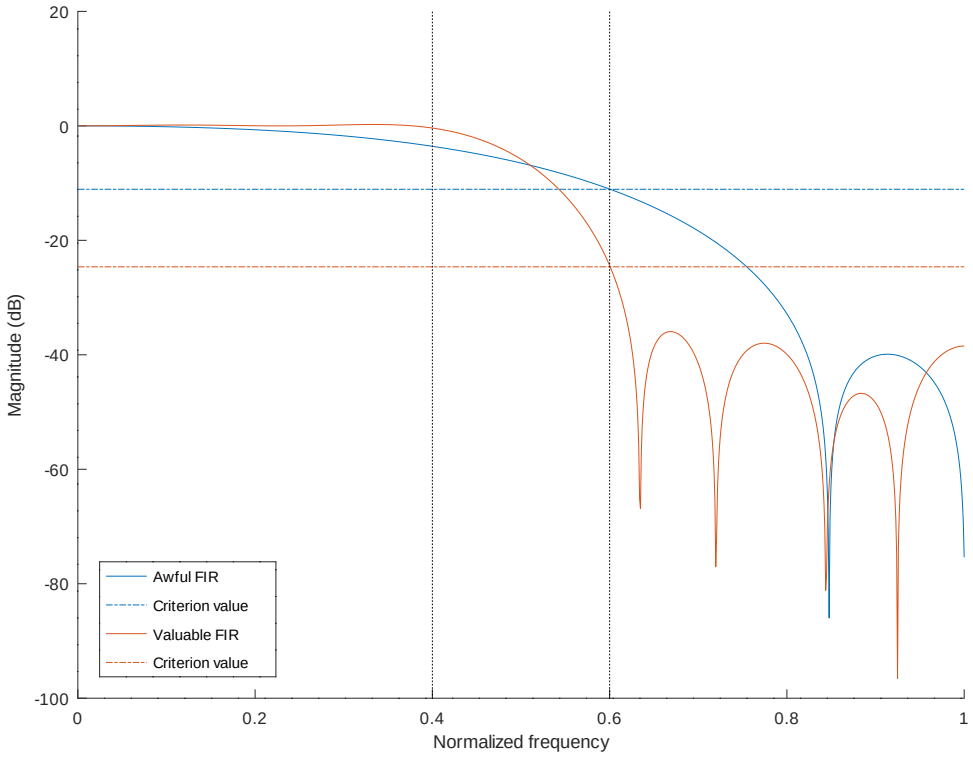


FIGURE 3.4 – Qualification de deux filtres avec la valeur maximum en stopband comme critère

compte de la stabilité en bande passante. Dans le cas où le filtre est bon, cela n'aura pas beaucoup d'incidance, mais dans le cas où le filtre est mauvais, notamment quand il y des déviations dans la bande passante, cela l'impactera plus fortement afin de privilégier les filtres les plus stable dans la bande passante. La figure 3.5 montre donc la fusion du critère précédent (prendre seulement le maximum de réjection dans la bande de coupure) et le critère sur la bande passante. Toujours en utilisant les deux mêmes jeux de coefficients que pour les figures précédentes. Comme nous pouvons le constater dans le cas où le filtre n'est pas très performant, la valeur de réjection est pénalisée à cause de l'écart qu'on peut constater dans la bande passante alors que le filtre ayant un bon niveau de réjection est, quant à lui, très peu pénalisé car il n'a que très peu de variation en bande passante. Ce critère impactera notamment l'élimination de filtres présentant d'excellentes réjections en bande de coupure mais une transition trop lente induisant une atténuation excessive dans la bande passante.

Maintenant que nous avons défini notre critère, nous allons générer un ensemble de jeux de coefficients et nous les évaluerons. Pour commencer, nous faisons appel aux fonctions de GNU Octave (l'alternative open-source de Matlab) `fir1` et `firls` pour créer les jeux de coefficients en faisant varier le nombre de coefficients. Le but n'est pas d'avoir des coefficients optimaux mais d'en avoir un nombre suffisant pour laisser une grande liberté

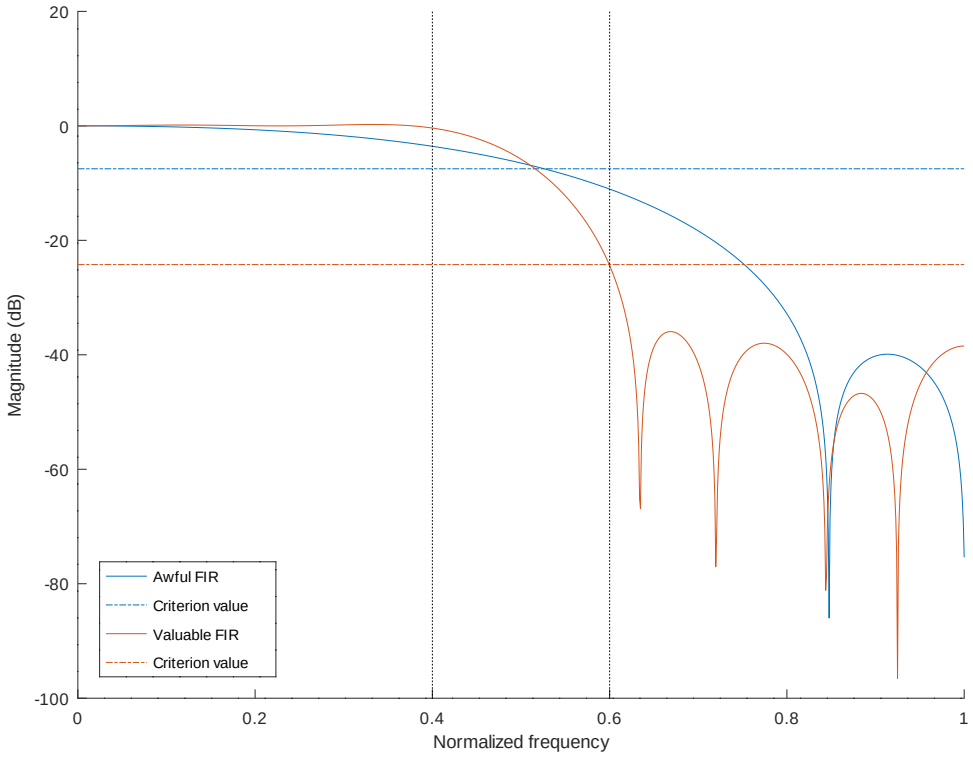


FIGURE 3.5 – Qualification de deux filtres avec la fusion de nos deux critères

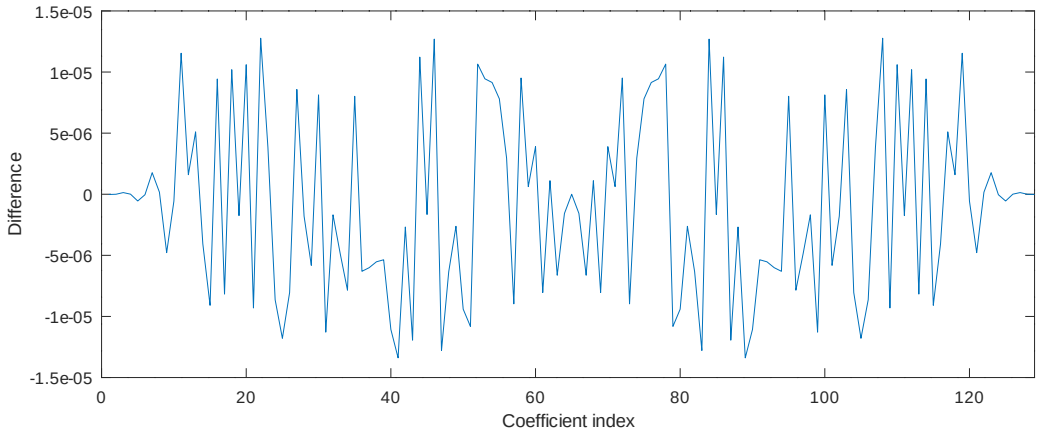
de choix au modèle. C'est pourquoi les fonctions `fir1` et `firls` sont sélectionnées, même s'il est tout à fait possible d'utiliser d'autres fonctions pour générer d'autres coefficients plus performants pour des situations plus spécifiques comme nous l'avons détaillé dans la section 1.3.3 du chapitre 1.

Les filtres ainsi générés auront un nombre différent de coefficients représentés par des nombres à virgule flottante. Cela pose un problème car les FPGA ne peuvent pas tous faire des opérations à virgule flottante et ceux qui le peuvent ont des performances moins bonnes que celles obtenues avec les opérations sur des nombres entiers. Il faut donc convertir ces jeux de coefficients en nombres réels en coefficients en nombres entiers. Pour calculer les valeurs entières, nous procédons en deux étapes :

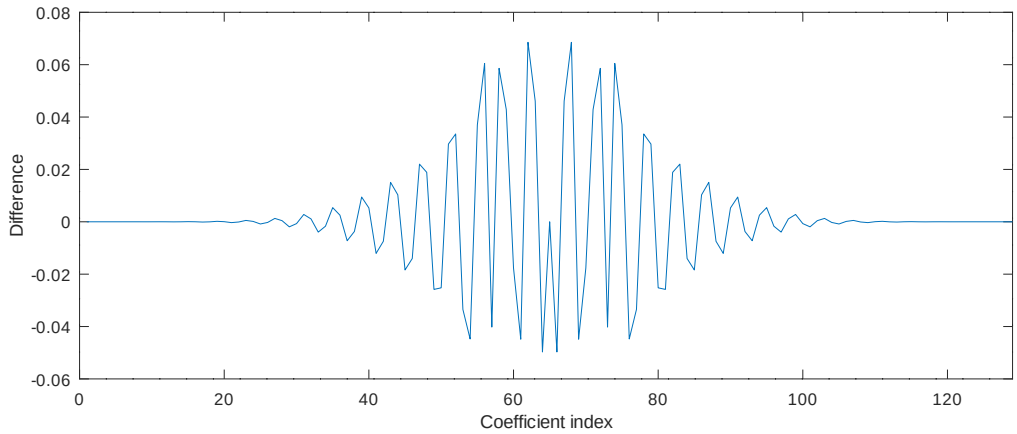
1. on normalise le jeu de coefficients pour que toutes les valeurs soit bornées dans l'intervalle $[-1, 1]$
2. on multiplie les coefficients par la valeur maximale entière signée possible pour le nombre n de bits voulu et on arrondit au nombre le plus proche. Ainsi, les valeurs sont bornées entre $[-2^{n-1}, 2^{n-1} - 1]$

Toutefois, cette conversion va impacter la réjection du filtre.

Dans la figure 3.6a nous avons pris un jeu de coefficients donné en nombre réel, puis on le convertit en entier signé codé sur 16 bits. On normalise les points par la valeur



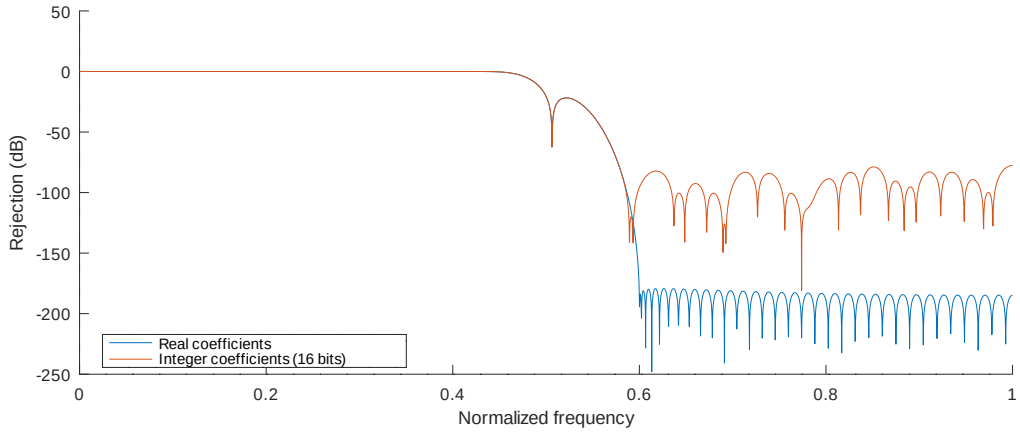
(a) Différence entre des coefficients de filtres codés avec des nombres à virgule flottante et des nombres entiers sur 16 bits



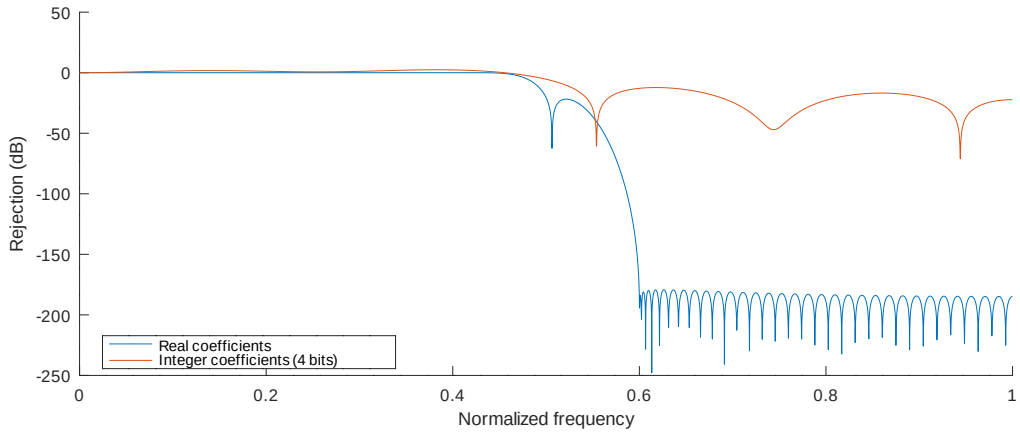
(b) Différence entre des coefficients de filtres codés avec des nombres à virgule flottante et des nombres entiers sur 4 bits

FIGURE 3.6 – Différence dans le jeu de coefficients entre les valeurs réelles et les valeurs entières

maximale puis on trace la différence entre les coefficients réels et entiers. La figure 3.6b suit le même principe sauf qu'on convertit les coefficients en entiers signés codés sur 4 bits. On constate que la différence de valeur n'est pas très grande dans les deux cas : de l'ordre de 10^{-6} pour les entiers 16 bits et 10^{-2} pour ceux sur 4 bits. Les entiers sur 16 bits sont cependant 10 000 fois plus proches de la valeur réelle que les entiers sur 4 bits. Ces différences entre les valeurs en virgule flottante et les valeurs converties a un impact significatif sur la réponse du filtre. Les figures 3.7a et 3.7b reprennent les mêmes coefficients que précédemment, mais cette fois nous traçons la réponse impulsionnelle du filtre. Comme on peut le voir, la réjection de base du filtre avec les coefficients réels est d'environ 170 dB, celle avec les entiers codés sur 16 bits de 80 dB et pour celle avec les entiers sur 4 bits elle est de 10 dB. Même avec une petite différence (de l'ordre de 10^{-6}), on perd 90 dB de réjection et on perd 160 dB avec la version sur 4 bits. Il nous faudra donc bien tenir compte de ces différences pour choisir les filtres à utiliser dans le traitement.



(a) Différence entre des nombres à virgule flottante et des nombres entiers sur 16 bits

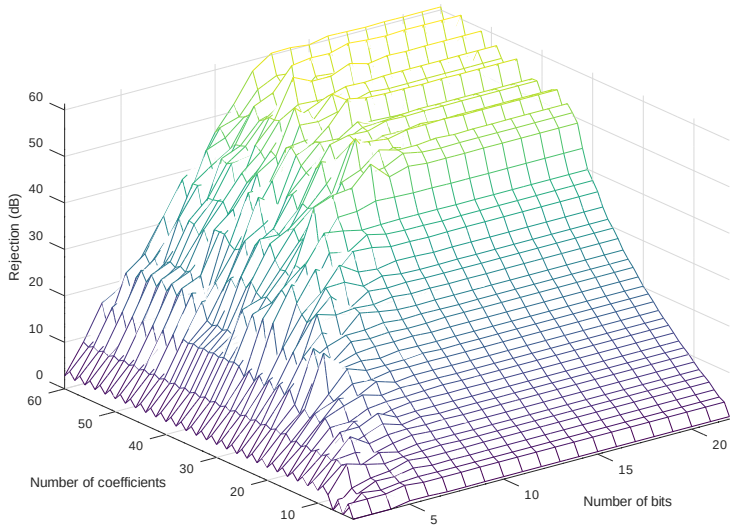


(b) Différence entre des nombres à virgule flottante et des nombres entiers sur 4 bits

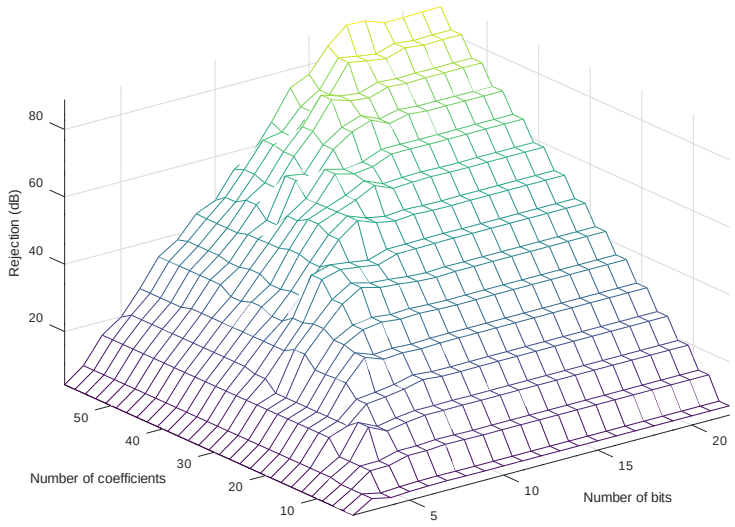
FIGURE 3.7 – Différence de rejection entre les valeurs réelles et les valeurs entières

Afin d'avoir un choix important de filtres, nous avons généré des filtres comportant de 3 à 60 coefficients, codés sur 2 à 22 bits. Pour chacune de ces configurations, nous avons tracé leur réjection selon le critère défini ci-dessus. La figure 3.8a montre les réjections obtenues pour les coefficients générés avec `fir1` et la figure 3.8b celles avec la fonction `firls`. On peut voir qu'avec `fir1`, la réjection maximale est moins haute (environ 60 dB contre plus de 80 dB avec `firls`) mais la progression est plus rapide sur les premiers filtres avec une sorte de palier à partir de 30 coefficients.

On constate également que ces deux figures ont la forme de pyramide : plus on s'approche du sommet, meilleure sera la réjection. Autrement dit, plus on a de coefficients codés sur beaucoup de bits, meilleure sera la réjection. Cependant, on peut voir aussi qu'il n'est pas forcément utile d'avoir trop de bits pour un nombre de coefficients donné ou, inversement, il n'est pas utile d'avoir trop de coefficients pour un nombre de bits donné. Prenons par exemple la figure 3.8b. Si on a 10 coefficients, au delà de 8 bits, la réjection ne semble pas évoluer. Et si on regarde pour 8 bits au delà de 30 coefficients on n'obtient pas une meilleure réjection. On peut supposer qu'il y a des compromis optimaux entre nombre de coefficients et nombre de bits qui sont donnés par les arrêtes des pyramides



(a) Filtres générés avec `fir1`

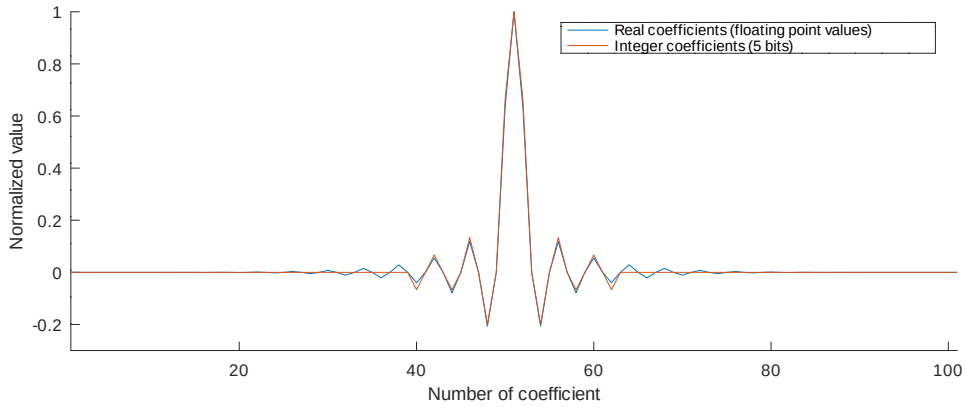


(b) Filtres générés avec `firls`

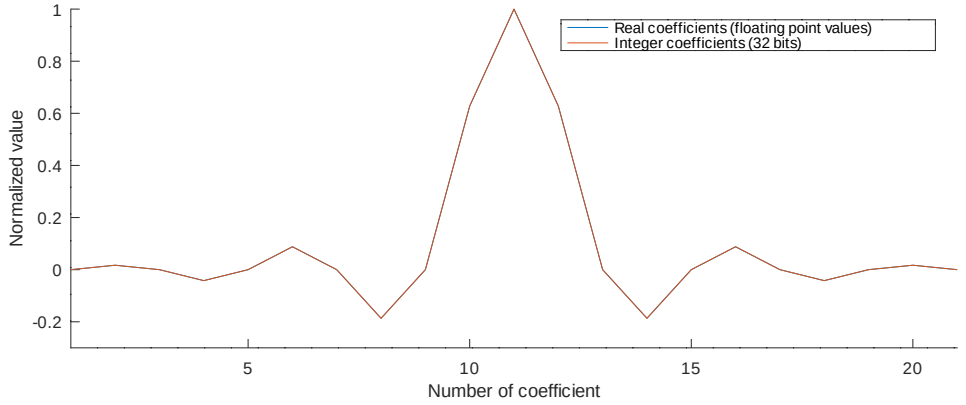
FIGURE 3.8 – Réjection des filtre en fonction du nombre de coefficients et du nombre de bits

La figure 3.9 nous permet de mieux comprendre pourquoi il est inutile d'avoir trop de coefficients ou trop de bits. Elle représente les deux cas :

1. on prend trop de coefficients (figure 3.9a)
2. on prend trop de bits (figure 3.9b)



(a) Cas où le nombre de coefficients est trop grand



(b) Cas où le nombre de bits est trop grand

FIGURE 3.9 – Impact de la conversion des coefficients en nombres entiers

Dans le premier cas, on peut voir que la conversion en nombre entier laisse de nombreux coefficients à zéro. Ajouter encore d'autres coefficients ne ferait qu'ajouter plus de zéro sans améliorer la réjection pour autant. Dans le second cas, la conversion en nombre entier est déjà bien assez proche des valeurs attendues. Rajouter plus de bits réduirait la différence entre les valeurs réelles et les valeurs entières mais ce changement ne serait pas significatif (ainsi qu'on peut le constater sur les deux pyramides).

Grâce à ce critère, nous sommes capables d'estimer la réjection des différents filtres que nous voulons utiliser dans le traitement. À défaut de connaître la fonction $\mathcal{F}(C_i, \pi_i^C)$ qui définit la réjection pour un filtre donné, nous sommes tout de même capables d'en donner une valeur pour un couple donné. Il reste néanmoins un problème. Puisque nous avons deux fonctions pour générer les jeux de coefficients, pour chaque couple $\{C_i, \pi_i^C\}$ nous avons deux réjections possibles (une pour `fir1` et l'autre pour `firls`). Pour être précis, nous avons dû rajouter une variable à l'ensemble $\mathcal{P}_i$ pour garder en mémoire comment les filtres ont été créés. Par souci de clarté, nous omettrons ce détail car cela n'influe pas

sur notre approche. Dans la section 3.3, nous expliquerons pourqoi nous sommes obligés de prendre des configurations des filtres car nous ne sommes pas en mesure d'estimer la réjection de façon générique. Les valeurs de réjection deviendront alors des constantes ce qui nous permettra de linéariser notre programme.

3.2 Définition formelle des problèmes

Nous venons de voir comment calculer la réjection pour chacun des filtres. Dans le chapitre précédent nous avions également vu comment était modélisée une chaîne de traitement. Étant donné que nous travaillons avec des FPGA, nous avons la possibilité de manipuler des entiers de tailles non alignées sur 8, 16, 32 ou 64 bits. Il est donc possible d'économiser de la place en ne gardant que le strict minimum de données.

Nous allons adapter notre chaîne pour rajouter des décalages logiques entre les différents filtres. Ainsi, nous pourrons diminuer le nombre de bits utilisés entre les étages et ainsi minimiser la consommation de ressources. Toutefois cela va nécessiter d'adapter le modèle pour qu'on puisse choisir la taille des données en sortie de chaque filtre.

Nous avons défini deux problèmes distincts :

1. comment maximiser la réjection avec une surface de FPGA limitée ?
2. comment minimiser la consommation de surface tout en atteignant un seuil de réjection ?

Bien que ces deux problèmes soient proches, ils différent au niveau de leurs contraintes et objectifs. Nous les considérons comme complémentaires car grâce à eux, nous serons capables de faire de l'optimisation multi-objectifs (minimiser la consommation de ressources et maximiser le bruit rejeté).

Quelle que soit la variante du problème, notre modèle doit représenter n étages de filtres successifs (dans le cas d'un filtre monolithique, $n = 1$). Chaque étage i ($1 \leq i \leq n$) sera composé d'un FIR et d'un décalage logique *(shift)*. Rappelons que chaque FIR est caractérisé par C_i le nombre de coefficients, et par π_i^C le nombre de bits. Le décalage logique quant à lui est représenté par π_i^S le nombre de bits supprimés par le décalage binaire. Lorsque nous parlerons d'un filtre, nous ferons référence à cette combinaison de FIR et de décalage logique comme s'il s'agissait d'un seul et même élément. Ainsi, nous nommons π_i^- la taille des données en entrée et π_i^+ la taille des données en sortie (après le décalage logique). La figure 3.10 schématise la définition des paramètres d'un étage de filtre.

Nous reprenons donc le modèle d'un FIR, présenté dans la section 2.4 du chapitre précédent, pour intégrer ce nouveau paramètre et pour définir les contraintes associées au système.

Puisque nous n'avons plus qu'un seul type de tâches à placer, nous savons que le tâche suivante du filtre i est celle à $i+1$ et que la précédente est à $i-1$. On peut donc remplacer la contrainte 2.5 par la contraintes 3.1. Nous nommerons Π^I la taille initiale en entrée

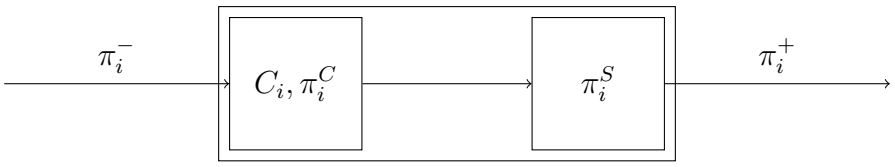


FIGURE 3.10 – Définition schématique d'un étage de filtrage (avec un FIR à gauche et un décalage logique à droite)

π_0^+. Dans les faits, il s'agit du nombre de bits effectif sur lequel le signal d'entrée est codé.

$$\forall i, \pi_i^- = \pi_{i-1}^+ \tag{3.1}$$

Comme nous avons ajouté un décalage logique à la sortie du filtre, la taille des données n'a plus la même expression. Nous définissons donc la taille des données sortantes par l'équation 3.2. Rappelons que nous considérons que l'impact du filtre sur la taille des données est limité qu'à la multiplication des coefficients. C'est pourquoi $\pi_{FIR}^+ = \pi_i^- + \pi_i^C$ et non $\pi_i^+ = \pi_i^- + \pi_i^C + \lceil \log_2(C_i) \rceil$.

$$\pi_i^+ = \pi_{FIR}^+ - \pi_i^S \iff \pi_i^+ = \pi_i^- + \pi_i^C - \pi_i^S \tag{3.2}$$

Pour finir, nous devons nous assurer que le décalage logique des données ne va pas induire plus de bruit que les filtres ne vont en rejeter. En effet, puisque décaler des bits va faire perdre des informations, il faut être sûr que l'information perdue ne soit pas critique. Pour déterminer quelle est cette limite, nous devons nous intéresser à la notion de ENOB vue à la section 1.2 du chapitre 1. Pour rappel, ENOB est l'acronyme de *Effective Number Of Bits* et il s'agit du nombre de bits contenant des informations utiles. Pour simplifier, on peut considérer qu'il s'agit des informations pertinentes dans les échantillons de notre signal. Si lors du décalage on venait à obtenir un nombre de bits plus petit que cet ENOB, le plancher de bruit qu'on pourrait mesurer serait rehaussé de 6 dB par bit manquant. Notre objectif sera donc de faire en sorte qu'on garde à chaque étage suffisamment de bits pour être au moins à la valeur du ENOB. Reste maintenant à savoir comment évolue l'ENOB au fur à mesure des filtres.

Dans notre cas, nous essayons de rejeter du bruit. Initialement, notre réjection est nulle, nous n'avons donc pas besoin du ENOB mais dès la sortie du premier étage nous avons une première réjection de bruit dont il faudra tenir compte. En effet, si avec 1 bit nous sommes capables de descendre jusqu'à 6 dB de réjection, nous pouvons exprimer notre ENOB ainsi :

$$\text{ENOB}_i = \left\lceil \frac{r_i}{6} + 1 \right\rceil \tag{3.3}$$

L'ajout d'un bit est nécessaire pour tenir compte du fait que les échantillons sont des valeurs entières signées.

Nous avons également dit que la taille des données en sortie doit être supérieure ou égale à l'ENOB, on pourrait donc dire que :

$$\pi_i^+ \geq \text{ENOB}_i \tag{3.4}$$

Cependant cela ne serait valable que pour le premier étage car pour les étages suivants, il faut tenir compte de la quantité de bruit déjà rejetée. La formule 3.4 devient donc :

$$\pi_i^+ \geq \sum_{k=1}^{i} \text{ENOB}_k \iff \pi_i^+ \geq \left\lceil \sum_{k=1}^{i} \frac{r_i}{6} + 1 \right\rceil \tag{3.5}$$

$$\tag{3.6}$$

Ainsi, nous sommes certains que le nombre de bits sera toujours suffisant pour ne pas influer sur le niveau de réjection de la cascade de filtres tout en gardant des tailles de données suffisamment petites.

Le système d'équations 3.7 est la version finale de notre modèle en ajoutant toutes les contraintes liées au problème qu'on cherche à résoudre. Nous l'utiliserons dans la suite de ce document pour définir nos chaînes de traitement.

$$\begin{cases} \mathcal{P}_i = \{C_i, \pi_i^C\} \\ a_i = C_i \times (\pi_i^C + \pi_i^-) \\ \pi_i^+ = \pi_i^- + \pi_i^C - \pi_i^S \\ r_i = \mathcal{F}(C_i, \pi_i^C) \\ \pi_{i-1}^+ = \pi_i^- \\ \pi_0^+ = \Pi^I \\ \pi_i^+ \geq \left\lceil \sum_{k=1}^{i} \frac{r_i}{6} + 1 \right\rceil \end{cases} \tag{3.7}$$

Nous avons défini les contraintes communes aux deux problèmes, par la suite, nous préciserons les contraintes spécifiques à chacun d'eux.

La première version du problème (la maximisation de la réjection avec des ressources restreintes) a pour principale contrainte que la taille de notre FPGA est limitée, on la notera $\mathcal{A}$. Cette taille maximum sera définie par l'utilisateur (en fonction de la plateforme de développement disponible). Plus elle sera grande, plus l'espace de recherche sera grand car on pourra mettre plus de filtres. Il est donc nécessaire de s'assurer que les n étages ne dépassent pas cette limite :

$$\sum_{i=1}^{n} a_i \leq \mathcal{A} \tag{3.8}$$

Ce problème vise à maximiser la réjection totale du signal. On va donc chercher à maximiser la somme totale de la réjection de chacun des étages de filtrage :

$$\text{Maximiser} \sum_{i=1}^{n} r_i \tag{3.9}$$

Enfin, le système 3.10 exprime sous forme plus formelle le problème à optimiser.

$$
\begin{aligned}
&\text{Maximiser } \sum_{i=1}^{n} r_i \\
&\sum_{i=1}^{n} a_i \leq \mathcal{A} \\
&\quad a_i = C_i \times (\pi_i^C + \pi_i^-), \quad \forall i \in [1,n] \\
&\quad \pi_i^+ = \pi_i^- + \pi_i^C - \pi_i^S, \quad \forall i \in [1,n] \\
&\quad r_i = F(C_i, \pi_i^C), \quad \forall i \in [1,n] \\
&\quad \pi_{i-1}^+ = \pi_i^-, \quad \forall i \in [1,n] \\
&\quad \pi_0^+ = \Pi^I \\
&\quad \pi_i^+ \geq \left\lceil \sum_{k=1}^{i} \frac{r_i}{6} + 1 \right\rceil, \quad \forall i \in [1,n]
\end{aligned}
\tag{3.10}
$$

Le problème complémentaire change peu. Nous devons juste remplacer la contrainte sur la taille occupée par une autre qui nous assure que l'objectif de réjection $\mathcal{R}$ est bien atteint :

$$\sum_{i=1}^{n} r_i \geq \mathcal{R} \tag{3.11}$$

On notera que cette limite est elle aussi définie par l'utilisateur selon les besoins de son expérimentation.

L'autre changement est bien évidemment la fonction objectif qui n'est plus la même. Ici nous allons chercher à minimiser la consommation de ressources. Nous obtenons ainsi le système 3.12.

$$
\begin{aligned}
&\text{Minimiser } \sum_{i=1}^{n} a_i \\
&\sum_{i=1}^{n} r_i \geq \mathcal{R} \\
&\quad a_i = C_i \times (\pi_i^C + \pi_i^-), \quad \forall i \in [1,n] \\
&\quad \pi_i^+ = \pi_i^- + \pi_i^C - \pi_i^S, \quad \forall i \in [1,n] \\
&\quad r_i = F(C_i, \pi_i^C), \quad \forall i \in [1,n] \\
&\quad \pi_{i-1}^+ = \pi_i^-, \quad \forall i \in [1,n] \\
&\quad \pi_0^+ = \Pi^I \\
&\quad \pi_i^+ \geq \left\lceil \sum_{k=1}^{i} \frac{r_i}{6} + 1 \right\rceil, \quad \forall i \in [1,n]
\end{aligned}
\tag{3.12}
$$

Dans les deux cas, les variables que nous utiliserons sont bornées. a_i sera comprise entre 0 et $+\infty$ puisque avoir une taille négative n'a pas de sens et que la taille peut être

arbitrairement grande. Dans le cas du premier problème, on peut également définir la borne supérieure comme étant $\mathcal{A}$. π_i^+, π_i^-, π_i^C et π_i^S représentent des nombres de bits, par définition, ils appartiennent donc à l'ensemble des entiers naturels. Il en va de même pour C_i qui représente le nombre de coefficients. Enfin, r_i n'est pas contrainte ($-\infty < r_i < +\infty$) car la réjection peut être positive dans la plupart des cas, notons pourtant qu'elle peut être négative dans le cas où les performances dans la bande passante sont plus mauvaises que dans la bande de coupure.

Nous venons de définir l'ensemble des nouvelles contraintes qu'a induit l'introduction du décalage logique. Dès à présent, il n'est plus possible de faire l'analogie avec le problème du sac à dos (et donc d'utiliser les outils de résolution développés pour celui-ci) car l'une des hypothèses fortes, dans ce cas précis, est que l'ordre des objets n'a pas d'importance. Or avec le nouveau modèle, puisqu'on introduit la variable π_i^S et la contrainte 3.4, l'ordre des FIR devient important. Prenons comme exemple un filtre a qui rejette beaucoup et un autre filtre b qui rejette peu. Si on place a avant b, à la sortie de a on ne pourra pas beaucoup décaler (décalage logique) car l'ENOB sera grand et le filtre b occupera plus de place car la taille des données en entrée sera grande. Au contraire, si on place b avant a, comme b rejette peu, les données en entrée de a seront petites et on consommera moins de ressources.

Nous allons donc devoir utiliser de nouvelles méthodes pour trouver une solution optimale. Ne parvenant pas à établir un lien avec un autre problème connu de la littérature scientifique, nous allons utiliser des méthodes plus génériques pour résoudre notre problème.

3.3 Transformation des contraintes en problème quadratique

Nous venons d'étoffer le modèle et nous avons défini tout un ensemble de contraintes qui régissent le système. Nous allons dans cette section adapter le modèle à un problème quadratique afin de pouvoir trouver une solution optimale.

Nous avons choisi cet outil issu de la recherche opérationnelle car transformer la modélisation en un programme quadratique n'est pas très compliqué et nous avons peu de cas qui ne sont pas linéaires ou quadratiques.

Dans cette section nous détaillerons les différentes étapes pour rendre quadratique le modèle.

Reprenons équation par équation la première version du modèle :

$$\text{Maximiser } \sum_{i=1}^{n} r_i$$

$$\sum_{i=1}^{n} a_i \leq \mathcal{A} \tag{3.13}$$

$$a_i = C_i \times (\pi_i^C + \pi_i^-), \qquad \forall i \in [1,n] \tag{3.14}$$

$$\pi_i^+ = \pi_i^- + \pi_i^C - \pi_i^S, \qquad \forall i \in [1,n] \tag{3.15}$$

$$r_i = F(C_i, \pi_i^C), \qquad \forall i \in [1,n] \tag{3.16}$$

$$\pi_{i-1}^+ = \pi_i^-, \qquad \forall i \in [1,n] \tag{3.17}$$

$$\pi_0^+ = \Pi^I \tag{3.18}$$

$$\pi_i^+ \geq \left\lceil \sum_{k=1}^{i} \frac{r_i}{6} + 1 \right\rceil, \qquad \forall i \in [1,n] \tag{3.19}$$

Les équations 3.13, 3.15, 3.17 et 3.18 sont toutes linéaires, on peut donc les intégrer directement. En revanche les autres ne sont pas linéaires, on doit les linéariser pour pouvoir les utiliser.

Commençons par l'équation 3.19. Dans ce cas, seul l'arrondi au supérieur nous pose problème. Pour remédier à cela, on ajoute 1 à la somme totale pour nous garantir d'être au-dessus de la valeur arrondie. Dans le pire des cas (si on était très proche de l'entier supérieur), on rajouterait presque un bit de plus mais ce qui ne serait pas grave puisque le but de cette contrainte est de s'assurer qu'on n'est pas en dessous du ENOB. Dans le meilleur des cas (si on était très proche de l'entier inférieur), l'ajout du bit en plus est presque identique à l'arrondi en lui même. Voici la nouvelle équation :

$$\pi_i^+ \geq 1 + \sum_{k=1}^{i} \left(\frac{r_i}{6} + 1 \right), \qquad \forall i \in [1,n] \tag{3.20}$$

Il reste encore les équations 3.14 et 3.16. Dans la première, on multiplie des variables entre elles et dans la seconde on ne sait pas comment trouver la fonction de réjection généralisée. Dans un cas comme dans l'autre, il n'est pas possible de linéariser simplement ces deux équations. Toutefois, dans la section 3.1 de ce chapitre, nous avons expliqué comment on pouvait calculer la réjection pour un couple $\{C_i, \pi_i^C\}$ donné. Nous créons ainsi des configurations de filtres allant de 1 à p qui seront les valeurs de chacun des points des pyramides présentées dans les figures 3.8a et 3.8b. Pour chaque configuration j, on associe les constantes C_{ij} et π_{ij}^C qui attribuent à l'étage i respectivement le nombre de coefficients ainsi que le nombre de bits pour la configuration j. On définit également la variable binaire δ_{ij} telle que si δ_{ij} vaut 1 alors à l'étage i ($1 \leq i \leq n$) la configuration j ($1 \leq j \leq p$) est choisie sinon δ_{ij} vaut 0. Par conséquent les anciennes variables C_i et π_i^C sont supprimées au profit de la nouvelle notation.

À cela on rajoute la contrainte suivante :

$$\sum_{j=1}^{p} \delta_{ij} \leq 1, \qquad \forall i \in [1, n] \tag{3.21}$$

Cette contrainte signifie qu'on a, au plus, un filtre par étage. On ne peut pas choisir deux configurations de filtre différentes pour un même étage i mais on peut éventuellement n'avoir aucun filtre pour un étage. Ce cas de figure peut se produire par exemple si la solution optimale ne comporte que 2 filtres et qu'on laisse la possibilité d'avoir jusqu'à 5 filtres. Le modèle permettra dans ce cas de laisser 3 étages vides.

L'introduction de la configurations nous oblige à réécrire la contrainte 3.15 puisque C_i et π_i^C n'existent plus. Elle devient ainsi :

$$\pi_i^+ = \pi_i^- + \left(\sum_{j=1}^{p} \delta_{ij} \pi_{ij}^C \right) - \pi_i^S, \qquad \forall i \in [1, n] \tag{3.22}$$

La différence majeure est la somme de tous les $\delta_{ij}\pi_{ij}^C$ qui revient à dire qu'on ne considère que le filtre choisi par étage. En effet comme pour tout j d'un étage i on aura au plus un seul $\delta_{ij} = 1$ cette somme sera seulement égale à ce π_{ij}^C choisi et dans le cas où aucun filtre n'est choisi, la somme vaudra 0 et la taille en sortie sera donc égale à l'entrée moins l'éventuel décalage. Notons que π_i^S n'est pas lié à δ_{ij} : il est donc possible d'avoir un décalage logique alors qu'aucun filtre n'a été sélectionné. Il restera néanmoins contraint par l'équation 3.20.

De même, les équations 3.13 et 3.16 doivent être ré-écrites comme ceci :

$$a_i = \sum_{j=1}^{p} \delta_{ij} \times C_{ij} \times (\pi_{ij}^C + \pi_i^-), \qquad \forall i \in [1, n] \tag{3.23}$$

$$r_i = \sum_{j=1}^{p} \delta_{ij} \times \mathcal{F}(C_{ij}, \pi_{ij}^C), \qquad \forall i \in [1, n] \tag{3.24}$$

Avec la même logique que précédemment, l'équation 3.23 reprend l'idée qu'au plus un filtre est choisi par étage, donc pour un étage i, soit la taille est égale à 0 (aucun filtre n'a été choisi), soit elle est égale au produit de $C_{ij} \times (\pi_{ij}^C + \pi_i^-)$ du filtre j choisi. Il est de même pour l'équation 3.24 qui indique que pour un étage i, on ne considère que la réjection du filtre choisi j ou 0 si aucun filtre n'est choisi.

Concernant la valeur de la fonction $\mathcal{F}$, on peut résoudre le problème grâce à l'utilisation de configurations. En effet, dans la partie 3.1, nous avons vu qu'il n'est pas simple de déterminer la réjection d'un filtre *a priori*. Cependant, nous pouvons calculer pour chacune des configurations de filtre ce critère de performance. Ainsi, en faisant un peu évoluer le modèle, on peut ajouter à l'ensemble de paramètres $\mathcal{P}_i$, la valeur F_{ij} de la réjection pour le filtre j à l'étage i. Nous pouvons donc réécrire la formule 3.24 :

$$r_i = \sum_{j=1}^{p} \delta_{ij} \times F_{ij}, \qquad \forall i \in [1, n] \tag{3.25}$$

Malgré ces changements, la formule 3.23 n'est toujours pas linéaire. En effet si on la décompose on obtient :

$$a_i = \sum_{j=1}^{p} \delta_{ij} \times C_{ij} \times \pi_{ij}^{C} + \delta_{ij} \times C_{ij} \times \pi_{i}^{-}, \qquad \forall i \in [1, n] \tag{3.26}$$

Le membre gauche de l'addition est bien linéaire car C_{ij} et π_{ij} sont devenues des constantes. En revanche le membre de droite n'est toujours linéaire car on multiplie deux variables du système. Ainsi, cette équation est pour l'instant quadratique, mais comme δ_{ij} est une variable binaire, nous serons en mesure de linéariser ce terme. En effet, la formule générique pour linéariser une relation du type $m = x \times y$ où x (bornée entre $0 \leq x \leq X^{max}$) est une variable réelle et où y est une variable binaire est :

$$m = x \times y \implies \begin{cases} m \geq 0 \\ m \leq y \times X^{max} \\ m \leq x \\ m \geq x - (1 - y) \times X^{max} \end{cases} \tag{3.27}$$

Cependant cette linéarisation complexifie la formulation du problème. C'est pourquoi nous nous contenterons d'exposer la version quadratique pour une meilleure lisibilité, tout en sachant qu'on peut passer à une version pleinement linéaire au besoin. De plus, nous allons utiliser Gurobi, un logiciel pour résoudre les programmes linéaires et quadratiques, capable de faire cette linéarisation implicitement. Nous reviendrons sur le sujet dans la section 4.1 du prochain chapitre.

En conclusion, voici les problèmes quadratiques que nous allons chercher à résoudre. Pour rappel, le problème 3.28 va chercher à maximiser la réjection du bruit tout en ne dépassant pas une taille donnée représentative des ressources de la plateforme, tandis que le problème 3.29 va minimiser l'espace occupé tout en visant à minimiser un palier de bruit.

Notre méthodologie se compose de quatre étapes. Dans le chapitre précédent, nous avons présenté la première étape qui était la définition d'un modèle abstrait de notre problème. Dans ce chapitre, nous détaillons la deuxième étape qui consiste en la recherche d'une solution optimale. Nous avons donc transformé le modèle en programme quadratique pour pouvoir trouver une solution optimale.

Dans le chapitre suivant, nous allons nous intéresser aux deux dernière étapes :

3. transformer et appliquer la solution abstraite dans un cas concret
4. comparer les résultats concrets avec les résultats abstraits pour s'assurer que le modèle est cohérent

$$
\begin{aligned}
&\text{Maximiser } \sum_{i=1}^{n} r_i \\
&\text{Constantes :} \\
&\qquad C_{ij}, && \forall i \in [1,n], \forall j \in [1,p] \\
&\qquad \pi_{ij}^{C}, && \forall i \in [1,n], \forall j \in [1,p] \\
&\qquad F_{ij}, && \forall i \in [1,n], \forall j \in [1,p] \\
&\qquad \mathcal{A} \\
&\qquad \Pi^{I} \\
&\text{Variables :} \\
&\qquad \delta_{ij} \in \{0,1\} && \forall i \in [1,n], \forall j \in [1,p] \\
&\qquad a_i \in [0,+\infty] && \forall i \in [1,n] \\
&\qquad \pi_i^{-} \in \mathbb{N}^* && \forall i \in [1,n] \\
&\qquad \pi_i^{+} \in \mathbb{N}^* && \forall i \in [1,n] \\
&\qquad \pi_i^{+} \in \mathbb{N}^* && \forall i \in [1,n] \\
&\qquad \pi_i^{S} \in \mathbb{N}^* && \forall i \in [1,n] \\
&\qquad r_i \in \mathbb{R} && \forall i \in [1,n] \qquad (3.28) \\
&\text{Contraintes :} \\
&\qquad \sum_{i=1}^{n} a_i \leq \mathcal{A} \\
&\qquad \sum_{j=1}^{p} \delta_{ij} \leq 1, && \forall i \in [1,n] \\
&\qquad a_i = \sum_{j=1}^{p} \delta_{ij} \times C_{ij} \times (\pi_{ij}^{C} + \pi_i^{-}), && \forall i \in [1,n] \\
&\qquad \pi_i^{+} = \pi_i^{-} + \pi_i^{C} - \pi_i^{S}, && \forall i \in [1,n] \\
&\qquad r_i = \sum_{j=1}^{p} \delta_{ij} \times F_{ij} && \forall i \in [1,n] \\
&\qquad \pi_{i-1}^{+} = \pi_i^{-}, && \forall i \in [1,n] \\
&\qquad \pi_0^{+} = \Pi^{I} \\
&\qquad \pi_i^{+} \geq 1 + \sum_{k=1}^{i} \left(\frac{r_i}{6} + 1 \right) && \forall i \in [1,n]
\end{aligned}
$$

FIGURE 3.11 – Programme quadratique pour maximiser la réjection du bruit tout en ne dépassant pas une taille donnée

$$
\begin{aligned}
&\text{Minimiser } \sum_{i=1}^{n} a_i \\
&\text{Constantes :} \\
&\quad C_{ij}, && \forall i \in [1,n], \forall j \in [1,p] \\
&\quad \pi_{ij}^{C}, && \forall i \in [1,n], \forall j \in [1,p] \\
&\quad F_{ij}, && \forall i \in [1,n], \forall j \in [1,p] \\
&\quad \mathcal{R} \\
&\quad \Pi^{I} \\
&\text{Variables :} \\
&\quad \delta_{ij} \in \{0,1\} && \forall i \in [1,n], \forall j \in [1,p] \\
&\quad a_i \in [0,+\infty] && \forall i \in [1,n] \\
&\quad \pi_i^{-} \in \mathbb{N}^* && \forall i \in [1,n] \\
&\quad \pi_i^{+} \in \mathbb{N}^* && \forall i \in [1,n] \\
&\quad \pi_i^{+} \in \mathbb{N}^* && \forall i \in [1,n] \\
&\quad \pi_i^{S} \in \mathbb{N}^* && \forall i \in [1,n] \\
&\quad r_i \in \mathbb{R} && \forall i \in [1,n] \qquad (3.29) \\
&\text{Contraintes :} \\
&\quad \sum_{i=1}^{n} r_i \geq \mathcal{R} \\
&\quad \sum_{j=1}^{p} \delta_{ij} \leq 1, && \forall i \in [1,n] \\
&\quad a_i = \sum_{j=1}^{p} \delta_{ij} \times C_{ij} \times (\pi_{ij}^{C} + \pi_i^{-}), && \forall i \in [1,n] \\
&\quad \pi_i^{+} = \pi_i^{-} + \pi_i^{C} - \pi_i^{S}, && \forall i \in [1,n] \\
&\quad r_i = \sum_{j=1}^{p} \delta_{ij} \times F_{ij} && \forall i \in [1,n] \\
&\quad \pi_{i-1}^{+} = \pi_i^{-}, && \forall i \in [1,n] \\
&\quad \pi_0^{+} = \Pi^{I} \\
&\quad \pi_i^{+} \geq 1 + \sum_{k=1}^{i} \left(\frac{r_i}{6} + 1 \right) && \forall i \in [1,n]
\end{aligned}
$$

FIGURE 3.12 – Programme quadratique pour minimiser l'espace occupé tout en visant à atteindre un palier de bruit

Troisième partie

Expérimentations et analyse des résultats

Chapitre 4

Application expérimentale : du solveur au FPGA

Nous avons détaillé les deux premières étapes de notre méthodologie dans les chapitres 2 et 3. Dans le chapitre 2, nous avons défini une abstraction de la chaîne de traitement à optimiser. Dans le chapitre 3, nous avons décliné le problème en deux variantes :

1. Comment maximiser la réjection d'une cascade de filtre avec une place limitée ?
2. Comment minimiser la place occupée tout en atteignant un niveau de réjection donné ?

Nous avons également donné un programme quadratique répondant à chacun de ces deux problèmes afin de trouver une solution optimale.

Nous allons détailler les deux dernières étapes dans ce chapitre. La troisième étape, consiste à traduire une solution optimale abstraite en un design FPGA. Dans la section 4.1 nous expliquerons comment nous automatisons cette étape, nous permettant ainsi de générer simplement de nombreuses expérimentations.

La quatrième étape, quant à elle, consiste à produire des résultats issus d'une expérimentation matérielle et à vérifier que ces résultats soient conformes à nos attentes (aux résultats abstraits). Dans la section 4.2, nous présenterons des résultats expérimentaux répondants aux deux problèmes adressés par les programmes quadratiques. Nous verrons également comment il est possible d'utiliser ces résultats pour faire de l'optimisation multi-objectifs.

Enfin dans la section 4.3, nous verrons comment il est possible d'améliorer notre modèle et ce que cela impliquerait.

4.1 Flux de travail

Afin de pouvoir tester facilement plusieurs instances des problèmes d'optimisation, nous avons automatisé la génération et l'exécution des expérimentations. Avant de présen-

ter le fonctionnement de notre méthode de travail, nous allons tout d'abord voir comment on génère des résultats manuellement.

Pour obtenir des résultats expérimentaux, il y a plusieurs étapes :

1. Choisir une instance d'un des deux problèmes et trouver une solution optimale pour celle-ci.
2. Créer le design FPGA correspondant à la solution optimale précédemment trouvée.
3. Faire l'expérimentation : configurer le FPGA, acquérir les données, les post-traiter.
4. Confronter les résultats expérimentaux aux résultats attendus.

La première étape consiste donc à résoudre une instance de du problème. On appelle instance, une version d'un des deux programmes quadratiques où toutes les constantes ont des valeurs définies : $\mathcal{A}$ ou $\mathcal{R}, \Pi^I, n, p....$ Autrement dit, une instance est juste un cas particulier du modèle.

Pour une instance donnée, nous devons chercher une solution. Cette solution nous indique quels doivent être les filtres sélectionnés pour obtenir une solution optimale au problème posé : quels sont les δ_{ij} qui valent 1 ? Pour trouver une solution nous faisons appel à un logiciel pour résoudre des programmes linéaires (solveur). Notre premier choix s'est porté sur GNU Linear Programming Kit (GLPK), un solveur open-source capable de traiter aussi bien les programmes linéaires en nombre réel (*Linear Programming*, LP) que ceux en nombre entier (*Mixed Integer Programming*, MIP).

L'inconvénient d'avoir des variables codées en nombre entier est que le programme linéaire devient bien plus complexe à résoudre. Quand on travaille en nombre réel, il existe l'algorithme du simplexe qui permet de trouver une solution optimale avec des coefficients réels. Malheureusement, cet algorithme n'est pas applicable avec des nombres entiers. Pour y remédier, on peut trouver une solution en nombre réel puis d'explorer tout l'espace de recherche proche afin de trouver la solution en nombre entier optimale. Cette approche est rapide dans le cas où il n'y a que deux variables entières. En revanche quand, comme dans notre cas, nous pouvons avoir plusieurs dizaines de variables, l'espace de recherche est particulièrement grand. C'est pourquoi nous avons recours à d'autres outils de recherche opérationnelle tel que le retour sur trace *(backtracking)* pour trouver la solution plus rapidement. C'est le solveur qui se chargera de faire cette recherche pour nous.

La principale contrainte de GLPK est qu'il n'est pas capable de traiter des problèmes quadratiques : nous avons donc utilisé la version linéarisée du modèle. Comme nous l'avions dit dans la section 3.1 du chapitre précédent, pour un même couple de $\{C_{ij}, \pi_{ij}^C\}$ nous avons deux valeurs de réjection car les coefficients sont générés par deux fonctions différentes (`fir1` et `firls`) de GNU Octave. Cela veut dire que pour un même nombre de coefficients et de bits nous avons deux jeux de coefficients différents. Afin de faciliter l'interprétation des résultats, nous avons rajouté la méthode de génération du filtre comme meta-donnée. Cette donnée n'entre pas directement en jeu dans le choix du filtre. En effet, du point de vue du solveur, seule la valeur de réjection lui importe, donc que ça soit l'une ou l'autre des fonctions il ne s'intéressera qu'à la quantité de bruit rejeté, nous gardons cette information uniquement pour retrouver plus facilement le filtre sélectionné afin de

pouvoir l'ajouter dans la chaîne de traitement finale.

Malheureusement, les performances de GLPK ne furent pas suffisantes. En effet, même pour des instances restreintes de notre programme (quand $n = 1$), les temps de calcul étaient beaucoup trop longs (de l'ordre de la semaine). Nous avons donc opté pour un autre solveur : Gurobi. Ce nouveau solveur gère également les LP et les MIP mais il nous permet également de travailler avec des modèles quadratiques. Nous ne sommes plus obligés d'utiliser la version linéaire du programme, nous pouvons garder la forme quadratique, plus facile à manipuler. De plus les performances sont suffisantes pour nous permettre de tester de nombreuses instances du modèle : nous détaillerons les performances dans la section 4.2 de ce chapitre.

Le choix du solveur étant fait, nous sommes en mesure d'avoir une solution optimale pour une instance spécifique du modèle. La deuxième étape est donc de traduire la solution donnée par Gurobi en un design sur FPGA. Comme nous l'avons dit, le solveur va donner une liste de filtres, il faut donc qu'il crée toute une chaîne de traitement avec les mêmes paramètres. Dans le cadre de nos recherches, nous travaillons avec des Zynq 7010 SoC et plus précisément avec des cartes Redpitaya. Ce type de carte nous offre la possibilité d'avoir un système d'exploitation Linux embarqué à coté d'un FPGA avec un bus de communication rapide entre les deux. De plus, la Redpitaya est une carte relativement petite en terme de ressources, elle dispose d'une puce FPGA aux performances restreintes si on la compare avec d'autres cartes telles que la ZedBoard ou la ZC706 de chez Xilinx. Cela nous permettra de prouver que l'approche que nous proposons est efficace quelle que soit la puissance de la carte : si nous sommes capables de tirer le plus possible d'une petite carte, il sera possible de faire de mieux avec une carte plus puissante. De plus la Redpitaya est une carte disposant de ses propres ADC et DAC : elle est donc souvent utilisée pour faire du traitement numérique du signal.

Le traitement en lui même sera réduit au minimum : on prend un bruit (généré par un générateur pseudo-aléatoire sur 20 bits dans le FPGA), on le filtre, puis on récupère une portion du signal traité. Le bruit généré est un bruit blanc, nous n'avons donc pas à tenir compte de bruits long terme qui pourraient complexifier notre analyse. Nous sommes tout à fait conscients de nous placer dans une situation très confortable mais le but est de nous assurer que notre démarche est viable, nous voulons donc limiter au maximum les paramètres du système. En effet, si nous prenions comme entrée un signal numérisé par l'ADC de la carte Redpitaya, cela rajouterait des facteurs caractérisant le bruit (dérive du signal, parasitage de l'environnement...). C'est pourquoi, nous préférons la solution de la génération d'un bruit blanc pour tester notre approche car le comportement du bruit est maîtrisé. Il sera donc plus simple de vérifier les résultats finaux.

La solution trouvée par le solveur va donc influer sur l'étape de filtrage du signal. C'est elle qui va nous dire quels filtres mettre et dans quel ordre mais la partie acquisition du signal et récupération des données sera la même pour tous les traitements. Puisque nous travaillons avec une puce Zynq, nous utilisons le logiciel de Xilinx, Vivado 2018.2, pour générer le design du FPGA (ou le bitstream).

Dès que notre design est prêt, il nous reste à configurer la puce FPGA de la carte

Redpitaya (flasher le FPGA). À partir de ce moment, le FPGA travaille et peut nous fournir les données brutes (entrée) et filtrées (sortie). Nous allons récupérer plusieurs paquets de 4096 données non contigus afin d'avoir une quantité suffisante pour gommer l'aspect aléatoire de l'entrée et avoir des spectres suffisamment lissés pour être analysés.

La troisième étape consiste à récupérer ces données et à les analyser. Puisque nous avons un système d'exploitation à coté du FPGA, nous pouvons lui demander de s'occuper de récupérer les données dans la mémoire interne du FPGA (les blocs de BRAM). Dans les faits, c'est même ce système qui va initialiser les filtres en envoyant les coefficients au FPGA.

Les blocs de traitement et les pilotes Linux nécessaires à générer le design et l'application logicielle embarquée dans la carte nous sont fournis par l'écosystème OcsImpDigital [33]. Il s'agit d'un ensemble d'outils et de composants open-source spécialisés dans l'étude d'oscillateurs utra-stables.

Une fois que nous avons récupéré suffisamment de données, nous les transférons à la machine hôte pour les traiter. Cette étape consiste à calculer la réjection du signal. Comme nous travaillons avec un bruit blanc en entrée, nous savons que le bruit est uniformément réparti sur toutes les fréquences. Néanmoins, par respect de la définition de la fonction de transfert, les caractéristiques du filtre sont calculées comme le ratio du spectre en sortie par le spectre du signal d'entrée.

Avec les données post-traitées, on peut comparer la réjection effective (celle donnée par les expérimentations) avec la réjection abstraite (celle donnée par la résolution du programme quadratique). De même, nous pouvons vérifier que les unités abstraites, utilisées dans le modèle pour estimer la consommation de ressources, coïncident avec les composants réellement utilisés dans le FPGA (BRAM, LUT, DSP...). Nous reviendrons sur ce point dans la section 4.2.

Toutes ces étapes doivent être adaptées et répétées pour chaque instanciation du problème (changement du nombre de paramètres, changement des objectifs, changement de la place limitée ou de la réjection ciblée...). Cependant, les étapes sont longues et nous allons devoir tester de nombreuses instanciations. Dans un premier temps nous avons simulé les résultats grâce à une bibliothèque logicielle que nous avons créée pour nos besoins (nous la détaillons dans l'annexe A). Puis dans un second temps, quand les résultats simulés étaient satisfaisant, nous avons automatisé toutes ces étapes. La figure 4.1 montre la nouvelle façon de générer nos résultats.

Le déroulement reste sensiblement le même, nous avons adapté les différentes étapes pour les rendre automatisables. La première étape cherche non seulement une solution optimale mais elle va générer deux scripts différents. Le premier script (l'étape 1a du schéma) est un script en TCL *(Tool Command Language)* qui sera lu par Vivado à l'étape 2 pour générer le design du FGPA. Ces blocs sont issus du framework ocsimpDigital [33]. Ce framework nous fournit un ensemble de skeleton (génériques et paramétrables) qui nous permet de construire le design du FPGA sans qu'on doive tout réimplémenter. De plus ce framework gère la communication entre les blocs ce qui nous permet d'être indépendant vis à vis de la plateforme FPGA. Nous avons mis en ligne les sources du pro-

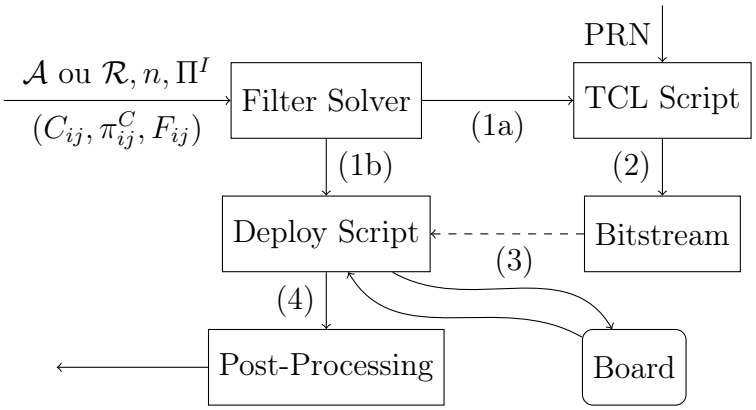


FIGURE 4.1 – Schéma de synthèse automatique des résultats expérimentaux

gramme linéaire : `https://github.com/oscimp/cascade_filters_solver`. On a généré un ensemble de filtres (`https://github.com/oscimp/cascade_filters_solver/tree/master/fir_data`) puis on calcul la solution optimale. À partir d'elle on génère les deux scripts.

Le second script quant à lui sera un script Shell qui sera exécuté lors de l'étape 3. Il sera chargé, une fois l'étape 2 terminée, de piloter toute l'expérimentation : configurer la carte Redpitaya (flasher le FPGA, configurer les filtres...), faire l'acquisition des données et les transférer pour faire le post-traitement. L'étape 4 reste la même qu'auparavant, il s'agit de calculer la réjection du filtre et de comparer les résultats avec la solution optimale de la première étape.

Avec cette nouvelle méthode, nous sommes capables d'automatiser du début à la fin la génération d'un de nos résultats. Il nous sera donc possible de générer facilement plusieurs résultats différents sans avoir à superviser toutes les étapes. Cela sera particulièrement utile pour établir le front de Pareto que nous détaillerons dans la section 4.2.3.

4.2 Expérimentations

Dans cette section nous allons présenter la dernière étape de notre méthodologie : l'interprétation des résultats. Nous avons mené deux expérimentations distinctes en utilisant les deux programmes quadratiques :

— Comment maximiser la réjection à surface donnée (section 4.2.1) ?

— Comment minimiser la surface occupée avec une réjection ciblée (section 4.2.2) ?

Pour répondre à ces deux questions, nous avons généré 1827 jeux de coefficients ($p = 1827$) variant de 2 à 22 bits et de 3 à 60 coefficients.

Pour chacune des ces deux expérimentations, nous utilisons la méthode de travail présentée dans la section précédente pour produire tous les résultats présentés. Nous verrons également dans la section 4.2.3 comment croiser ces résultats pour obtenir des critères multi-objectifs. Pour finir nous ferons la synthèse de ces résultats dans la section 4.2.4.

4.2.1 Maximisation de la réjection à surface donnée

Dans cette première expérimentation nous créons des designs FPGA en imposant une consommation de ressources limite tout en cherchant à maximiser la réjection totale.

Nous prenons trois instances du programme quadratique 3.28 que nous avons vu dans la section 3.3 du chapitre précédent.

1. MAX/500 : la surface limite sera fixée à 500 u.a.
2. MAX/1000 : la surface limite sera fixée à 1000 u.a.
3. MAX/1500 : la surface limite sera fixée à 1500 u.a.

Pour chacune des ces instances nous faisons varier n le nombre de filtres consécutifs de 1 à 5. Ainsi nous pouvons comparer les performances d'un filtre monolithique (cas où $n = 1$) avec des filtres en cascade ($n > 1$).

Pour finir, nous définissons $\Pi^I = 16$ bits. Rappelons qu'il s'agit de la taille des données en entrée du premier filtre. Dans le cas présent, il s'agira de la taille des nombres donnés par le générateur pseudo-aléatoire.

La section 4.2.1.1 montre les résultats théoriques issus du programme quadratique. Dans la section 4.2.1.2 nous verrons la consommation de ressources dans le FPGA. La section 4.2.1.3 nous exposera les résultats issus de la carte Redpitaya.

4.2.1.1 Résultats du solveur quadratique

Les tableaux 4.1, 4.2 et 4.3 représentent les solutions optimales données par Gurobi après la résolution du programme quadratique respectivement pour les instances MAX/500, MAX/1000 et MAX/1500.

Les tableaux se lisent tous de la même façon. Chaque ligne donne la réponse pour un n différent, la suite des filtres choisis se lit de gauche à droite avec, dans chaque parenthèse, la configuration choisie pour l'étage i à savoir dans l'ordre : le nombre de coefficients C_i, la taille des coefficients π_i^C et le nombre de bits décalés π_i^S. Les deux dernières colonnes donnent respectivement la réjection attendue et la surface estimée. Toutes les configurations de filtres choisies sont celles de `firls`.

Examinons, par exemple, la première ligne du tableau 4.1. Dans ce cas, $n = 1$ nous n'avons donc qu'un filtre (filtre monolithique). la première case nous donne la configuration de filtre choisi pour l'étage $i = 1$. Comme nous n'avons qu'un étage, les autres cases sont vides. Avec cette solution nous nous attendons à avoir 32 dB de réjection et à occuper 483 unités arbitraires.

Le premier constat à tirer du tableau 4.1 est que toutes les solutions respectent la contrainte de taille : aucune d'elles n'est supérieure à 500 u.a.

Nous voyons également que la meilleure réjection est obtenue lorsque $n = 5$. La réjection est croissante en fonction du nombre du filtres : plus on a de filtres, meilleure

TABLE 4.1 – Configurations (C_i, π_i^C, π_i^S), réjections et surface (en unités arbitraires) pour MAX/500

n	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	Réjection	Surface
1	(21, 7, 0)	-	-	-	-	32 dB	483
2	(3, 3, 15)	(31, 9, 0)	-	-	-	58 dB	460
3	(3, 3, 15)	(27, 9, 0)	(5, 3, 0)	-	-	66 dB	488
4	(3, 3, 15)	(19, 7, 0)	(11, 5, 0)	(3, 3, 0)	-	74 dB	499
5	(3, 3, 15)	(23, 8, 0)	(3, 3, 1)	(3, 3, 0)	(3, 3, 0)	78 dB	489

TABLE 4.2 – Configurations (C_i, π_i^C, π_i^S), réjections et surface (en unités arbitraires) pour MAX/1000

n	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	Réjection	Surface
1	(37, 11, 0)	-	-	-	-	56 dB	999
2	(3, 3, 15)	(51, 14, 0)	-	-	-	87 dB	975
3	(3, 3, 15)	(35, 11, 0)	(19, 7, 0)	-	-	99 dB	1000
4	(3, 4, 16)	(27, 8, 0)	(19, 7, 1)	(11, 5, 0)	-	103 dB	998
5	(3, 3, 15)	(31, 9, 0)	(19, 7, 0)	(3, 3, 1)	(3, 3, 0)	111 dB	984

est la réjection, même si on voit que la réjection semble tendre vers une limite quand $n > 5$.

Si on observe les filtres sélectionnés, on peut s'apercevoir qu'un schéma semble se répéter (excepté pour le cas $n = 1$) : le premier filtre est petit (nombre de coefficients et de bits très bas) et on décale le plus de bits possibles, puis le second est le plus gros filtre de la cascade sans décalage et la fin de la cascade sont des filtres de plus en plus petits avec très peu de décalage.

Le tableau 4.2 nous donne les mêmes conclusions que le tableau précédent :
— La contrainte de taille est respectée (moins de 1000 unités arbitraires).
— La réjection est croissante avec le meilleur résultat quand $n = 5$.
— On observe le même schéma sur le choix des filtres : un premier petit filtre avec un grand décalage, puis un gros filtre avec aucun décalage et pour finir plusieurs petits filtres avec peu de décalage.
Le tableau 4.3 donne en partie les mêmes conclusions :

TABLE 4.3 – Configurations (C_i, π_i^C, π_i^S), réjections et surface (en unités arbitraires) pour MAX/1500

n	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	Réjection	Surface
1	(47, 15, 0)	-	-	-	-	71 dB	1457
2	(19, 6, 15)	(51, 14, 0)	-	-	-	103 dB	1489
3	(3, 3, 15)	(35, 11, 0)	(35, 11, 0)	-	-	122 dB	1492
4	(3, 3, 15)	(27, 8, 0)	(19, 7, 0)	(27, 9, 0)	-	129 dB	1498
5	(3, 3, 15)	(23, 9, 2)	(27, 9, 0)	(19, 7, 0)	(3, 3, 0)	136 dB	1499

TABLE 4.4 – Temps de calcul pour résoudre le programme quadratique

n	Temps (MAX/500)	Temps (MAX/1000)	Temps (MAX/1500)
1	0,1 s	0,1 s	0,3 s
2	1,1 s	2,2 s	12 s
3	17 s	137 s ($\approx$ 2 min)	275 s ($\approx$ 4 min)
4	52 s	5448 s ($\approx$ 90 min)	5505 s ($\approx$ 17 h)
5	286 s ($\approx$ 4 min)	4119 s ($\approx$ 68 min)	235479 s ($\approx$ 3 jours)

— La contrainte de taille est toujours respectée (moins de 1500 unités arbitraires).
— La réjection est croissante avec le meilleur résultat quand $n = 5$.

Cependant, le schéma d'affectation des filtres n'est pas tout à fait le même. Le premier filtre reste encore un petit filtre avec le plus possible de décalage mais la suite est répartie plus équitablement. Dans le cas où $n = 3$, le filtre à l'étage 2 ($i = 2$) et l'étage 3 ($i = 3$) sont les mêmes. Pour le cas où $n = 4$, les filtres aux étages 2 et 4 sont très proches, seulement un bit de différence sur la taille des coefficients. Et concernant l'étage 3, le filtre est légèrement plus petit (moins de coefficients mais le nombre de bits est très proche). Pour le dernier cas où $n = 5$, le premier filtre est le même petit filtre que dans les autres cas avec toujours un grand décalage. Les filtres à l'étage 2, 3 et 4 sont proches tout comme ils l'étaient dans les deux cas précédents mais cette fois, le dernier filtre est de nouveau un petit filtre.

De ces tableaux ressortent trois constatations :
— Il est possible de trouver une solution qui respecte les contraintes de taille.
— Plus il y a de filtres, meilleure est la réjection totale. Le cas monolithique est toujours le moins bon en terme de réjection.
— Bien qu'on ne puisse pas trouver un schéma de choix des filtres, il y a un point commun entre tous les cas : ils commencent par un petit filtre avec un décalage important.

On peut expliquer le dernier point grâce à la contrainte 3.20 de la section 3.3. Pour rappel, cette contrainte imposait que la taille des données en sortie d'un filtre soit suffisamment grande pour avoir un nombre satisfaisant de bits utiles (ENOB). Or, à l'étage 1, le nombre de bits utiles ne dépend que de la réjection du premier filtre. En prenant un petit filtre, donc une réjection faible, le nombre de bits nécessaires en sortie du premier étage sera faible et il sera possible de décaler beaucoup de bits. Ce faisant, à l'étage 2 la taille des données sera moindre et la place occupée par cet étage de filtre sera donc moins grande.

Le tableau 4.4 quant à lui montre les temps de calcul nécessaires pour résoudre les différentes instances du programme linéaire. Comme on peut le voir, le temps de calcul dépend de deux paramètres : la limite de surface et le nombre d'étages. Le temps de calcul semble augmenter exponentiellement en fonction du nombre d'étages : dans le cas MAX/1500 pour $n = 3$, nous sommes à moins de 5 minutes de calcul, puis pour $n = 4$ nous sommes à presque 1 jour de calcul et pour $n = 5$ nous sommes à plusieurs jours de calcul.

TABLE 4.5 – Occupation des ressources avec en dernière colonne la quantité de ressources disponibles

n	Ressources	MAX/500	MAX/1000	MAX/1500	*Zynq 7010*
1	LUT	249	453	627	*17600*
	BRAM	1	1	1	*120*
	DSP	21	37	47	*80*
2	LUT	2374	5494	691	*17600*
	BRAM	2	2	2	*120*
	DSP	0	0	70	*80*
3	LUT	2443	3304	3521	*17600*
	BRAM	3	3	3	*120*
	DSP	0	19	35	*80*
4	LUT	2634	3753	2557	*17600*
	BRAM	4	4	4	*120*
	DSP	0	19	46	*80*
5	LUT	2423	3047	2847	*17600*
	BRAM	5	5	5	*120*
	DSP	0	22	46	*80*

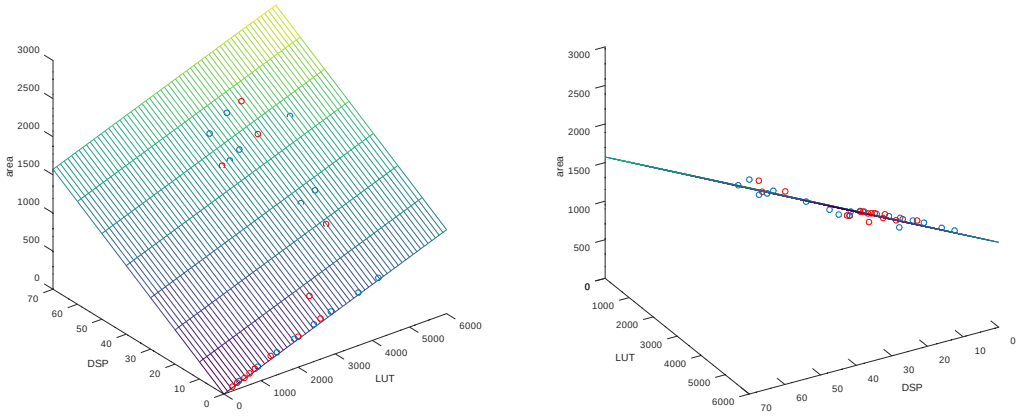
4.2.1.2 Consommation des ressources du FPGA

Dans cette section nous allons observer les rapports de consommation de ressources que Vivado dresse à chaque design, séparant chaque entité de traitement et permettant donc d'éliminer les contributions des blocs de communication PL-PS qui ne nous concernent pas dans cette analyse. Cela nous permettra donc de connaître la consommation réelle de ressources dans le FPGA.

Le tableau 4.5 nous donne donc pour chacune des expérimentations la quantité de ressources utilisées. Ces chiffres ne concernent que les différents filtres. Toutes les autres ressources utilisées ne sont pas comptabilisées ici.

Si on regarde la consommation des BRAM, on voit qu'elle correspond au nombre d'étages de filtres que nous avons. Cela reste logique puisque nous utilisons un bloc par FIR pour stocker les jeux de coefficients.

Afin de pouvoir comparer les unités abstraites et la consommation réelle de ressources, il faut connaître la relation entre la surface abstraite et la consommation réelle (LUT et DSP). La figure 4.2 donne la surface abstraite en fonction des LUT et DSP à partir des résultats issus des différentes expérimentations. En bleu, les points issus de la résolution du problème de maximisation de la réjection et en rouge les résultats issus de la minimisation de la consommation de ressources. Les expérimentations qui n'apparaissent pas de cette section seront présentés dans la section 4.2.2 ou 4.2.3 À partir de ce nuages de points, on calcule l'équation de plan par régression linéaire (le quadrillage) ce qui nous donne la relation entre la modélisation la consommation réelle. On obtient l'équation de



(a) Plan vue de face (b) Plan vue de coté

FIGURE 4.2 – Nuage de points représentant la surface abstraite occupée en fonction des LUT et DSP réellement consommés

plan suivante :

$$\text{Area} = 0,183 \times \text{LUT} + 22,469 \times \text{DSP} \tag{4.1}$$

À partir de cela, on peut déduire que :

1. 1 DSP = 122 LUT
2. 1 u.a. = 5,45 LUT

En ce qui concerne l'utilisation des LUT et DSP, cela reste cohérent avec notre modèle. En effet, d'après les solutions optimales présentées dans la section précédente, les designs ont tendance à utiliser toute la surface disponible (500, 1000 ou 1500 u.a. selon le cas). En partant de cette idée et en considérant que 1 $DSP = 122$ LUT on obtient le tableau 4.6

Hormis le cas MAX/1000 avec $i = 1$ et MAX/1500 $i = 1$, on voit que les tailles de LUT sont plus ou moins les mêmes. Ces deux exceptions peuvent s'expliquer parce qu'il s'agit de filtres monolithiques et que Vivado doit être capable de faire des optimisations de placement. Cependant, le fait que les tailles soient stables nous amène à conclure que notre abstraction de la surface arbitraire est cohérente avec la consommation de ressources réelles. On voit même qu'il y a une relation entre les unités arbitraires et les LUT : 500 u.a. correspondent à 2400 LUT, 1000 u.a. correspondent à 5750 LUT et 1500 u.a correspondent à 8000 LUT.

4.2.1.3 Données issues du traitement sur la carte Redpitaya

Dans cette dernière section nous allons nous intéresser aux résultats issus de la carte Redpitaya. Il s'agit de récupérer les données brutes issues du FPGA, puis de calculer le

TABLE 4.6 – Occupation des ressources en équivalent LUT avec l'hypothèse que 1 DSP = 122 LUT

n	Ressources	MAX/500	MAX/1000	MAX/1500	*Zynq 7010*
1	éq. LUT	2811	4967	6361	*17600*
	BRAM	1	1	1	*120*
2	éq. LUT	2374	5494	9231	*17600*
	BRAM	2	2	2	*120*
3	éq. LUT	2443	5622	7791	*17600*
	BRAM	3	3	3	*120*
4	éq. LUT	2634	6071	8169	*17600*
	BRAM	4	4	4	*120*
5	éq. LUT	2423	5731	8459	*17600*
	BRAM	5	5	5	*120*

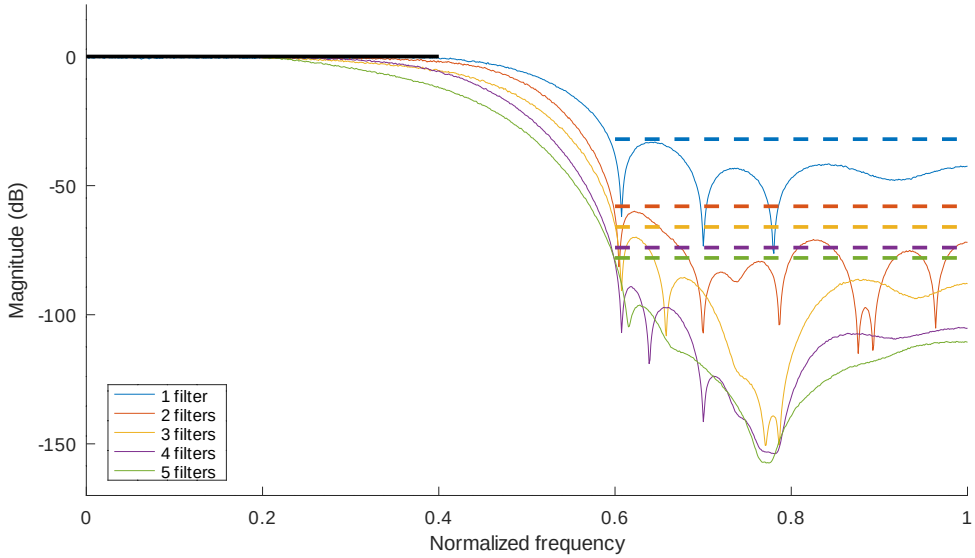


FIGURE 4.3 – Spectre du signal pour MAX/500

spectre du signal obtenu pour vérifier la réjection de notre étage de filtres par calcul de la fonction de transfert.

Les figures 4.3, 4.4 et 4.5 présentent respectivement les résultats pour MAX/500, MAX/1000 et MAX/1500. Dans chacune de ces figures, nous avons tracé le gabarit des filtres avec une barre noire pour le critère en passe-bande et des barres en pointillés de couleur pour le critère en stop-bande (donné par la réjection annoncée par la solution optimale), tandis que les courbes en couleur sont les fonctions de transfert moyennées à partir des données expérimentales. Les couleurs, quant à elles, correspondent aux différentes valeurs de i ($1 \leq i \leq 5$).

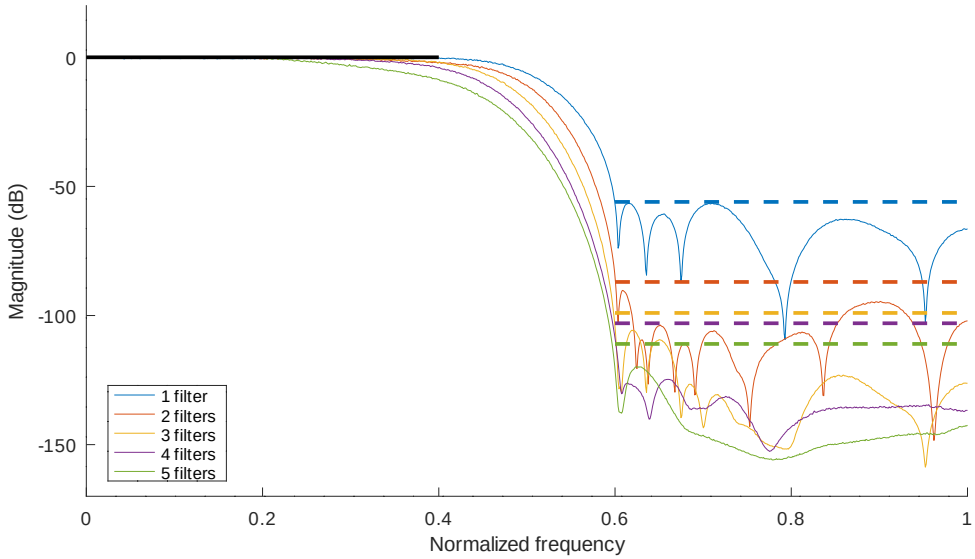


FIGURE 4.4 – Spectre du signal pour MAX/1000

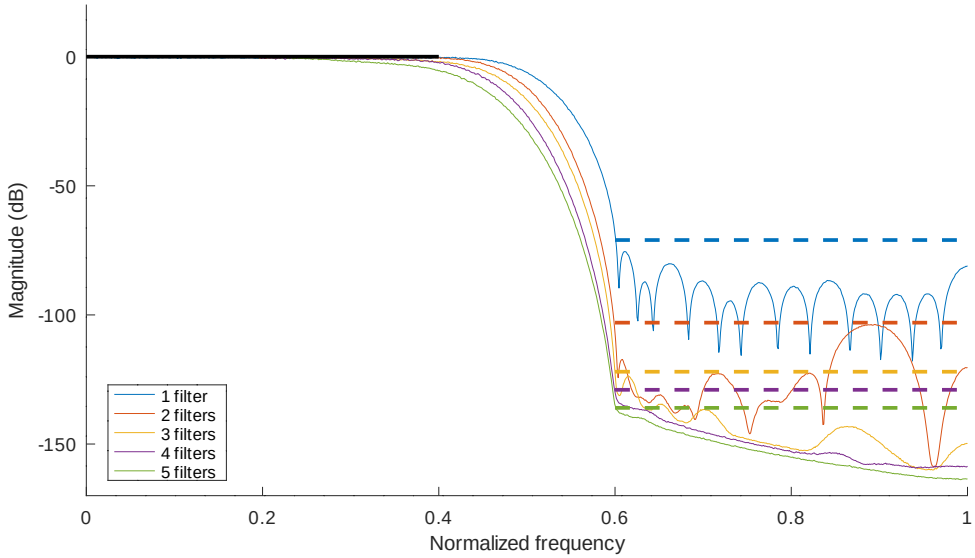


FIGURE 4.5 – Spectre du signal pour MAX/1500

Le premier constat est que les résultats issus du FPGA coïncident avec ceux issus du solveur. En effet, on voit que la réjection est de plus en plus grande quand le nombre d'étages de filtres augmente, avec la même limite quand n devient grand. On voit également que la réjection est au moins aussi bonne que celle attendue (les barres de couleur) et dans les faits, elle est même généralement meilleure que celle annoncée par le solveur.

Dans tous les cas, on voit qu'il est préférable d'avoir une cascade de filtres plutôt que d'avoir un filtre monolithique. En effet, dans les trois expérimentations, le cas du filtre monolithique est toujours le moins bon comparé ne serait-ce qu'au cas où $n = 2$ avec environ 30 dB d'écart entre les deux.

Néanmoins, on s'aperçoit que dans la bande passante vers la bande de transition, on a une forte déviation (> 10 dB) quand on cascade plus de deux filtres. Dans la section 4.3, nous expliquerons qu'il s'agit d'un problème avec le critère de réjection utilisé pour évaluer les filtres. On verra également comment corriger ce problème.

4.2.2 Minimisation de la surface à réjection donnée

Dans cette section, nous allons nous intéresser au problème complémentaire de la section précédente. En effet, nous allons chercher à optimiser la place occupée tout en atteignant une réjection spécifique.

Nous allons considérer quatre expérimentations différentes du programme quadratique 3.29 vu dans la section 3.3 :

1. MIN/40 : la réjection ciblée est fixée à 40 dB
2. MIN/60 : la réjection ciblée est fixée à 60 dB
3. MIN/80 : la réjection ciblée est fixée à 80 dB
4. MIN/100 : la réjection ciblée est fixée à 100 dB

Comme précédemment, nous ferons varier i entre 1 et 5 pour chacune des expérimentations.

Dans la section 4.2.2.1 nous verrons les résultats obtenus lors de la résolution du programme quadratique. La section 4.2.2.2 nous donnera les consommations de ressources réelles dans le FPGA. Enfin, la section 4.2.2.3 nous présentera les résultats obtenus grâce à la carte Redpitaya.

4.2.2.1 Résultats du solveur quadratique

Les tableaux 4.7, 4.8, 4.9 et 4.10 présentent les résultats pour chacune des expérimentations MIN/40, MIN/60, MIN/80 et MIN/100 et cette fois encore toutes les configurations de filtres proviennent de `firls`.

Comme nous le voyons sur le tableau 4.7 il manque la ligne pour $i = 5$. En effet, le solveur n'a trouvé de meilleure solution qu'avec seulement 4 filtres. Nous n'avons donc pas dupliqué cette ligne.

Dans le tableau 4.10, la ligne pour $i = 1$ est vide car il n'existe pas de solution possible pour atteindre 100 dB de réjection avec seulement un étage de filtrage.

Pour toutes les autres expérimentations, nous constatons qu'il existe une solution optimale qui arrive à atteindre les objectifs fixés.

TABLE 4.7 – Configurations (C_i, π_i^C, π_i^S), réjections et surface (en unités arbitraires) pour MIN/40

n	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	Réjection	Surface
1	(27, 8, 0)	-	-	-	-	41 dB	648
2	(3, 2, 14)	(19, 7, 0)	-	-	-	40 dB	263
3	(3, 3, 15)	(11, 5, 0)	(3, 3, 0)	-	-	41 dB	192
4	(3, 3, 15)	(3, 3, 0)	(3, 3, 0)	(3, 3, 0)	-	42 dB	147

TABLE 4.8 – Configurations (C_i, π_i^C, π_i^S), réjections et surface (en unités arbitraires) pour MIN/60

n	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	Réjection	Surface
1	(39, 13, 0)	-	-	-	-	60 dB	1131
2	(3, 3, 15)	(35, 10, 0)	-	-	-	60 dB	547
3	(3, 3, 15)	(27, 8, 0)	(3, 3, 0)	-	-	62 dB	426
4	(3, 2, 14)	(11, 5, 1)	(11, 5, 0)	(3, 3, 0)	-	60 dB	344
5	(3, 2, 14)	(3, 3, 1)	(3, 3, 0)	(3, 3, 0)	(3, 3, 0)	60 dB	279

TABLE 4.9 – Configurations (C_i, π_i^C, π_i^S), réjections et surface (en unités arbitraires) pour MIN/80

n	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	Réjection	Surface
1	(55, 16, 0)	-	-	-	-	81 dB	1760
2	(3, 3, 15)	(47, 14, 0)	-	-	-	80 dB	903
3	(3, 3, 15)	(23, 9, 0)	(19, 7, 0)	-	-	80 dB	698
4	(3, 3, 15)	(27, 9, 0)	(7, 7, 4)	(3, 3, 0)	-	80 dB	605
5	(3, 2, 14)	(27, 8, 0)	(3, 3, 1)	(3, 3, 0)	(3, 3, 0)	81 dB	534

TABLE 4.10 – Configurations (C_i, π_i^C, π_i^S), réjections et surface (en unités arbitraires) pour MIN/100

n	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	Réjection	Surface
1	-	-	-	-	-	-	-
2	(15, 7, 17)	(51, 14, 0)	-	-	-	100 dB	1365
3	(3, 3, 15)	(27, 9, 0)	(27, 9, 0)	-	-	100 dB	1002
4	(3, 3, 15)	(31, 9, 0)	(19, 7, 0)	(3, 3, 0)	-	101 dB	909
5	(3, 3, 15)	(23, 8, 1)	(19, 7, 0)	(3, 3, 0)	(3, 3, 0)	101 dB	810

TABLE 4.11 – Temps nécessaire à la résolution du problème quadratique avec Gurobi

n	Temps (MIN/40)	Temps (MIN/60)	Temps (MIN/80)	Temps (MIN/100)
1	0,07 s	0,02 s	0,01 s	-
2	7,8 s	16 s	14 s	1,8 s
3	4,7 s	14 s	28 s	39 s
4	39 s	20 s	193 s	522 s ($\approx$ 9 min)
5	-	12 s	170 s	1048 s ($\approx$ 17 min)

L'écart entre l'espace occupé par les filtres monolithiques et la cascade de cinq filtres est important : pour MIN/40, MIN/60 et MIN/80, le filtre monolithique occupe plus de trois fois la surface de la cascade de 5 filtres.

En revanche l'écart de surface entre les différentes cascades est réduit (entre 100 et 300 unités arbitraires) bien que la surface décroisse en fonction du nombre d'étages de filtre (n).

En ce qui concerne les choix des filtres, on faire plusieurs remarques :

1. Les cascades de filtres commencent souvent par un petit filtre suivi d'un grand décalage de bits.
2. Le second filtre est souvent un gros filtre avec peu de décalage.
3. La fin de la cascade ($i \geq 3$) est le plus souvent composée de petits filtres sans décalage.

Le choix de prendre un petit filtre suivi d'un grand décalage au premier étage s'explique de la même façon que précédemment. Le but est de pouvoir réduire le taille des données en entrée du second étage afin de minimiser la taille que va occuper le deuxième filtre. Ce schéma rappelle celui que nous avons vu dans la section 4.2.1.1 lors du choix du filtre pour l'autre programme quadratique. Ceci nous conforte dans l'idée que ces deux programmes sont bien complémentaires et qu'ils sont liés.

Le tableau 4.11 nous montre les temps de calcul nécessaires pour trouver la solution optimale de chacune des configurations précédentes. Nous constatons que les temps de calcul sont beaucoup plus courts que lors de la résolution du programme complémentaire (trouver la réjection maximale, cf. tableau 4.4). En effet, le temps le plus long ici n'est que de 17 minutes au lieu de 3 jours.

4.2.2.2 Consommation des ressources du FPGA

Dans cette section nous allons voir quelle est la consommation réelle de ressources grâce aux rapports générés par Vivado lors de la synthèse du bitstream.

Le tableau 4.12 donne la consommation de ressources pour chacune des expérimentations. Ces consommations ne concernent que les étages de filtres : le générateur de nombres pseudo-aléatoires et le stockage des données ne sont pas inclus.

TABLE 4.12 – Occupation des ressources. La dernière colonne indique les ressources disponibles sur le Zynq-7010 de la Redpitaya.

n	Ressources	MIN/40	MIN/60	MIN/80	MIN/100	*Zynq 7010*
1	LUT	343	334	772	-	*17600*
	BRAM	1	1	1	-	*120*
	DSP	27	39	55	-	*80*
2	LUT	1252	2862	5099	640	*17600*
	BRAM	2	2	2	2	*120*
	DSP	0	0	0	66	*80*
3	LUT	891	2148	2023	2448	*17600*
	BRAM	3	3	3	3	*120*
	DSP	0	0	19	27	*80*
4	LUT	662	1729	2451	2893	*17600*
	BRAM	4	4	4	4	*120*
	DSP	0	0	7	19	*80*
5	LUT	-	1259	2602	2505	*17600*
	BRAM	-	5	5	5	*120*
	DSP	-	0	0	19	*80*

Étant donné qu'il n'y a pas de solution optimale pour le cas $i = 1$ de MIN/100, il est impossible d'avoir la consommation de ressources pour cette situation là. De même pour le cas $i = 5$ de MIN/40 qui n'est pas différent du cas $i = 4$.

On peut constater de nouveau que la consommation de BRAM dépend du nombre de filtres n de la configuration. Pour chaque étage, nous consommons une BRAM pour stocker les différents jeux de coefficients.

Tout comme dans le cas précédent, il est compliqué de comparer la consommation des ressources car il faut déterminer à combien de LUT correspond l'utilisation d'un DSP. Si on reprend notre précédente hypothèse à savoir que 1 DSP correspond à 122 LUT, on obtient le tableau 4.13 :

Le tableau 4.13 nous montre, tout d'abord, que la consommation de ressources est bien de plus en plus faible quand on rajoute des étages de filtrage comme l'annonçaient les solutions optimales vues dans la section précédente. Cependant, le gain de place n'est pas aussi important que prévu. Le tableau 4.14 représente le pourcentage de place occupée en fonction de la configuration monolithique. Pour chaque cas nous calculons le ratio pour le cas abstrait et pour le cas réel.

Dans le cas de MIN/40 les résultats obtenus sont légèrement meilleurs à ceux attendus. Cependant pour MIN/60 et MIN/80, le gain de place entre le filtre monolithique et la première cascade ($i = 2$) n'est pas celle attendue : au lieu de gagner environ 50 % de ressources, on en gagne respectivement 44 % et 32 %. Cependant, la différence se réduit grandement dans les dernière cascades ($i = 5$) : pour MIN/60, on arrive à l'objectif attendu et pour MIN/80, on est 5% en dessous de la valeur théorique.

TABLE 4.13 – Occupation des ressources en équivalent LUT avec l'hypothèse que 1 DSP = 122 LUT

n	Ressources	MIN/40	MIN/60	MIN/80	MIN/100	*Zynq 7010*
1	éq. LUT	3637	5092	7482	-	*17600*
	BRAM	1	1	1	-	*120*
2	éq. LUT	1252	2862	5099	8692	*17600*
	BRAM	2	2	2	2	*120*
3	éq. LUT	891	2148	4341	5742	*17600*
	BRAM	3	3	3	3	*120*
4	éq. LUT	662	1729	3305	5211	*17600*
	BRAM	4	4	4	4	*120*
5	éq. LUT	-	1259	2602	4823	*17600*
	BRAM	-	5	5	5	*120*

TABLE 4.14 – Amélioration de l'occupation des ressources en pourcentage par rapport au cas monolithique

n		MIN/40	MIN/60	MIN/80
1	Solveur	0%	0%	0%
	Vivado	0%	0%	0%
2	Solveur	64%	52%	49%
	Vivado	66%	44%	32%
3	Solveur	70%	62%	60%
	Vivado	76%	58%	42%
4	Solveur	77%	70%	66%
	Vivado	82%	66%	56%
5	Solveur	-	75%	70%
	Vivado	-	75%	65%

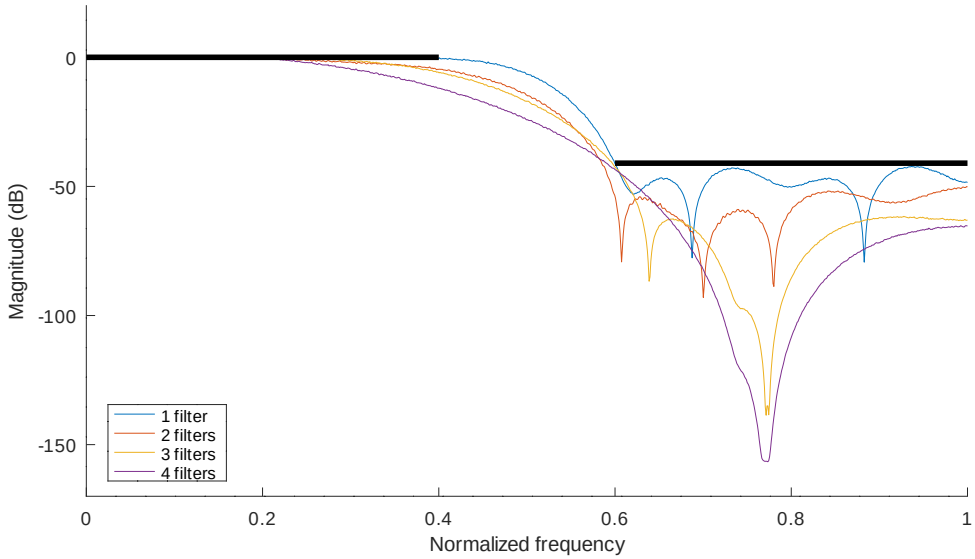


FIGURE 4.6 – Spectre du signal pour MIN/40

Malgré ce petit écart, la tendance reste la même : à performance égale, plus on a de filtres, plus on gagne de place. Plus précisément, on gagne entre 80% et 60% selon les cas. Nous pouvons supposer qu'en augmentant la limite de n, on finirait par toujours atteindre les 80% de gain de place comme pour MIN/40.

4.2.2.3 Données issues du traitement sur la carte Redpitaya

Dans cette dernière partie, nous allons analyser les résultats obtenus par la carte Redpitaya. Nous obtenons ces résultats toujours grâce à la méthode de travail présentée dans la section 4.1.

Les figures 4.6, 4.7, 4.8 et 4.9 représentent respectivement les résultats pour MIN/40, MIN/60, MIN/80 et MIN/100. Dans chacune de ces figures, on a tracé en noir le gabarit attendu des filtres et les courbes en couleur sont les fonctions de transfert moyennées issues des données expérimentales. Les couleurs, quant à elles, correspondent aux différentes valeurs de i ($1 \leq i \leq 5$). Les deux exceptions sont pour le cas $i = 5$ de MIN/40 car la solution est identique avec le cas $i = 4$ et pour $i = 1$ de MIN/100 car aucune solution n'a été trouvée.

Nous constatons une fois encore que les résultats expérimentaux coïncident avec les résultats théoriques. En effet, dans chaque expérimentation (MIN/40, MIN/60, MIN/80, MIN/100) la contrainte de réjection minimale est respectée. De plus les courbes sont toutes rapprochées avec une réjection proche (moins de 10 dB d'écart entre la moins bonne et la meilleure).

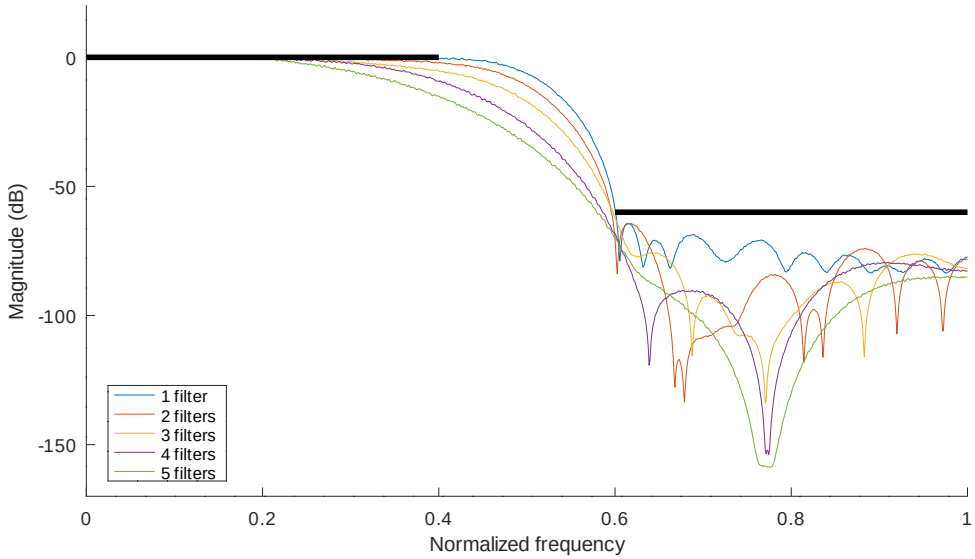


FIGURE 4.7 – Spectre du signal pour MIN/60

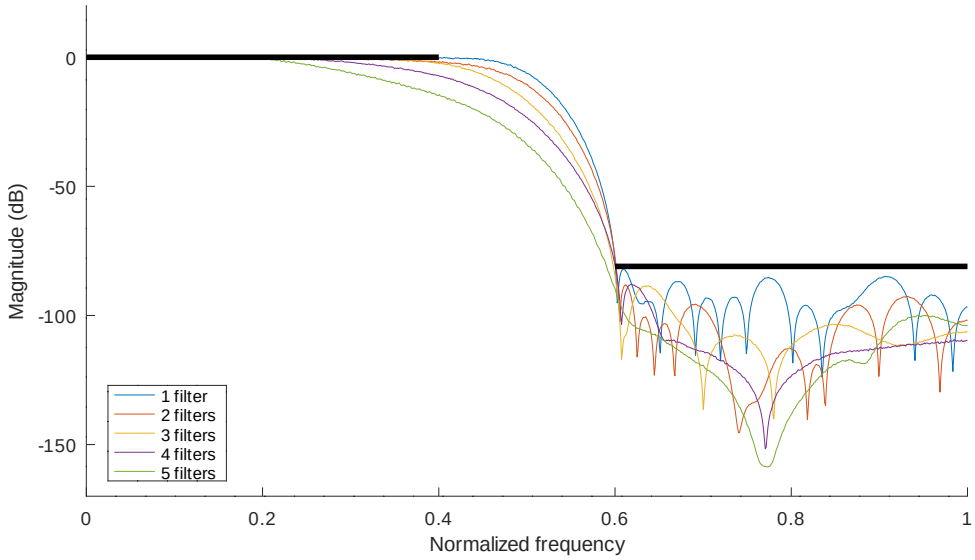


FIGURE 4.8 – Spectre du signal pour MIN/80

On s'aperçoit encore que dans la bande passante, on a une forte déviation (> 10 dB) quand on cascade plus de deux filtres. On verra dans la section 4.3 comment corriger ce problème.

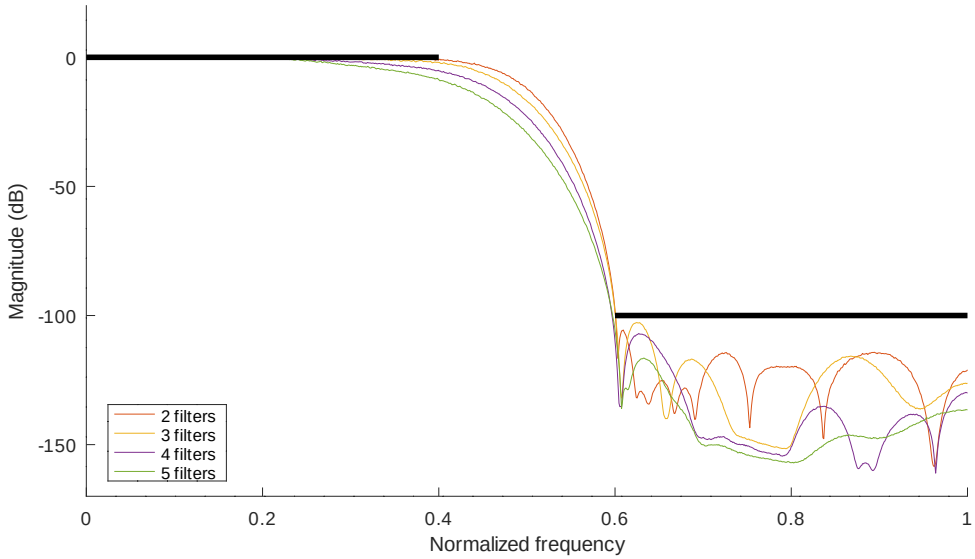


FIGURE 4.9 – Spectre du signal pour MIN/100

4.2.3 Optimisation multi-objectifs

Dans les deux dernières sous-sections, nous présentions les résultats pour savoir d'une part comment maximiser la réjection, d'autre part comment minimiser la consommation de ressources. La suite logique serait de savoir comment avoir la meilleure réjection tout en minimisant l'utilisation de ressources. C'est ce qu'on appelle de l'optimisation multi-objectifs.

Répondre à cette question revient à déterminer quel est le meilleur compromis entre réjection maximale et consommation minimale. Selon les besoins, il faudra accepter un compromis sur l'un ou l'autre des paramètres pour avoir une chaîne qui réponde complètement au besoin du traitement.

Un outil utilisé pour nous aider à trouver ce meilleur compromis est le front de Pareto. Il s'agit de représenter graphiquement l'ensemble des solutions trouvées par un nuage de points et de tracer la limite (ou frontière) optimale au delà de laquelle il n'existe pas de solution. Ce faisant, on détermine graphiquement quel est le meilleur choix possible pour trouver des designs optimaux.

Dans notre cas, nous n'allons pas avoir un ensemble de points dont il faudrait graphiquement déterminer lesquels sont optimaux, puisque chacun des points est déjà un optimal. En effet, la résolution de notre programme quadratique nous garantit d'avoir une solution optimale. Les points obtenus dessineront directement le front de Pareto.

Afin d'avoir des fronts de Pareto suffisamment précis, nous avons augmenté le nombre d'instances de nos problèmes. Nous avons trouvé une solution optimale de MIN/10 à MIN/200 par pas de 10 dB et de MAX/100 à MAX/2000 par pas de 100 u.a. Pour chacun

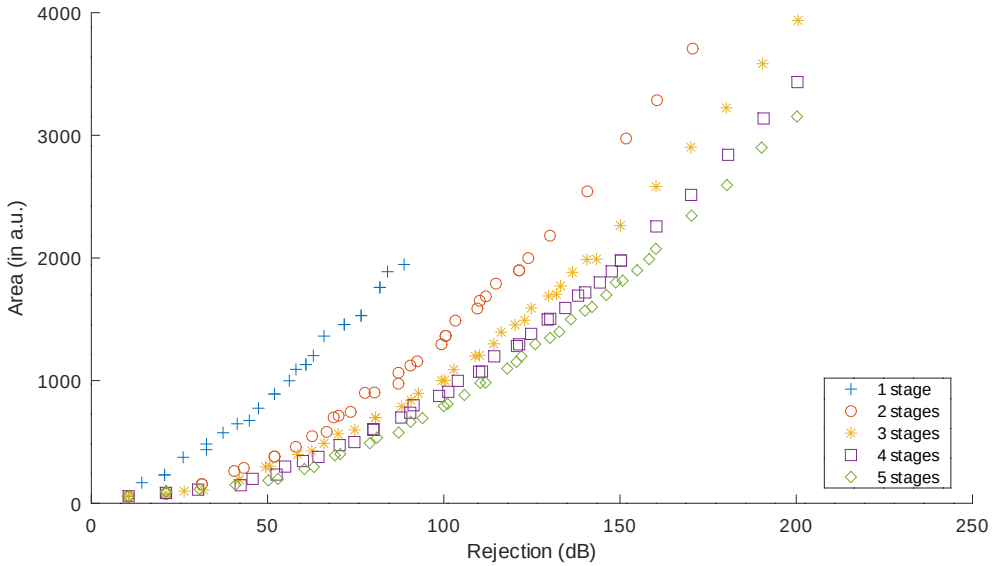


FIGURE 4.10 – Fronts de Pareto obtenus en faisant varier le nombre d'étages n entre 1 et 5

de ces cas, nous faisons varier n de 1 à 5. La figure 4.10 montre les différents fronts de Pareto obtenus avec les différents n. Plus on se rapproche du coin en bas à droite, plus la solution est bonne car on a une grande réjection pour une surface minimale.

Nous constatons ici encore que la cascade améliore grandement l'efficacité de la réjection. Pour le cas monolithique ($n = 1$), nous sommes vite limités car le meilleur filtre disponible parmi ceux que nous avons générés n'atteint qu'environ 90 dB de réjection pour 2000 u.a. d'espace occupé tandis que pour la même réjection, avec $n = 5$ on n'occupe que 1000 u.a. d'espace et pour la même surface avec $n = 5$ à nouveau, on atteint jusqu'à 160 dB de réjection.

Dans les cas avec cascade ($n \geq 2$), on constate qu'avant 25 dB, les résultats sont sensiblement identiques. En revanche plus on veut rejeter de bruit, plus la différence est grande. Si on regarde le dernier point avec $n = 2$, pour rejeter 170 dB, il faut utiliser 3700 u.a. alors que pour $n = 3$, il ne faut que 2900 u.a., pour $n = 4$ seulement 2500 u.a. et pour $n = 5$ on consomme seulement à 2300 u.a.. On gagne donc 1400 u.a. entre la solution avec 2 étages est celle avec 5 étages. Cependant, cet écart se réduit d'étage en étage : toujours avec le même exemple, entre $n = 2$ et $n = 3$, on gagne 800 u.a. alors qu'entre $n = 3$ et $n = 4$ on ne gagne que 400 u.a., et entre $n = 4$ et $n = 5$ le gain est seulement de 200 u.a.. Cela explique pourquoi les courbes se rapprochent de plus en plus et nous laissent penser que si on avait pu tester le cas $n \geq 6$, on aurait pu avoir des courbes qui se superposent et donc avoir trouvé une valeur de n optimale.

Un dernier constat qu'on peut faire concerne la forme des courbes. Pour le cas $n = 1$, le front semble être linéaire alors que pour les cascades où $n \geq 2$, on voit que la forme est

parabolique. Pour le cas monolithique la linéarité s'explique par la forme de la pyramide de réjection (figure 3.8b) que nous avions présentée section 3.1 du chapitre 3. Sur cette pyramide, on voit que l'arête (les filtres optimaux) croît linéairement en fonction du nombre de coefficients et du nombre de bits. Dans le cas où $n = 1$, on va donc se contenter de placer des filtres sélectionnés sur cette arête, donc le front de Pareto sera de géométrie similaire. En revanche, pour les cascades de filtres, comme entre chaque étage on peut réduire la taille des données et ainsi diminuer la consommation de ressources pour les traitements suivants, la chaîne n'est plus un simple choix de filtres, c'est pourquoi on observe cette forme parabolique.

Grâce à ces différents fronts de Pareto, nous avons une confirmation que les solutions avec des filtres en cascades sont toujours meilleures que les solutions monolithiques, et que les meilleurs résultats sont obtenus pour $n = 5$.

4.2.4 Synthèse des résultats

Dans cette section nous allons dresser un bilan global de nos expérimentations. Dans les sections précédentes, nous avons traité les deux versions du programme quadratique. Dans un premier temps nous nous sommes attachés à obtenir la meilleure réjection possible étant donnée une surface imposée et dans un second temps, nous avons cherché à obtenir la plus faible consommation de ressources tout en atteignant un niveau de réjection donné.

Toutes les expérimentations que nous avons faites (MAX/500 à MAX/1500 et MIN/40 à MIN/100) démontrent que le modèle est suffisamment proche de la réalité pour que les résultats obtenus par le solveur puissent être valables lors de leur application. En particulier, la modélisation simple de la consommation de ressources est suffisante pour être corrélé avec l'utilisation de LUT dans le FPGA.

Grâce à ce modèle suffisamment abstrait mais aussi suffisamment proche de la réalité nous disposons de suffisamment de degrés de liberté pour créer des chaînes de traitement non triviales. Plus précisément, nous avons remarqué un schéma d'attribution des filtres : la chaîne commence par un filtre le plus petit possible pour garder des données les plus petites possibles, puis un deuxième filtre beaucoup plus gros s'occupe de rejeter beaucoup de bruit, et pour finir les filtres suivants seront plus petits pour continuer à rejeter de plus en plus de bruit tout en limitant la taille de données.

Le dernier point remarquable est que nos expériences montrent qu'il est toujours préférable de cascader des filtres, que ce soit pour maximiser la réjection ou pour minimiser la consommation de ressources. Cette conclusion est en cohérence avec les résultats passés d'autres chercheurs tel que nous l'avons identifié au chapitre 1 ([59, 96, 60]). Le fait que notre méthode confirme ces résultats démontre que cette approche est viable et qu'il doit être possible de la généraliser.

Ces trois points nous permettent d'avoir confiance dans la méthodologie que nous proposons. Nous avons modélisé une situation, nous avons défini les contraintes liées au

problème ainsi que l'objectif à optimiser. Le modèle est suffisamment robuste pour que les résultats obtenus après la résolution du modèle puissent être directement appliqués. Donc pour traiter n'importe quel problème, il suffit de pouvoir évaluer les solutions répondant à des contraintes linéaires et/ou quadratiques, afin que nous puissions chercher des solutions optimales.

Toutefois, nous voyons une limite concernant le temps de calcul. Dans nos expérimentations sur des instances restreintes (peu de filtres possibles et d'étages, les tailles maximales ou les réjections à atteindre sont faibles), les temps de calcul pour trouver une solution optimale peuvent prendre plusieurs jours. Si jamais nous voulions augmenter le nombre d'étages, cela prendrait trop de temps. Pour remédier à cela, nous pourrions éventuellement définir une heuristique grâce au schéma d'attribution des filtres que nous avons pu observer.

4.2.5 Confrontation des résultats avec des outils existants

Nous avons vu dans la section précédente que notre méthodologie est capable de produire des résultats pertinents. Il nous reste à savoir si notre approche est au moins aussi bonne que ce qui existe déjà. Nous allons donc comparer nos résultats avec ceux du FIR Compiler de Vivado (v.2018.2) en terme de consommation de ressources.

Pour ce faire, nous allons nous intéresser aux filtres monolithiques. En prenant les mêmes jeux de coefficients, nous allons vérifier que la réjection est bien identique et que la consommation de ressources est au moins aussi bonne. Le cas MIN/100 est exclu car il ne présente pas de solution monolithique. Par ailleurs, les cascades de filtres ne peuvent être comparées car l'outil de synthèse de filtres de Xilinx n'est pas conçu pour gérer ce genre de situation.

Le tableau 4.15 montre les résultats obtenus. Le FIR Compiler n'utilise jamais de BRAM alors que notre implémentation en utilise. Cette différence s'explique par le fait que notre FIR est dynamiquement reconfigurable depuis le PS : on n'a pas besoin de re-synthétiser tout le design pour changer les coefficients, on peut le faire depuis l'OS embarqué. Au contraire, FIR Compiler de Xilinx se contente de coder en dur ces coefficients lors de la création du bitstream, ce qui oblige de re-générer l' image binaire si on change les coefficients.

On peut constater qu'on consomme environ 150 LUT de plus que la solution de Xilinx. Cette différence est également attribuée au fait que notre FIR est reconfigurable dynamiquement. La consommation des DSP, ressource la plus critique, est strictement identique entre les deux implémentations. Pour ces deux raisons, nous concluons que notre solution est aussi bonne que celle que peut proposer Xilinx.

Avec ce dernier point, nous nous sommes assurés que notre implémentation est aussi bonne que celle d'un autre outil existant. À cela s'ajoute le fait que nos résultats sont cohérents avec ceux attendus et même avec ceux déjà trouvés sur les cascades de filtres [59, 96, 60].

TABLE 4.15 – Comparaison de la consommation de ressources entre le FIR Compiler de Xilinx et notre implémentation de FIR

	Xilinx			Our FIR block		
	LUT	BRAM	DSP	LUT	BRAM	DSP
MAX/500	177	0	21	249	1	21
MAX/1000	306	0	37	453	1	37
MAX/1500	418	0	47	627	1	47
MIN/40	225	0	27	347	1	27
MIN/60	322	0	39	334	1	39
MIN/80	482	0	55	772	1	55

TABLE 4.16 – Configurations (C_i, π_i^C, π_i^S), réjection et surface (en unités arbitraires) pour MIN/20

n	$i = 1$	$i = 2$	$i = 3$	Réjection	Surface
2	(3, 3, 15)	(3, 3, 0)	-	21 dB	78
3	(3, 3, 14)	(3, 3, 0)	-	21 dB	81

En l'état, nous avons prouvé que notre méthodologie fonctionne dans ce contexte. Toutefois les résultats peuvent être encore améliorés en affinant encore un peu plus le modèle.

4.3 Améliorations possibles de notre modélisation

Au cours de nos expérimentations, nous avons pu relever quelques détails dans nos résultats qui indiquaient que nous pourrions adapter notre modèle pour l'améliorer. Dans cette section, nous allons donner deux exemples de points améliorables avec la correction apportée et l'impact de celle-ci sur les résultats.

4.3.1 Amélioration d'une contrainte du programme quadratique

Lorsque nous avons fait nos expérimentations pour calculer les fronts de Pareto, notre attention a été attirée par des cas troublants. Prenons le cas $n = 2$ et $n = 3$ de MIN/20. Le tableau 4.16 donne les deux solutions trouvées.

La solution optimale est donnée pour $n = 2$ alors qu'à $n = 3$ on obtient un résultat moins bon (la surface occupée est plus grande). La seule différence entre les deux solutions est qu'on décale un bit de moins dans $n = 3$, ce qui explique la différence de surface. Cependant ce cas ne devrait pas se produire car, normalement, rien n'empêche le solveur de reprendre la solution précédente.

Nous avons donc cherché dans le programme quadratique quelle contrainte pouvait poser problème. Si on regarde la contrainte 3.20 (rappelée ci-après), on peut voir qu'il y un souci. Pour s'en convaincre, nous allons la développer pour l'exemple donné.

$$\pi_i^+ \geq 1 + \sum_{k=1}^{i} \left(\frac{r_i}{6} + 1 \right), \qquad \forall i \in [1, n] \tag{3.20}$$

Dans le cas où $n = 2$, on sait que $r_1 = r_2 = 10,5$ donc si on déroule la contrainte 3.20 on obtient l'équation 4.2. Dans le cas $n = 3$, on doit juste rajouter $\frac{r_3}{6}$ à l'équation précédente puisque les deux filtres choisis sont les mêmes. Donc $r_3 = 0$ car aucun filtre n'a été choisi à l'étage 3 donc on obtient l'équation 4.3. On voit ainsi clairement que la contrainte est plus forte que nécessaire car elle force à avoir un bit de plus à chaque étage même si on ne place pas de filtre.

$$\pi_i^+ \geq 3 + \frac{10,5}{3} = 6,5 \tag{4.2}$$

$$\pi_i^+ \geq 3 + \frac{10,5}{3} + \left(\frac{0}{6} + 1 \right) = 7,5 \tag{4.3}$$

Pour corriger cela, il faudrait changer la contrainte 3.20 par la nouvelle contrainte 4.4. Avec cette nouvelle version, si aucun filtre n'est choisi à l'étage i la somme des δ_{ij} vaudra 0 et aucun bit ne sera inutilement ajouté. En revanche, si un filtre est choisi, cette somme vaudra bien 1 et pas plus car la contrainte 3.21 nous le garantit.

$$\pi_i^+ \geq 1 + \sum_{k=1}^{i} \left(\frac{r_i}{6} + \sum_{j=1}^{p} (\delta_{ij}) \right), \qquad \forall i \in [1, n] \tag{4.4}$$

Le problème de cette correction est que cela augmente considérablement le temps d'exécution du solveur : le temps nécessaire pour trouver la solution peut être doublé pour les grandes instance (on passe de 3 jours de calculs à 6 pour le cas $n = 5$ de MAX/1500). De plus, cet impact n'est visible que dans certains cas qui peuvent être corrigés facilement à la main. C'est pourquoi nous avons gardé la version précédente, plus rapide, pour aboutir à une solution optimale.

4.3.2 Déviation importante dans la bande passante

Un autre point améliorable concerne le gabarit des filtres. Nous avons mentionné dans les sections 4.2.2.3 et 4.2.1.3 qu'il y a une forte déviation (plus de 10 dB) dans la bande

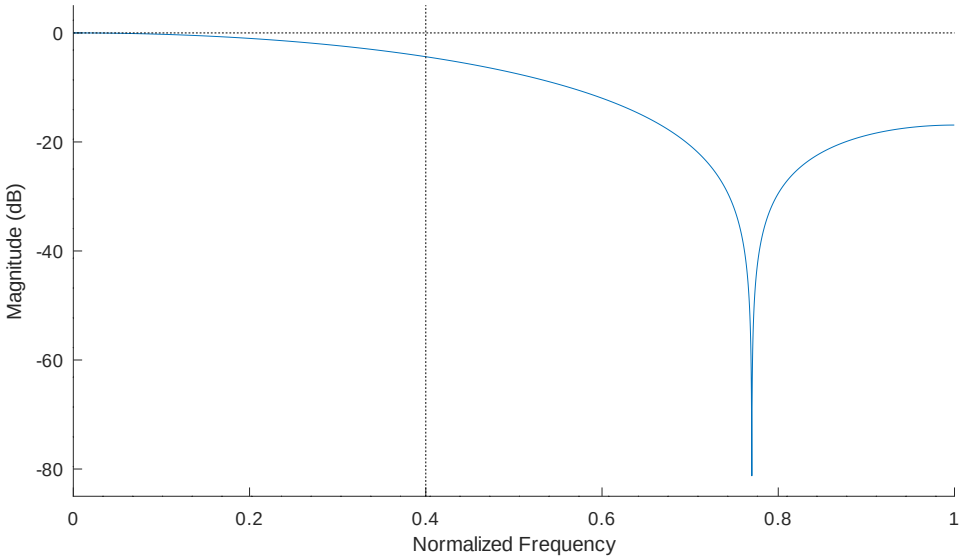


FIGURE 4.11 – Réponse impulsionnelle d'un filtre avec 3 coefficients codés sur 3 bits

passante quand on s'approche de la bande de transition. Bien que nous ayons essayé d'empêcher cette dérive en pénalisant les filtres qui présentaient ce genre de problème, cela n'a pas été suffisant. En effet, la figure 4.11 montre la réponse impulsionnelle du filtre avec 3 coefficients codés sur 3 bits et générés par la fonction `firls`.

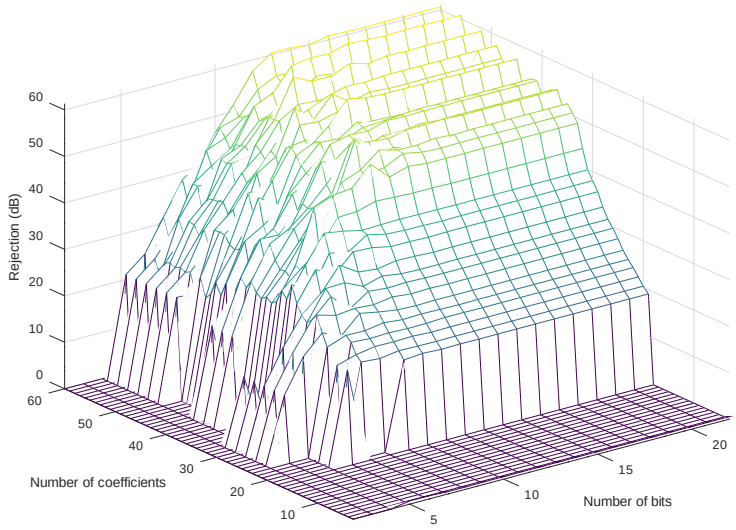
On observe, à la fin de la bande passante, une chute de près de 5 dB avant d'atteindre la bande de transition. Si on place x fois ce filtre, dans la chaîne, on aura au minimum $x \times 5$ dB de déviation à cet endroit. Dans certaines applications, il peut être inacceptable d'avoir une si forte atténuation. Si on veut éviter d'avoir un tel impact sur nos filtres, il est impératif de changer l'évaluation de notre critère de réjection pour un jeu de coefficients.

Nous avons donc décidé d'écarter les filtres qui ont plus de 1 dB d'amplitude (*ripples)* dans la bande passante, c'est à dire plus de 1 dB entre la valeur maximale et la valeur minimale. Ainsi, le filtre de la figure 4.11 est éliminé car il présente une déviation trop forte.

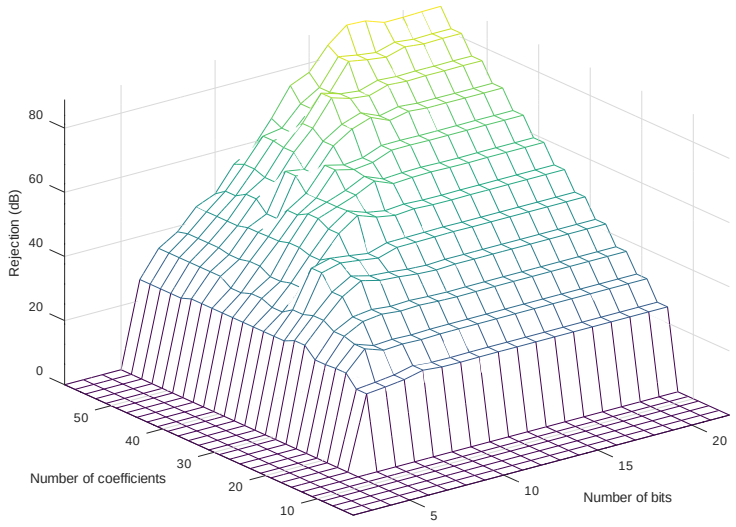
Les figures 4.12a et 4.12b donnent respectivement les nouvelles valeurs de réjection pour les filtres générés avec `fir1` et `firls`. Les valeurs à 0 dB sur les cotés représentent tous les filtres qui ont été éliminés sur ce critère, passant ainsi d'un total de 1827 jeux de coefficients à seulement 1133.

Nous avons donc relancé les expérimentations avec les nouvelles valeurs de réjection. Les résultats obtenus étant différents, nous allons les détailler.

Les tableaux 4.17 donnent les nouveaux résultats optimaux obtenus avec le solveur pour MAX/500, MAX/1000 et MAX/1500 et les tableaux 4.18 pour MIN/40, MIN/60, MIN/80 et MIN/100. Afin d'alléger les tableaux, nous n'avons pas mis les lignes qui



(a) Filtres générés avec `fir1`



(b) Filtres générés avec `firls`

FIGURE 4.12 – Réjection des filtres en fonction du nombre de coefficients et du nombre de bits avec le nouveau critère

étaient identiques, donc quand les tableaux s'arrêtent pour une valeur de n inférieure à 5, cela signifie que les solutions suivantes sont les mêmes que celles de la ligne précédente.

La première différence marquante est qu'on trouve une valeur de n optimale pour

TABLE 4.17 – Configurations (C_i, π_i^C, π_i^S), réjection et surface (en unités arbitraires) pour MAX/500, MAX/1000 et MAX/1500 avec les nouvelles valeurs de réjection

n	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	Réjection	Surface
1	(21, 7, 0)	-	-	-	-	32 dB	483
2	(3, 5, 18)	(33, 10, 0)	-	-	-	48 dB	492
3	(3, 5, 18)	(19, 7, 1)	(15, 7, 0)	-	-	56 dB	493

n	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	Réjection	Surface
1	(37, 11, 0)	-	-	-	-	56 dB	999
2	(15, 8, 17)	(35, 11, 0)	-	-	-	80 dB	990
3	(3, 13, 26)	(31, 9, 1)	(27, 9, 0)	-	-	92 dB	999
4	(3, 5, 18)	(19, 7, 1)	(19, 7, 0)	(19, 7, 0)	-	98 dB	994

n	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	Réjection	Surface
1	(47, 15, 0)	-	-	-	-	71 dB	1457
2	(19, 6, 15)	(51, 14, 0)	-	-	-	102 dB	1489
3	(15, 9, 18)	(31, 8, 0)	(27, 9, 0)	-	-	116 dB	1488
4	(3, 9, 22)	(31, 9, 1)	(27, 9, 0)	(19, 7, 0)	-	125 dB	1500

chacune des expérimentations. En effet pour MIN/40, la solution optimale se trouve à $n = 2$: au-delà, les solutions restent identiques. Pour MAX/500, MIN/60 et MIN/80, l'optimal est à $n = 3$ et pour MAX/1000, MAX/1500 et MIN/100, il est à $n = 4$.

Malgré les changements de résultats, on peut tirer les mêmes conclusions qu'avant :
- les cascades sont meilleures que le cas monolithique
- on a le même schéma d'affectation : un petit filtre avec un grand décalage, puis un gros filtre suivi d'une succession de filtres plus petits

Ce dernier point n'est pas tout à fait juste pour les cas MIN/60 et MIN/100 où, pour respectivement $n = 3$ et $n = 4$, le filtre le plus gros n'est pas en deuxième position, mais à la place le même filtre est utilisé pour réduire encore un peu la taille des données tout en rejetant plus de bruits qu'au premier étage.

Une fois les nouvelles solutions abstraites calculées, nous pouvons les utiliser pour obtenir les résultats expérimentaux pour les sept instances différentes du problème à résoudre.

La figure 4.13 présente les résultats pour MAX/500, MAX/1000 et MAX/1500 quant à la figure 4.14 elle présente les résultats pour MIN/40, MIN/60, MIN/80 et MIN/100.

Quand on regarde le comportement dans la bande passante, on voit que tous les filtres respectent la nouvelle contrainte imposée de ne pas avoir plus de 1 dB d'amplitude.

D'autre part, les conclusions précédentes restent valides. Nous voyons qu'il est préférable de cascader des filtres quand on veut maximiser la réjection ou quand on cherche à minimiser la consommation de ressources. En outre, toutes les expérimentations respectent les contraintes d'occupation d'espace et de réjection.

TABLE 4.18 – Configurations (C_i, π_i^C, π_i^S), réjection et surface (en unités arbitraires) pour MIN/40, MIN/60, MIN/80 et MIN/100 avec les nouvelles valeurs de réjection

n	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	Réjection	Surface
1	(27, 8, 0)	-	-	-	-	41 dB	648
2	(3, 5, 18)	(27, 8, 0)	-	-	-	42 dB	360

n	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	Réjection	Surface
1	(39, 13, 0)	-	-	-	-	60 dB	1131
2	(15, 6, 16)	(23, 9, 0)	-	-	-	60 dB	675
3	(3, 5, 18)	(15, 6, 2)	(23, 8, 0)	-	-	60 dB	543

n	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	Réjection	Surface
1	(55, 16, 0)	-	-	-	-	81 dB	1760
2	(15, 8, 17)	(35, 11, 0)	-	-	-	80 dB	990
3	(3, 7, 20)	(31, 9, 1)	(19, 7, 0)	-	-	80 dB	783

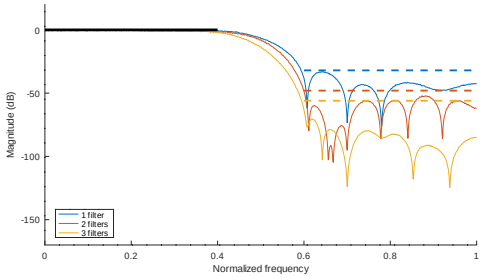
n	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	Réjection	Surface
1	-	-	-	-	-	-	-
2	(27, 9, 15)	(35, 11, 0)	-	-	-	100 dB	1410
3	(3, 5, 18)	(35, 11, 1)	(27, 9, 0)	-	-	100 dB	1147
4	(3, 5, 18)	(15, 6, 2)	(27, 9, 0)	(19, 7, 0)	-	100 dB	1067

Nous n'allons pas présenter les tableaux de consommation de ressources car bien que les valeurs soient différentes, les conclusions restent les mêmes. En revanche les tableaux 4.19 présentent les nouveaux temps de calcul pour les différentes expérimentations.

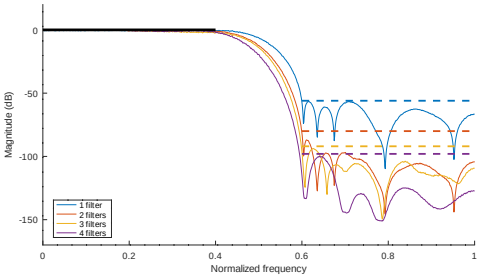
On observe cette fois-ci une différence radicale en ce qui concerne les temps de calcul nécessaires. On passe en effet de 3 jours de calculs dans le pire des cas à seulement 18 minutes. Cette différence s'explique notamment par le fait qu'on soit passé d'un total de plus de 1800 filtres différents à environ 1100, ce qui représente une diminution de près de 30% des cas à analyser. Du fait de cette réduction, le solveur a moins de choix à tester entre les différents étages et la recherche de la solution optimale est donc accélérée car on a réduit l'espace de recherche.

Ce qui ressort de ces nouvelles expérimentations est que, globalement, les conclusions sont similaires : cascader des filtres est préférable en terme de réjection et d'économie de ressources. De plus, nous avons démontré que notre méthodologie peut s'adapter à des changements de contexte. Ici, nous avons dû contraindre plus fortement les comportements des filtres dans la bande passante, mais nous n'avons pas été obligés de repartir de zéro (refaire un nouveau modèle, l'adapter en programme linéaire ou quadratique, etc...). Adapter le critère de réjection des filtres à suffi à répondre au nouveau besoin.

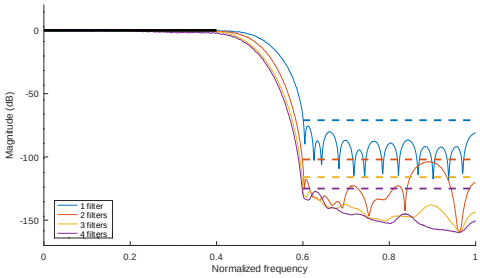
À la suite de nos différentes expérimentations et des deux adaptations présentées, on peut dire que notre méthodologie donne des résultats convaincants et qu'il est possible de l'adapter si besoin. Grâce à celà, on pourrait étendre le modèle pour tenir compte en plus de la décimation du signal. Ainsi nous aurions modélisé l'intégralité d'un étage de



(a) Résultats pour MAX/500

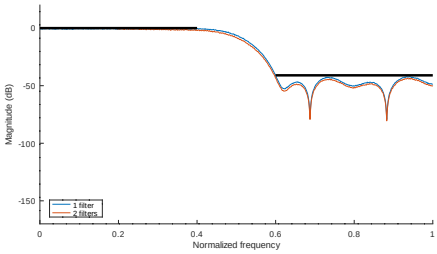


(b) Résultats pour MAX/1000

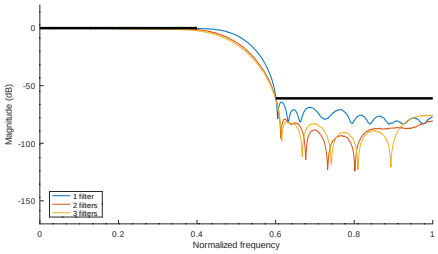


(c) Résultats pour MAX/1500

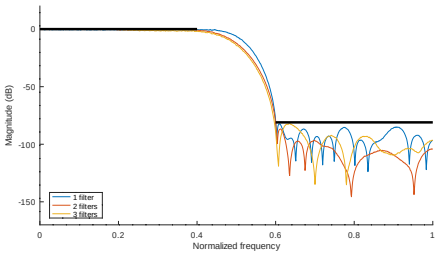
FIGURE 4.13 – Résultats issus de la RedPitaya avec les nouvelles solutions optimales des instances MAX/500, MAX/1000 et MAX/1500



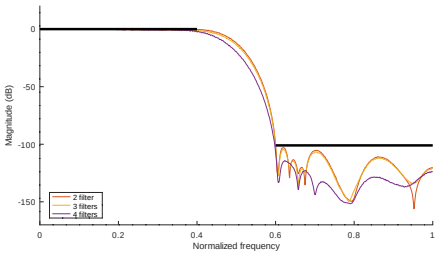
(a) Résultats pour MIN/40



(b) Résultats pour MIN/60



(c) Résultats pour MIN/80



(d) Résultats pour MIN/100

FIGURE 4.14 – Résultats issus de la RedPitaya avec les nouvelles solutions optimales des instances MIN/40, MIN/60, MIN/80 et MIN/100

TABLE 4.19 – Temps pour résoudre le programme quadratique avec Gurobi en tenant compte du nouveau critère

n	Temps (MAX/500)	Temps (MAX/1000)	Temps (MAX/1500)
1	0,01 s	0,02 s	0,03 s
2	0,1 s	1 s	2 s
3	5 s	27 s	351 s ($\approx$ 6 min)
4	-	141 s ($\approx$ 3 min)	1134 s ($\approx$ 18 min)

n	Temps (MIN/40)	Temps (MIN/60)	Temps (MIN/80)	Temps (MIN/100)
1	0,04 s	0,01 s	0,01 s	-
2	2,7 s	2,4 s	2,4 s	0,8 s
3	-	7 s	7 s	18 s
4	-	-	-	220 s ($\approx$ 3 min)

filtrage : FIR, décimation puis shift.

Il serait également possible de changer complètement de contexte et d'essayer de résoudre une nouvelle problématique, du moment qu'on est capable de modéliser le problème et d'évaluer les solutions.

Quatrième partie

Synthèse et conclusion

Chapitre 5

Conclusion

Le but de cette thèse était de proposer une nouvelle approche permettant d'optimiser une chaîne de traitement numérique du signal. Pour la tester, nous avons étudié un cas concret : les cascades de filtres. Dans la partie II, nous avons présenté en détail le modèle décrivant le problème ainsi que les contraintes spécifiques au contexte des FPGA.

Dans un premier temps, nous avons défini une modélisation abstraite d'un filtre. Dans le modèle que nous avons créé, nous caractérisons les filtres par leur nombre de coefficients et par la taille de ceux-ci exprimée en bits, la quantité de bruit filtrée (la réjection), par une taille de données en entrée et en sortie et enfin par la surface de silicium occupée représentant la consommation de ressources dans le FPGA (exprimée en unités arbitraires).

En partant de ce modèle de FIR, nous avons défini les cascades de filtres comme étant une succession de filtres suivis éventuellement par un décalage logique permettant de réduire la taille des données en sortie du filtrage pour permettre d'économiser des ressources dans l'étage suivant.

Grâce à cette modélisation, nous avons créé deux programmes quadratiques (3.28 et 3.29) qui répondent à deux questions différentes : comment maximiser la réjection à taille fixée et comment minimiser la taille à réjection fixée.

Grâce aux modèles quadratiques il est possible de trouver des solutions abstraites optimales et à partir d'elles, de générer automatiquement des designs FPGA complets. Il reste ensuite à automatiser l'expérimentation grâce un script Shell. En dernier lieu, nous post-traitons les données afin de nous assurer que les résultats expérimentaux coïncident avec les résultats abstraits.

Afin de nous assurer de la pertinence de notre approche, nous avons produit plusieurs données expérimentales. Une première série avec pour objectif de maximiser la réjection avec 500, 1000 ou 1500 unités arbitraires comme taille maximale (MAX/500, MAX/1000 et MAX/1500), puis une seconde série avec cette fois-ci le but de minimiser la place occupée pour une réjection minimale à atteindre de 40, 60, 80 ou 100 dB (MIN/40, MIN/60, MIN/80 et MIN/100).

Pour chaque expérimentation, nous faisons varier le nombre d'étages n de 1 à 5. Ce faisant, nous pouvons comparer les résultats des filtres monolithiques ($n = 1$) avec les cascades de filtres. De plus, nous pouvons essayer de trouver une valeur optimale de n au delà de laquelle il n'y aurait plus d'améliorations.

Le constat global qu'on peut tirer de ces sept expérimentations est que les résultats expérimentaux coïncident avec nos attentes en terme de réjection. Dans les faits, les résultats issus du FPGA sont même souvent meilleurs que ceux attendus. Un autre constat est que les performances des filtres monolithiques sont toujours moins bonnes (en terme de réjection et d'espace) que les cascades de filtres. De plus, on constate que plus il y a d'étages, meilleurs sont les résultats. Cette conclusion confirme celles d'autres chercheurs, ce qui nous permet d'avoir confiance en nos résultats et en notre méthodologie.

Lors de la conception du modèle, nous avions fait une hypothèse forte : la place qu'occupe un filtre se limite au stockage de coefficients et des données nécessaires pour faire les produits de convolutions. En regardant la consommation réelle de ressources (en terme de composants du FPGA : LUT, BRAM, DSP...), on remarque que l'estimation abstraite coïncide avec la réalité. Cela signifie que notre hypothèse était juste. L'abstraction nous a permis de laisser au programme quadratique plus de libertés dans le choix des solutions sans pour autant nuire aux résultats expérimentaux.

Toutefois, notre approche est limitée par le temps de calculs pour trouver les solutions optimales abstraites. Pour les petites instances (MIN/40 à MIN/100, MAX/500 et MAX/1000 avec $n < 5$) les temps de calculs sont raisonnables (moins de 2 heures). En revanche avec MAX/1500, on passe de 17 h de calcul quand $n = 4$, à 3 jours quand $n = 5$. Ainsi, tester des cas avec une valeur de n plus grande, avec des limites plus grandes (une surface maximale trop grande) ou avec plus de jeux de filtres prendrait trop temps.

Une solution pour palier cela serait de définir des heuristiques. Le temps de calcul serait plus raisonnable mais les solutions produites ne seraient plus optimales. Toutefois, on a vu qu'un certain schéma dans le choix des filtres semblait se profiler. Une voie de développement originale serait donc de créer une heuristique sur ce schéma. Ainsi, on augmenterait les chances d'avoir des solutions proches de l'optimal.

Après l'analyse de ces premiers résultats, nous avons continué les expérimentations pour pouvoir donner le front de Pareto. Nous avons donc calculé les solutions optimales pour MIN/10 à MIN/200 et pour MAX/100 à MAX/2000. Avec ces nouveaux résultats, nous disposons de plus de données permettant de mieux voir les fronts de Pareto pour chaque valeur de n. On constate que le cas monolithique est très vite restreint car les filtres que nous avons générés ne sont pas capables de rejeter plus de 90 dB de bruit. De plus, à taille ou réjection équivalente, la cascade de cinq filtres est la meilleure.

Toutefois, en analysant les résultats supplémentaires, nous nous sommes aperçus d'une anomalie. Lorsque la solution optimale ne place pas un filtre par étage, il arrive que les résultats soient moins bons que pour les étage précédents. Après une analyse de notre modèle, nous nous sommes rendus compte qu'une contrainte était fausse sans incidence toutefois sur les résultats car il est simple de rectifier l'erreur (nous détaillons cela dans la section 4.3.1 du chapitre 4). Nous l'avons corrigée et nous avons relancé les expéri-

mentations mais les temps de calculs se trouvaient multipliés par trois. Étant donné que l'erreur produite est simple à corriger, nous avons préféré garder la première version des programmes quadratiques afin d'avoir des temps de calculs plus rapides.

Satisfait de nos premiers résultats, nous avons essayé d'améliorer notre modèle. En effet, les premiers résultats montraient de fortes fluctuations (plus de 10 dB) dans la bande passante. Nous avons proposé de définir un critère de réjection plus strict lors de l'évaluation des jeux de coefficients. Pour cela, on a supprimé tous les filtres qui présentaient une fluctuations de plus de 1 dB dans la bande passante, passant ainsi d'un total de plus de 1800 filtres à 1300. Avec près de 30 % de filtres en moins, la recherche de solutions optimales fut grandement accélérée (le pire cas n'étant qu'à 18 minutes). Les autres conclusions sont restées valables : la cascade de filtres reste préférable en terme d'occupation de ressources et de réjection maximale et les solutions abstraites coïncident avec les résultats expérimentaux. Ainsi, on apporte une nouvelle confirmation de nos résultats et nous démontrons que le modèle peut être adapté pour répondre à d'autres contraintes.

Pour étendre notre modèle, on pourrait prendre en compte la décimation du flux. Avec cette étape supplémentaire, nous aurions un étage de filtrage classique : FIR, décimation et décalage. Le problème de la décimation est que cela induit du repliement spectral. Par conséquent, le design des filtres doit tenir compte de cet effet de repliement spectral, il faudra donc changer notre critère de réjection et aussi adapter la conception du modèle.

Sur le plus long terme, il serait intéressant d'appliquer notre méthodologie à de nouvelles chaînes de traitement. Ainsi nous pourrions vérifier que notre méthode fonctionne également pour d'autres situations et dans d'autres contextes.

Cinquième partie

Annexe

Annexe A

Bibliothèque de simulation des chaînes de traitement sur FPGA

Lors de nos expérimentations, nous nous sommes rapidement aperçus qu'il était long de débugger nos designs FPGA. En effet, le temps de trouver la solution optimale, de créer le design et de récolter les données prenait en général 30 minutes.

Nous avons donc créé une bibliothèque C++ pour nous permettre de simuler le comportement matériel de nos chaînes de traitement que nous avons appelée *Digital Signal Processing Simulation* (dsps). L'idée était d'appliquer un traitement en temps réel et donnée par donnée comme le ferait un FPGA. Nous voulions également laisser la possibilité de travailler soit en nombre à virgule flottante soit en entier signé.

La bibliothèque est basée sur l'idée des Skeletons : les tâches qu'on implémente sont génériques et paramétrables. Nos tirons parti du paradigme orienté objet, en définissant une classe `Task` que sera la classe mère de tous nos traitements.

Toutes les tâches possèdent un nombre fini d'entrées et/ou de sorties. Si une tâche n'a pas de sorties, c'est qu'il s'agit d'une tâche de fin qui va consommer des données (par exemple la classe `FileSink` qui se charge de stocker les données reçues dans un fichier). En revanche, si une tâche n'a pas d'entrées, il s'agit alors d'une tâche source qui va produire des données (par exemple la classe `FileSource` qui se charge de lire les données d'un fichier et de les propager dans la chaîne de traitement). Les tâches ont des méthodes pour contrôler leur état :

- `isReady` permet de savoir si une tâche a suffisamment de données pour faire son traitement (par exemple pour faire une FFT, il faut N données).
- `hasFinished` indique si la tâche a fini son traitement.
- `compute` permet de lancer le traitement de la tâche.

Les tâches sont reliées entre elles par des files de données qui garantissent que les données soient traitées dans l'ordre. C'est la classe `Channel` qui encapsule ces files. Toutefois pour nous permettre de rendre ces communications communes, les files ne transfèrent que des octets, ce sont les tâches qui doivent savoir quel type de données elles envoient ou récupèrent. À l'initialisation d'une tâche, nous renseignons le type des données reçues et

```
sources = Tâches sources
sinks = Tâches finales
tasks = Toutes les autres tâches

tant que Toutes les tâches finales n'ont pas terminé leur traitement faire
    pour chaque source dans sources faire
        si source.isReady() alors // Toujours vrai
            source.compute()
        fin
    fin

    pour chaque task dans tasks faire
        si task.isReady() alors
            task.compute()
        fin
    fin

    pour chaque sink dans sinks faire
        si sink.isReady() alors
            sink.compute()
        fin
    fin
fin
```

Algorithme 1 : Pseudo-code de la gestion d'une chaîne de traitement

celui des données envoyées. Le type d'entrée peut être différent de celui de sortie comme par exemple lors du calcul d'une FFT qui prend des données réelles et qui sort des données complexes.

Afin de nous assurer que les tâches connectées sont bien compatibles, nous avons défini la fonction `connect` qui permet de relier des tâches entre elles et qui s'assure que les types correspondent bien.

Grâce à ces différentes méthodes communes et aux files reliant les tâches, nous sommes capables de représenter une chaîne de traitement sans boucle d'action (une chaîne de traitement qu'on puisse représenter sous forme d'un DAG). Pour faire le traitement, il faut savoir quelles sont les tâches sources (qui vont générer les données) et quelles sont les tâches finales (qui vont consommer les données) et on applique le pseudo-code 1.

Dès lors que la boucle principale est finie, les données attendues sont prêtes. On ne risque pas d'avoir de boucle infinie car nous utilisons un DAG et il n'y a pas de cycles entre nos tâches.

Afin d'étoffer notre bibliothèque, nous avons récupéré un générateur de bruit blanc issu de la bibliothèque SigmaTheta [28]. Nous avons adapté le code de SigmaTheta pour que la génération de bruits blanc se fasse en continu et non plus par paquet de données.

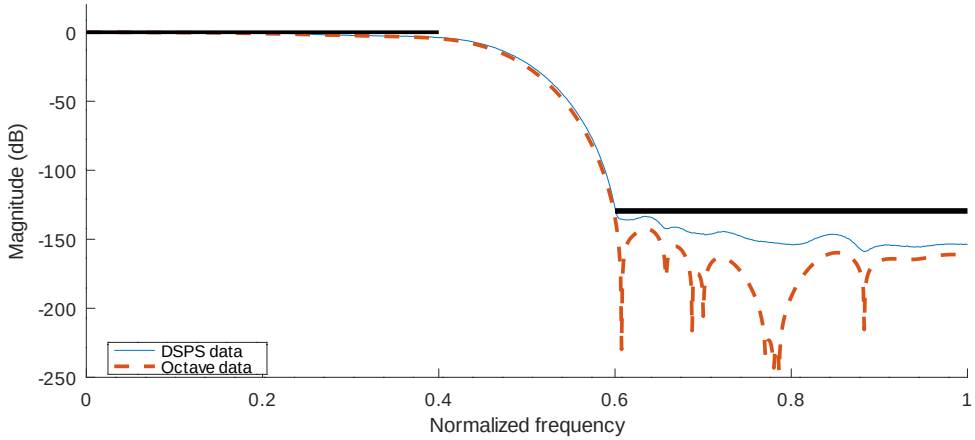


FIGURE A.1 – Simulation d'un cascade de filtres avec la bibliothèque dsps

Nous avons également intégré du code métier développé en interne. Ce code nous a permis d'avoir des fonctions de traitement et de post-traitement rapidement sans devoir tout redévelopper.

Dans le cadre de cette thèse nous avons principalement utilisé cette bibliothèque pour simuler le comportement d'une cascade de filtres. La figure A.1 montre une simulation obtenue pour une cascade de 4 filtres. En bleu la courbe simulée avec dsps, en pointillé rouge, la courbe idéale obtenue avec Octave et en noir le gabarit du filtre attendu. Il est par ailleurs possible d'utiliser la bibliothèque dsps pour traiter des chaînes plus complexes. La figure A.2 présente le résultat de la simulation d'un banc de bruit de phases où on a calculé trois décades.

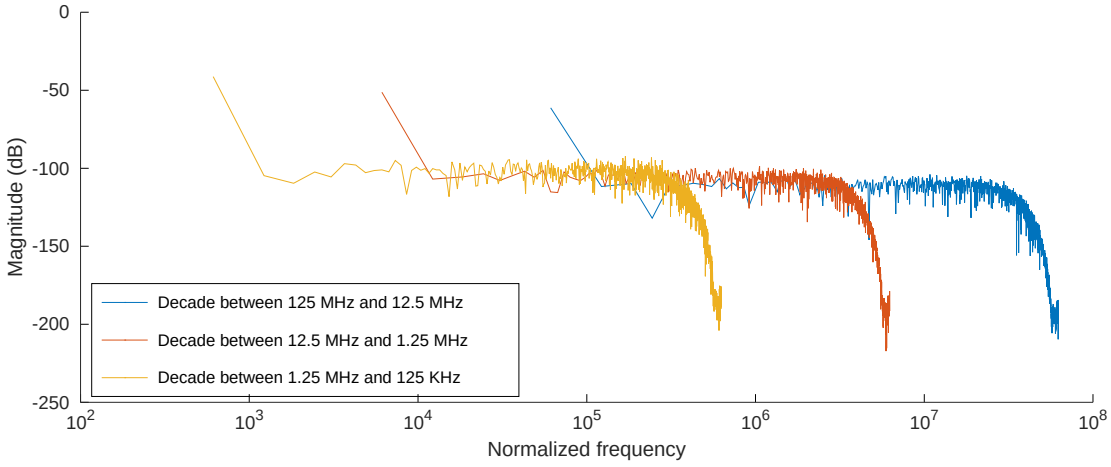


FIGURE A.2 – Banc de bruits de phase simulé avec dsps

L'intégralité de cette bibliothèque est disponible sur le dépôt public : `https://`

github.com/oscimp/libdsps.

Contributions scientifiques

Journal international

1. A. Hugeat, J. Bernard, G. Goavec-Merou, P.-Y. Bourgeois et J.-M. Friedt. Filter optimization for real time digital processing of radiofrequency signals : application to oscillator metrology, *IEEE Trans. Ultrasonics Ferroelectrics and Frequency Control*, 2019

Conférences

1. A. Hugeat, J. Bernard, G. Goavec-Merou, P.-Y. Bourgeois et J.-M. Friedt. Optimization of Signal Processing Chains : Application to Cascaded Filters *In 27th European Signal Processing Conference (EUSIPCO)*, septembre 2019.
2. B. Maréchal, A. Hugeat, G. Goavec-Merou, G. Cabodevila, J. Millo, C. Lacroûte, et P.-Y. Bourgeois. Digital Implementation of Various Locking Schemes of Ultrastable Photonics Systems, *In IEEE International Frequency Control Symposium (IFCS)*, mai 2018.
3. A. Hugeat, J. Bernard, G. Goavec-Merou, P.-Y. Bourgeois et J.-M. Friedt. Filter Optimization for Real Time Digital Processing of Radiofrequency Signals : Application to Oscillator Metrology. *In IEEE International Frequency Control Symposium (IFCS)*, mai 2018.
4. J.-M. Friedt, A. Hugeat, S. Lamare, F. Chérioux, D. Rabus et L. Arapan Subsurface wireless chemical sensing strategy compatible with Ground Penetrating RADAR *In 9th International Workshop on Advanced Ground Penetrating Radar (IWAGPR)*, 2017.

Bibliographie

[1] SpinalHDL. Available online : `https://spinalhdl.github.io/SpinalDoc-RTD/`. Accessed : 2019-08-22.

[2] U. Ali, M. B. Malik, and K. Munawar. FPGA/soft-processor based real-time object tracking system. In *2009 5th Southern Conference on Programmable Logic (SPL)*, pages 33–37, April 2009.

[3] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel : Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, June 2012.

[5] D. G. Bariamis, D. K. Iakovidis, D. E. Maroulis, and S. A. Karkanis. An FPGA-based architecture for real time image feature extraction. In *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, volume 1, pages 801–804 Vol.1, Aug 2004.

[6] J.E. Beasley. *Advances in linear and integer programming*. Oxford University Press, Inc., 1996.

[7] R. Bellman. Dynamic programming. *Science*, 153(3731) :34–37, 1966.

[8] K. Benkrid, S. Belkacemi, and A. Benkrid. HIDE : A hardware intelligent description environment. *Microprocessors and Microsystems*, 30(6) :283 – 300, 2006. Special Issue on FPGA's.

[9] K. Benkrid, A. Benkrid, and S. Belkacemi. Efficient FPGA hardware development : A multi-language approach. *Journal of Systems Architecture*, 53(4) :184 – 209, 2007.

[10] K. Benkrid, D. Crookes, J. Smith, and A. Benkrid. High level programming for FPGA based image and video processing using hardware skeletons. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pages 219–226, March 2001.

[11] R. G. Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2(2) :103–107, 1977.

[12] D. Burger. Microsoft unveils project brainwave for real-time AI. *Microsoft Research, Microsoft*, 22, 2017.

[13] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J.H. Anderson, S. Brown, and T. Czajkowski. LegUp : High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM.

[14] A.C. Cárdenas-Olaya, E. Rubiola, J.-M. Friedt, P.-Y. Bourgeois, M. Ortolano, S. Micalizio, and C.E. Calosso. Noise characterization of analog to digital converters for amplitude and phase noise measurements. *Review of Scientific Instruments*, 88(6) :065108, 2017.

[15] R. Cemes and D. Ait-Boudaoud. Genetic approach to design of multiplierless FIR filters. *Electronics Letters*, 29(24) :2090–2091, Nov 1993.

[16] A. Chandra and S. Chattopadhyay. Design of hardware efficient FIR filter : A review of the state-of-the-art approaches. *Engineering Science and Technology, an International Journal*, 19(1) :212 – 226, 2016.

[17] C. Chen, A. Srivastav, D. Ariando, S. Mandal, Y. Tang, and Y. Song. Real-time data inversion methods for low-field nuclear magnetic resonance (NMR). In *2018 IEEE 23rd International Conference on Digital Signal Processing (DSP)*, pages 1–5, Nov 2018.

[18] D. Chiou. The microsoft catapult project. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 124–124, Oct 2017.

[19] M.I. Cole. *Algorithmic skeletons : structured management of parallel computation*. Pitman London, 1989.

[20] A. Colorni, M. Dorigo, V. Maniezzo, et al. Distributed optimization by ant colonies. In *Proceedings of the first European conference on artificial life*, volume 142, pages 134–142. Cambridge, MA, 1992.

[21] R.W Conway, W.L. Maxwell, and L.W. Miller. *Theory of scheduling*. Courier Corporation, 2003.

[22] M. Coqblin. Optimisation du débit pour des applications linéaires multi-tâches sur plateformes distribuées incluant des temps de reconfiguration, 2012.

[23] S. Coric, M. Leeser, E. Miller, and M. Trepanier. Parallel-beam backprojection : An FPGA implementation optimized for medical imaging. In *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-programmable Gate Arrays*, FPGA '02, pages 217–226, New York, NY, USA, 2002. ACM.

[24] A. C. Cárdenas Olaya, C. E. Calosso, J. Friedt, S. Micalizio, and E. Rubiola. Phase noise and frequency stability of the red-pitaya internal pll. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 66(2) :412–416, Feb 2019.

[25] D. Li, Y. C. Lim, Y. Lian, and J. Song. A polynomial-time algorithm for designing FIR filters with power-of-two coefficients. *IEEE Transactions on Signal Processing*, 50(8) :1935–1941, Aug 2002.

[26] H. H. Dam, A. Cantoni, K. L. Teo, and S. Nordholm. Fir variable digital filter with signed power-of-two coefficients. *IEEE Transactions on Circuits and Systems I : Regular Papers*, 54(6) :1348–1357, June 2007.

[27] A. G. Dempster and M. D. Macleod. Use of minimum-adder multiplier blocks in FIR digital filters. *IEEE Transactions on Circuits and Systems II : Analog and Digital Signal Processing*, 42(9) :569–577, Sep. 1995.

[28] Observatoire des Sciences de l'Univers Terre Homme Environment Temps Astronomie de Bourgogne Franche-Comté. SigmaTheta. `https://theta.obs-besancon.fr/spip.php?article103`. Accessed : 2019-08-23.

[29] F. Dittmann. Algorithmic skeletons for the programming of reconfigurable systems. In R. Obermaisser, Y. Nah, P. Puschner, and F.J. Rammig, editors, *Software Technologies for Embedded and Ubiquitous Systems*, pages 358–367, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[30] A. Dobrila. Optimisation du débit en environnement distribué incertain, 2011.

[31] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system : optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1) :29–41, Feb 1996.

[32] ACE Associated Compiler Experts. Cosy. `http://www.ace.nl`. Accessed : 2019-08-21.

[33] Time FEMTO-ST and Frequency department. Oscimp digital. `https://github.com/oscimp/oscimpDigital`. Accessed : 2019-08-23.

[34] Z. G. Feng and K. L. Teo. A discrete filled function method for the design of FIR filters with signed-powers-of-two coefficients. *IEEE Transactions on Signal Processing*, 56(1) :134–139, Jan 2008.

[35] GNU Radio Foundation. GNU radio. `https://www.gnuradio.org/`. Accessed : 2019-08-23.

[36] M. Genovese and E. Napoli. FPGA-based architecture for real time segmentation and denoising of hd video. *Journal of Real-Time Image Processing*, 8(4) :389–401, Dec 2013.

[37] P. Gentili, F. Piazza, and A. Uncini. Efficient genetic algorithm design for power-of-two FIR filters. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 1268–1271 vol.2, May 1995.

[38] F. Glover. Tabu search - Part I. *ORSA Journal on Computing*, 1(3) :190–206, 1989.

[39] F. Glover. Tabu search - Part II. *ORSA Journal on Computing*, 2(1) :4–32, 1990.

[40] G. Goavec-Merou. Générateur de coprocesseur pour le traitement de données en flux (vidéo ou similaire) sur FPGA, 2014.

[41] D.E. Goldberg and J.H. Holland. Genetic algorithms and machine learning. *Machine Learning*, 3(2) :95–99, Oct 1988.

[42] O. Gustafsson. A difference based adder graph heuristic for multiple constant multiplication problems. In *2007 IEEE International Symposium on Circuits and Systems*, pages 1097–1100, May 2007.

[43] O. Gustafsson, H. Johansson, and L. Wanhammar. *An MILP approach for the design of linear-phase FIR filters with minimum number of signed-power-of-two terms*. Citeseer, 2001.

[44] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W.J. Dally. ESE : Efficient speech recognition engine with sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 75–84, New York, NY, USA, 2017. ACM.

[45] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2) :100–107, July 1968.

[46] G. Hoeksma. Schéma architecture superhétérodyne. `https://fr.wikipedia.org/wiki/R%C3%A9cepteur_superh%C3%A9t%C3%A9rodyne#/media/Fichier:Superhet2_French_Version.png`. Accessed : 2019-08-21.

[47] J. Hoozemans, R. Heij, J. van Straten, and Z. Al-Ars. VLIW-based FPGA computation fabric with streaming memory hierarchy for medical imaging applications. In S. Wong, A.C. Beck, K. Bertels, and L. Carro, editors, *Applied Reconfigurable Computing*, pages 36–43, Cham, 2017. Springer International Publishing.

[48] A. Ibrahim, W. Simon, D. Doy, E. Pignat, F. Angiolini, M. Arditi, J. . Thiran, and G. De Micheli. Single-FPGA complete 3d and 2d medical ultrasound imager. In *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 1–6, Sep. 2017.

[49] R. Ito and R. Hirabayashi. Optimal design of FIR filter with SP2 coefficients based on semi-infinite linear programming method. In *2006 14th European Signal Processing Conference*, pages 1–5, Sep. 2006.

[50] R. Ito, K. Suyama, and R. Hirabayashi. Optimal design method for FIR filter with discrete coefficients based on integer semi-infinite linear programs. In *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221)*, volume 6, pages 3805–3808 vol.6, May 2001.

[51] J. Chen, S. Zhou, and H. Min. Implementation of parallel medical ultrasound imaging algorithm on CAPI-enabled FPGA. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 311–314, Dec 2016.

[52] D. I. Kaplun, V. V. Gulvanskiy, D. M. Klionskiy, M. S. Kupriyanov, and A. V. Veligosha. Implementation of digital filters in the residue number system. In *2016 IEEE NW Russia Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW)*, pages 220–224, Feb 2016.

[53] D. Karaboga. An idea based on honey bee swarm for numerical optimization, technical report - tr06. Technical report, 01 2005.

[54] S. Kawahito, M. Kameyama, and T. Higuchi. Multiple-valued radix-2 signed-digit arithmetic circuits for high-performance vlsi systems. *IEEE Journal of Solid-State Circuits*, 25(1) :125–131, Feb 1990.

[55] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, Nov 1995.

[56] T. Kryjak, M. Komorkiewicz, and M. Gorgon. Real-time background generation and foreground object segmentation for high-definition colour video stream in FPGA device. *Journal of Real-Time Image Processing*, 9(1) :61–77, Mar 2014.

[57] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3) :497–520, 1960.

[58] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, and W. Sung. FPGA-based low-power speech recognition with recurrent neural networks. In *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 230–235, Oct 2016.

[59] Y. C. Lim and B. Liu. Design of cascade form FIR filters with discrete valued coefficients. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 36(11) :1735–1739, Nov 1988.

[60] Y.-C. Lim, R. Yang, and B. Liu. The design of cascaded FIR filters. In *1996 IEEE International Symposium on Circuits and Systems. Circuits and Systems Connecting the World. ISCAS 96*, volume 2, pages 181–184 vol.2, May 1996.

[61] A. Lindahl and L. Bengtsson. A low-power FIR filter using combined residue and radix-2 signed-digit representation. In *8th Euromicro Conference on Digital System Design (DSD'05)*, pages 42–47, Aug 2005.

[62] W. Lu. Design of FIR filters with discrete coefficients : a semidefinite programming relaxation approach. In *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196)*, volume 2, pages 297–300 vol. 2, May 2001.

[63] V. J. Manoj and E. Elias. On the design of multiplier-less nonuniform filterbank transmultiplexer using particle swarm optimization. In *2009 World Congress on Nature Biologically Inspired Computing (NaBIC)*, pages 55–60, Dec 2009.

[64] V.J. Manoj and Elizabeth Elias. Artificial bee colony algorithm for the design of multiplier-less nonuniform filter bank transmultiplexer. *Information Sciences*, 192 :193 – 203, 2012. Swarm Intelligence and Its Applications.

[65] M. Manuel and E. Elias. Design of multiplier-less FRM FIR filter using artificial bee colony algorithm. In *2011 20th European Conference on Circuit Theory and Design (ECCTD)*, pages 322–325, Aug 2011.

[66] S. J. Melnikoff, S. F. Quigley, and M. J. Russell. Implementing a simple continuous speech recognition system on an FPGA. In *Proceedings. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 275–276, April 2002.

[67] K.G. Murty. *Linear programming*. Springer, 1983.

[68] R. Nane, V. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels. DWARV 2.0 : A CoSy-based C-to-VHDL hardware compiler. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 619–622, Aug 2012.

[69] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10) :1591–1604, Oct 2016.

[70] A. Nannarelli, M. Re, and G. C. Cardarilli. Tradeoffs between residue number system and traditional FIR filters. In *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196)*, volume 2, pages 305–308 vol. 2, May 2001.

[71] N.I. Cho and S.U. Lee. Optimal design of finite precision FIR filters using linear programming with reduced constraints. *IEEE Transactions on Signal Processing*, 46(1) :195–199, Jan 1998.

[72] M. Odersky. Scala programming language. Available online : `https://www.scala-lang.org/`. Accessed : 2019-08-22.

[73] A. V. Oppenheim. *Discrete-time signal processing*. Pearson Education India, 1999.

[74] C.H. Papadimitriou and K. Steiglitz. *Combinatorial optimization*, volume 24. Prentice Hall Englewood Cliffs, 1982.

[75] D.A. Parker and K.K. Parhi. Low-area/power parallel FIR digital filter implementations. *Journal of VLSI signal processing systems for signal, image and video technology*, 17(1) :75–92, Sep 1997.

[76] C. Pilato and F. Ferrandi. Bambu : A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4, Sep. 2013.

[77] C. Pilato, F. Ferrandi, and D. Sciuto. A design methodology to implement memory accesses in high-level synthesis. In *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 49–58, Oct 2011.

[78] M. Pinedo. *Scheduling*, volume 29. Springer, 2012.

[79] X. Qin, Y. Xie, R. Li, X. Rong, X. Kong, F. Shi, P. Wang, and J. Du. High-time-resolution nuclear magnetic resonance with nitrogen-vacancy centers. *IEEE Magnetics Letters*, 7 :1–5, 2016.

[80] L. Rabiner. Linear program design of finite impulse response (FIR) digital filters. *IEEE Transactions on Audio and Electroacoustics*, 20(4) :280–288, October 1972.

[81] E. Rubiola. *Phase noise and frequency stability in oscillators*. The Cambridge RF and microwave engineering series. Cambridge University Press, Cambridge, 2009.

[82] B. Schrauwen, M. D'Haene, D. Verstraeten, and J. V. Campenhout. Compact hardware liquid state machines on fpga for real-time speech recognition. *Neural Networks*, 21(2) :511 – 523, 2008. Advances in Neural Networks Research : IJCNN '07.

[83] J. Schuster, K. Gupta, R. Hoare, and A.K. Jones. Speech silicon : An FPGA architecture for real-time hidden markov-model-based speech recognition. *EURASIP Journal on Embedded Systems*, 2006(1) :048085, Nov 2006.

[84] J.A. Sherman and R. Jördens. Oscillator metrology with software defined radio. *Review of Scientific Instruments*, 87(5) :054711, 2016.

[85] M. A. Soderstrand and K. Al-Marayati. VLSI implementation of very-high-order FIR filters. In *Proceedings of ISCAS'95 - International Symposium on Circuits and Systems*, volume 2, pages 1436–1439 vol.2, April 1995.

[86] M.A. Soderstrand, W.K. Jenkins, G.A. Jullien, and F.J. Taylor. *Residue number system arithmetic : modern applications in digital signal processing*. IEEE press, 1986.

[87] G. A. Stimpson, M. S. Skilbeck, R. L. Patel, B. L. Green, and G. W. Morley. An open-source high-frequency lock-in amplifier. *Review of Scientific Instruments*, 90(9) :094701, 2019.

[88] K. Takeda. A highly integrated FPGA-based nuclear magnetic resonance spectrometer. *Review of scientific instruments*, 78(3) :033103, 2007.

[89] S. Traferro, F. Capparelli, F. Piazza, and A. Uncini. Efficient allocation of power of two terms in FIR digital filter design using tabu search. In *ISCAS'99. Proceedings of the 1999 IEEE International Symposium on Circuits and Systems VLSI (Cat. No.99CH36349)*, volume 3, pages 411–414 vol.3, May 1999.

[90] Y. Tsao and K. Choi. Area-efficient parallel FIR digital filter structures for symmetric convolutions based on fast FIR algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(2) :366–371, Feb 2012.

[91] P.P. Vaidyanathan. Chapter 2 - design and implementation of digital FIR filters. In Douglas F. Elliott, editor, *Handbook of Digital Signal Processing*, pages 55 – 172. Academic Press, San Diego, 1987.

[92] F. L. Vargas, R. D. R. Fagundes, and D. B. Junior. A FPGA-based viterbi algorithm implementation for speech recognition systems. In *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221)*, volume 2, pages 1217–1220 vol.2, May 2001.

[93] Y. Voronenko and M. Püschel. Multiplierless multiple constant multiplication. *ACM Trans. Algorithms*, 3(2), May 2007.

[94] G. Wade, A. Roberts, and G. Williams. Multiplier-less FIR filter design using a genetic algorithm. *IEE Proceedings - Vision, Image and Signal Processing*, 141 :175–180(5), June 1994.

[95] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi. The risc-v instruction set manual. volume 1 : User-level isa, version 2.0. Technical report, Univ of Berkeley, California, Dept. of Electrical Engineering and Computer Sciences, 2014.

[96] C. Young and D. L. Jones. Improvement in finite wordlength FIR digital filter design by cascading. In *[Proceedings] ICASSP-92 : 1992 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 109–112 vol.5, March 1992.

[97] X. Zhang, X. Liu, A. Ramachandran, C. Zhuge, S. Tang, P. Ouyang, Z. Cheng, K. Rupnow, and D. Chen. High-performance video content recognition with long-term recurrent convolutional network for FPGA. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sep. 2017.

[98] X. Zhang, B. Zhou, H. Li, X. Zhao, W. Mu, and W. Wu. The design of photoelectric signal processing system for a nuclear magnetic resonance gyroscope based on fpga. volume 10464, 2017.

Table des figures

Liste des tableaux

SPIM
école doctorale **sciences pour l'ingénieur et microtechniques**

Titre : Optimisation de chaînes de traitement de signaux numériques radiofréquences et application à une cascade de filtres

Mots-clés : optimisation, algorithmique du bruit de quantification, métrologie temps-fréquence numérique, programmation linéaire en nombres entiers

Résumé :

La thèse s'articule autour de deux axes : d'une part la caractérisation, d'un point de vue du traitement numérique du signal, de la résolution et des ressources occupées par les algorithmes pertinents pour l'analyse de signaux issus d'oscillateurs stables (phase, fréquence). Nous ne nous intéressons pas ici à l'implémentation proprement dite de ces algorithmes mais dans une description de leurs entrées, sorties, ressources, latences et acceptances. Ayant identifié ces caractéristiques, elles sont exploitées dans un contexte de recherche d'ordonnancement optimal pour atteindre les meilleures performances possibles (en terme de résolution et de bande passante) du traitement de signaux issus d'oscillateurs sur une plate-forme matérielle donnée. La question de la limitation de la résolution numérique de traitement est abordée en regard du bruit intrinsèque des composants (convertisseurs analogique-numériques, horloges, bruits des références de tension, jitter sur l'échantillonnage), problème qui est actuellement considéré dans une thèse engagée sur l'analyse des composants numériques dans le contexte du temps-fréquence. Ainsi, le travail de thèse considéré ici est clairement orienté sur les aspects informatiques du traitement de signaux radiofréquences, mais en abordant des problématiques classiques de résolution de calculs et d'ordonnancement dans le contexte spécifique de la métrologie du temps-fréquence (corrélations, transformée de Fourier aux diverses échelles visant plus de 6-ordres de grandeur entre la gamme de fréquence la plus élevée et la plus lente – 10 MHz à moins de 1 Hz de la porteuse – filtres anti-repliement, extraction de phase à partir des signaux échantillonnés, oscillateur local numérique). Pour mettre en contexte la complexité de la tâche, rappelons qu'un niveau de bruit de -160 dBrad²/Hz se traduit par 16 décimales pertinentes dans le résultat du calcul.

Title: Optimization of Radiofrequency Digital Signal Processing Chains and Application to a Cascade of Filters

Keywords: optimization, quantization noise algorithmics, digital time & frequency metrology, mixed-integer linear programming

Abstract:

The thesis tackles along two axis the challenge of optimizing digital implementations of radiofrequency filters and signal processing chains for stable oscillator noise characterization: on one hand, from the digital signal processing perspective, the resolution and resources occupied by efficient algorithms to analyze signals from ultra stable oscillators (phase, frequency) are characterized. We are not interested here in the implementation of these algorithms but analyze the different characteristics such as input, output, resource, latency... Then, on the other hand, having identified these characteristics, they are exploited in the context of an optimal scheduling search to obtain the best performances (in terms of resolution and bandwidth) of a given hardware platform designed to collect data from ultra stable oscilators. The question of the limitation induced by the numerical resolution of the processing steps is approached with regard to the intrinsic noise of the components (analog-to-digital converters, clocks, voltage reference noise, jitter on sampling). This question has already been dealt with as part of another thesis defended at the Time-Frequency department. Thus, the thesis work considered here is clearly focused on the IT aspects of radiofrequency signal processing but by addressing classical problem solving and scheduling problems in the specific context of time-frequency metrology (correlations, Fourier transform at various scales aiming at more than 6-orders of magnitude between the highest and the slowest frequency range - 10 MHz to less than 1 Hz from the carrier - anti-aliasing filters, phase extraction from sampled signals, digital local oscillator). To put into context the complexity of the task, let us recall that a noise level of -160 dBrad²/Hz results relevant to 16 decimal places in the calculation result.

UBFC UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ
Université Bourgogne Franche-Comté
32, avenue de l'Observatoire
25000 Besançon, France