HAL
open science

Implémentation optimale de filtres linéaires en arithmétique virgule fixe

Benoit Lopez

To cite this version:

Benoit Lopez. Implémentation optimale de filtres linéaires en arithmétique virgule fixe. Systèmes embarqués. Université Pierre et Marie Curie - Paris VI, 2014. Français. NNT : 2014PA066357 . tel-01127376

HAL Id: tel-01127376

https://theses.hal.science/tel-01127376v1

Submitted on 7 Mar 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT de
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité
Informatique

École doctorale Informatique, Télécommunications et Électronique
(Paris)

Présentée par

Benoit LOPEZ

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Implémentation optimale de filtre linéaire en arithmétique virgule fixe

Soutenue le 27 Novembre 2014
devant le jury composé de :

Président du jury	F. de Dinechin	Professeur, INSA Lyon (CITI)
Examinateur	C. Jego	Professeur, Institut Polytechnique de Bordeaux (IMS)
Examinateur	S. Graillat	Professeur, Université Pierre et Marie Curie (LIP6)
Rapporteur	D. Ménard	Professeur, INSA Rennes (IETR)
Rapporteur	M. Martel	Maître de conférences (HDR), Université de Perpignan Via Domitia (LIRMM)
Directeur de thèse	L.-S. Didier	Professeur, Université de Toulon (IMATH)
Co-encadrant de thèse	T. Hilaire	Maître de conférences, Université Pierre et Marie Curie (LIP6)

À mon fils.

Remerciements

Pour commencer, je souhaite remercier Daniel Ménard, professeur à l'INSA Rennes, et Matthieu Martel, maître de conférences à l'Université de Perpignan Via Domitia, pour avoir accepté de rapporter ma thèse. Leurs retours m'ont permis de déterminer les prochaines pistes à explorer dans mes recherches suite au travail déjà réalisé.

Je tiens également à remercier Florent de Dinechin, professeur à l'INSA Lyon, d'avoir accepté de présider mon jury de thèse. Les discussions scientifiques avec Florent sont toujours très enrichissantes et j'espère qu'elles seront encore nombreuses. Je remercie aussi Christophe Jego, Professeur à l'Institut Polytechnique de Bordeaux, et Stef Graillat, Professeur à l'Université Pierre et Marie Curie et responsable de l'équipe PEQUAN au sein de laquelle j'ai effectué ma thèse, pour avoir fait partie de mon jury.

Je voudrai remercier mes directeur et co-encadrant de thèse, dans l'ordre Laurent-Stéphane et Thibault, tout d'abord pour m'avoir fait confiance en me choisissant pour réaliser cette thèse. Même si la carrière de Laurent-Stéphane l'a amené a quitté le LIP6, ses conseils pour ma thèse et mon avenir ont toujours été avisés et pertinents.

Un grand merci à Thibault pour m'avoir encadré pendant trois ans. J'ai énormément appris, tant sur le fond dans un domaine que je ne maîtrisais pas du tout, que sur la forme où ses conseils en matière de rédaction ou de présentation ont toujours été utiles. Nous nous sommes rapidement bien entendus avec Thibault et je pense que c'est grâce à cela que notre collaboration s'est si bien déroulée, et je ne doute pas qu'elle continuera encore ainsi pour de nombreux résultats scientifiques.

Je remercie également l'équipe PEQUAN pour la bonne atmosphère qu'il y règne, tant professionnellement dans les discussions scientifiques que dans le relâchement des repas au restaurant administratif.

Je souhaite surtout remercier mes co-bureaux, anciens ou actuels. Travailler avec des gens dans la même galère dans une si bonne ambiance permet d'aborder les périodes de stress plus sereinement et de tenir le coup, mais aussi de se détendre dans les périodes plus tranquilles. Plus particulièrement, merci

à Anastasia pour son amitié au quotidien et son soutien notamment dans les dernières semaines avant ma soutenance, à Julien pour répondre présent quand j'en ai besoin, sans que je n'ai forcément à le demander, et à Christophe pour sa cool-attitude et pour les grands moments de fun dans le bureau 301 avec Julien (même si jouer au "ballon" la fenêtre ouverte n'était pas notre idée la plus brillante). Sans oublier Jean-Claude qui, malgré son emploi du temps chargé, avait très régulièrement la simplicité et la disponibilité d'accompagner notre fine équipe de doctorants pour quelques verres après le travail.

Pour conclure ces remerciements et passer à la science, je tiens à dire un immense merci à ma famille. Tout d'abord mes parents sans qui je n'en serai certainement pas arrivé là et plus particulièrement ma mère, mon premier et plus indéfectible soutien depuis toujours. J'ai évidemment une pensée spéciale pour ma sœur et mon frère que j'adore. Le plus grand des mercis revient à Hélène, ma compagne et mon plus grand soutien pendant ces trois années, qui m'a supporté dans les moments de stress comme dans les moments de fatigues. Dans le même ordre d'idée, je tiens à remercier mon fils qui, s'il n'en a pas conscience, m'a énormément aidé, que ce soit en faisant ses nuits rapidement, en étant toujours des plus adorables mais aussi et surtout en me motivant encore plus pour le rendre fier de moi. Je remercie également Hélène pour s'être formidablement occupé de Gauthier lors des périodes de grandes rédactions, quand j'étais peu disponible, sans jamais se plaindre.

Ah oui et comment oublier, j'en reviens à Christophe qui nous a initié au jeu Curve Fever (anciennement Achtung die Kurve). Je ne sais pas si je dois le remercier pour ça mais je suis vite devenu un grand joueur de ce jeu en ligne (grand par le nombre de parties jouées, pas par le niveau...) et il m'a permis de me défouler durant ma thèse. J'ai donc une pensée pour mes camarades de jeu, dont certains ont déjà été nommés dans ces remerciements : Krissssss, Grogdush, Toto_le_clown, Alanor, Fü Bär, Testuo, etc.

Table des matières

Table des matières v

Table des figures x

Liste des tableaux xiii

Introduction 1

1 Filtres linéaires 9
- 1.1 Signal discret et filtre linéaire . . . 9
  - 1.1.1 Rappels sur les signaux . . . 9
  - 1.1.2 Filtres linéaires invariants dans le temps . . . 11
    - 1.1.2.1 Réponse impulsionnelle . . . 11
    - 1.1.2.2 Filtres FIR et IIR . . . 12
    - 1.1.2.3 Fonction de transfert . . . 13
    - 1.1.2.4 Réalisations . . . 14
    - 1.1.2.5 Normes de filtres . . . 14
- 1.2 Représentation d'état . . . 16
  - 1.2.1 Le calcul de normes par la représentation d'état . . . 17
  - 1.2.2 Calcul de sensibilité . . . 18
- 1.3 Différentes réalisations . . . 19
  - 1.3.1 Formes Directes . . . 21
    - 1.3.1.1 Forme Directe I . . . 21
    - 1.3.1.2 Forme Directe II . . . 21
    - 1.3.1.3 Formes Directes Transposées . . . 22
    - 1.3.1.4 ρDFIIt . . . 23
  - 1.3.2 Structure LGS . . . 24
  - 1.3.3 Structure LCW . . . 25
  - 1.3.4 Décompositions en sous-filtres . . . 25
    - 1.3.4.1 Structure en cascade . . . 25

1.3.4.2 Structure parallèle 26
1.4 Forme implicite 27
1.4.1 Exemple : Structures LGS et LCW 29
1.5 Conclusion 33

2 Arithmétique virgule fixe 35
2.1 Représentations binaires des nombres 35
2.1.1 Nombres entiers 36
2.1.1.1 Représentation Signe - Valeur Absolue 36
2.1.1.2 Représentation en complément à un 37
2.1.1.3 Représentation en complément à deux 38
2.1.1.4 Comparaison des représentations pour les nombres entiers 39
2.1.2 Nombres réels 39
2.1.2.1 Virgule Flottante (Norme IEEE754) 40
2.1.2.2 Virgule Fixe 42
2.1.2.3 Comparaison entre virgule flottante et virgule fixe 43
2.2 Conversion en virgule fixe 45
2.2.1 Conversion à largeur connue 45
2.2.1.1 Format virgule fixe inconnu 45
2.2.1.2 Format virgule fixe connu 48
2.2.2 Conversion à largeur inconnnue 49
2.2.3 Conversion à partir d'un intervalle 50
2.3 Opérations virgule fixe 51
2.3.1 Formatage 51
2.3.1.1 Décalage sur un entier 52
2.3.1.2 Formatage sur une cible matérielle 52
2.3.1.3 Formatage sur une cible logicielle 53
2.3.1.4 Débordement 54
2.3.2 Addition virgule fixe 56
2.3.2.1 Choix du format 57
2.3.2.2 Mise au format commun des opérandes 60
2.3.2.3 Calcul de la somme 61
2.3.3 Multiplication virgule fixe 63
2.3.3.1 Cas matériel 64
2.3.3.2 Cas logiciel 67
2.4 Quantification 67
2.4.1 Lois de quantification 68
2.4.1.1 Troncature (ou arrondi vers $-\infty$) 68
2.4.1.2 Arrondi au plus proche 70
2.4.1.3 Arrondi vers $+\infty$ et arrondi fidèle 71
2.4.2 Calcul de l'erreur 72
2.4.2.1 Erreurs absolue et relative 74

2.4.2.2 Bruit de quantification 74
2.4.2.3 Intervalles d'erreur 75
2.4.2.4 Produit et quantification 76
2.5 Conclusion 77

3 Vers l'implémentation de filtres linéaires en virgule fixe 79
3.1 Évolution d'un signal à travers un filtre 79
3.2 Analyse de l'erreur en sortie d'un filtre 82
3.2.1 Propagation des erreurs de calcul 82
3.2.2 Propagation des erreurs de calcul et des erreurs paramétriques 85
3.3 Des filtres aux produits scalaires 88
3.3.1 Relation entre filtres et produits scalaires 89
3.3.2 Spécificités des produits scalaires de filtres 89
3.4 Exemple de filtre 91
3.4.1 Forme Directe I 91
3.4.1.1 Analyse d'erreur 92
3.4.1.2 SIF 93
3.4.2 State-Space 94
3.4.2.1 Analyse d'erreur 94
3.5 Conclusion 95

4 Implémentation logicielle de filtres et produits scalaires 97
4.1 Produits scalaires ordonnés 98
4.2 Propagation 106
4.3 Évaluation de l'erreur et choix d'un oSoP 109
4.3.1 Bruit de quantification 109
4.3.2 Intervalle d'erreur 110
4.3.3 Autre critère de choix, le parallélisme 111
4.3.4 Choix d'un oSoP 113
4.4 Génération de code virgule fixe pour un oSoP 113
4.4.1 Algorithme virgule fixe 114
4.4.2 Code `C` entier 116
4.4.3 Code `C++` virgule fixe 117
4.4.4 Code `VHDL` 120
4.5 Résumé de la méthode et illustration sur un filtre réalisé par un state-space 120
4.5.1 Exemple de traitement d'un filtre réalisé par plusieurs produits scalaires 121
4.6 Conclusion 125

5 Optimisation des largeurs 127
5.1 État de l'art autour de l'optimisation des largeurs en virgule fixe 128

5.1.1 Quelques notions d'optimisation combinatoire et principaux algorithmes . . . 128
5.1.2 Optimisations des largeurs . . . 131
5.2 Première approche de réduction des bits des largeurs : le formatage de bits . . . 134
5.2.1 Problématique et notations . . . 135
5.2.2 Calcul de δ et résumé de la méthode . . . 136
5.2.3 Exemple . . . 138
5.3 Formalisation du problème d'optimisation . . . 139
5.3.1 Formalisation des contraintes de formatage de bits . . . 141
5.3.2 Formalisation des contraintes d'erreurs . . . 142
5.3.2.1 Mode d'arrondi par troncature . . . 142
5.3.2.2 Mode d'arrondi au plus proche . . . 144
5.3.3 Discussion sur les différentes contraintes . . . 144
5.4 Résolution du problème . . . 145
5.4.1 Cas particulier . . . 146
5.4.2 Cas général . . . 147
5.5 Exemples . . . 148
5.5.1 Forme Directe I . . . 149
5.5.2 State-Space . . . 150
5.6 Conclusion . . . 153

6 Exemple complet du flot `FiPoGen` 155
6.1 Méthodologie `FiPoGen` . . . 155
6.1.1 Solution logicielle . . . 156
6.1.2 Solution matérielle . . . 156
6.2 Description de l'exemple . . . 157
6.3 Implémentation logicielle . . . 162
6.4 Implémentation matérielle . . . 166
6.5 Conclusion . . . 169

Conclusion 171

A Structures LGS et LCW 175
A.1 Paramétrisation JSS . . . 175
A.2 Structure LGS . . . 177
A.3 Structure LCW . . . 179

B Codes 183
B.1 Code de l'exemple 4.5.1 . . . 183

C Preuves 185
C.1 Preuve de la proposition 3.1 (page 79) . . . 185
C.2 Preuve de la proposition 3.2 (page 80) . . . 187

C.3 Preuve de la proposition 4.2 (page 103) 188
C.4 Preuve de la proposition 5.1 (page 136) 191
C.4.1 Arrondi par troncature 191
C.4.2 Arrondi au plus proche 192

Bibliographie **195**

Table des figures

1 Méthodologie du passage d'un filtre linéaire au code l'implémentation en virgule fixe. 5
2 Plan schématisé du mémoire. 6

1.1 Le filtre $\mathcal{H}$ calcule la sortie $\boldsymbol{y}(k)$ à partir de l'entrée $\boldsymbol{u}(k)$ à tout instant k. 11
1.2 Les trois types d'opérateurs d'un GFD d'un filtre LTI. 20
1.3 Graphe flot de données pour la représentation d'état. 21
1.4 Graphe flot de données pour la Forme Directe I. 22
1.5 Graphe flot de données pour la Forme Directe II. 22
1.6 Graphe flot de données pour la Forme Directe II transposée. 23
1.7 La ρDFIIt par d'une DFIIt et change tous les q^{-1} en ρ_k^{-1}. 24
1.8 Décomposition en cascade d'un filtre. 26
1.9 Décomposition parallèle d'un filtre. 26

2.1 Décomposition binaire d'un entier naturel X. 36
2.2 Représentation Signe-Valeur Absolue des entiers de 4 bits 37
2.3 Représentation en complément à un des entiers de 4 bits 38
2.4 Représentation en complément à deux des entiers de 4 bits 39
2.5 Représentation de la répartition des bits pour un nombre en virgule flottante simple précision dans la norme IEEE754. 40
2.6 Représentation de $\tilde{\pi}$ en flottant simple précision dans la norme IEEE754. 41
2.7 Représentation d'un nombre en arithmétique virgule fixe. 42
2.8 D'autres représentations de nombres en arithmétique virgule fixe. Dans le premier cas ℓ est positif et dans le second m est négatif. . . 44
2.9 Représentation virgule fixe du nombre 5.5 où la position du msb a été surestimé. En effet le FPF est $(5, -1)$ alors que $(3, -1)$ était suffisant, et donc il y a deux bits de signes inutiles dans ce cas là. . 45
2.10 Extension du bit de signe dans le cas $m' > m$. 48

2.11 Illustration d'un cas limite avec la conversion de $c = 127.6$ sur 8 bits (cas limite) et sur 9 bits (cas non limite). Sur 8 bits, la retenue de l'arrondi au plus proche de c engendre un changement de la position du bit le plus significatif. 50
2.12 Illustration des différents cas de formatage pour une constante c en matériel. 53
2.13 Décalage à droite de C dans le cas où $m' \geqslant m$ avec $\ell' > \ell$. 54
2.14 Cas $m' < m$, on calcule $(C \ll d) \gg d$. 54
2.15 Illustration de la somme de deux opérandes dans trois différents cas. 61
2.16 Exemple d'une somme en considérant trois cas différents. 64
2.17 Pire cas atteint pour $x = c \cdot v$ avec $c = -2^{m_c}$ et $v = -2^{m_v}$. 65
2.18 Illustration de la multiplication en virgule fixe. 66
2.19 Exemple de multiplication virgule fixe. 67
2.20 Arrondi par troncature. $Q(x)$ est l'opération de quantification de $x \in \mathbb{R}$ et q le pas de quantification. 68
2.21 Arrondi au plus proche. $Q(x)$ est l'opération de quantification de $x \in \mathbb{R}$ et q le pas de quantification. 70
2.22 Illustration des différents modes d'arrondi d'une constante réelle c. 73
2.23 Modélisation du processus de quantification. La dégradation e peut être modélisée par un intervalle ou par un bruit. 73
2.24 Illustration des pires cas pour l'erreur d'arrondi par troncature. . . 76
2.25 Illustration des pires cas pour l'erreur d'arrondi au plus proche. . . 76
2.26 Propagation d'une erreur dans une multiplication. 77

3.1 Équivalence entre le système dégradé $\mathcal{H}^*$ et la décomposition en deux systèmes distincts $\mathcal{H}$ et $\mathcal{H}_\varepsilon$. 83
3.2 Équivalence entre le système dégradé $\widetilde{\mathcal{H}}$ et la décomposition en deux systèmes distincts $\mathcal{H}$ et $\widetilde{\mathcal{H}}_\varepsilon$. 87

4.1 $(4 + 1.5) + 0.5 = 5$. 98
4.2 $(1.5 + 0.5) + 4 = 6$. 98
4.3 Légende des différents noeuds des arbres syntaxiques. 99
4.4 oSoP correspondant au calcul $((\boldsymbol{p}_1 + \boldsymbol{p}_4) + \boldsymbol{p}_2) + \boldsymbol{p}_3$. 100
4.5 oSoP correspondant au calcul $(\boldsymbol{p}_1 + \boldsymbol{p}_3) + (\boldsymbol{p}_2 + \boldsymbol{p}_4)$. 100
4.6 Passage d'un multiplieur constante/variable à une feuille produit. . 100
4.7 Passage d'un oSoP de taille $N = 2$ à trois oSoP de taille $N = 3$. . . 101
4.8 Les cinq oSoP considérés par le nombre de catalan C_3. 102
4.9 Les deux premières étapes de la génération des oSoP par l'utilisation d'une pile, pour $N = 3$. 106
4.10 oSoP avant propagation des formats. 108
4.11 oSoP après propagation des intervalles. 108
4.12 Deux oSoP de même ordre illustrant la notion de parallélisme. . . . 112
4.13 oSoP de hauteur minimale. 113
4.14 oSoP annoté par l'algorithme de Sethi-Ullman. 116

4.15 Les 4 meilleurs oSoP pour l'exemple considéré avec calculs sur 32 bits. 122
4.16 Différence entre la sortie y flottante double précision et la sortie 32 bits du code généré. 124
4.17 Différence entre la sortie y flottante double précision et la sortie 32 bits du code généré en considérant l'ensemble des erreurs. 125

5.1 Illustration du formatage de bits. 136
5.2 oSoP obtenu en appliquant le formatage. 139
5.3 Illustration sous forme d'une somme de la résolution du problème des largeurs minimales dans le cas d'une DFI. 150
5.4 Illustration sous forme d'un oSoP de la résolution du problème des largeurs minimales dans le cas d'une DFI. 151
5.5 Représentation des sommes avec les formats obtenus d'après la résolution du problème d'optimisation. 153
5.6 Différence entre la sortie y flottante double précision et la sortie virgule fixe du code généré à partir des résultats de l'optimisation. Les droites vertes (extérieures) représentent les bornes attendues sur l'erreur, et les droites rouges (intérieures) illustrent les bornes théoriques sur l'erreur. 154

6.1 Schéma du flot `FiPoGen` pour l'implémentation logcielle. 156
6.2 Schéma du flot `FiPoGen` pour l'implémentation matérielle. 157
6.3 Comparaison des erreurs produites par les différents state-space par rapport à une implémentation double précision de référence. Légende : SSeq = SS_{eq}, SSal = SS_{al}, SSL2 = $SS_{\mathcal{L}_2}$, SSEH = SS_{ε_H}, SSEp = SS_{ε_p}, SSEHp= $SS_{\varepsilon_{Hp}}$. 165
6.4 Comparaison des erreurs produites par les différents state-space par rapport à une implémentation double précision de référence. Légende : SSeq = SS_{eq}, SSL2 = $SS_{\mathcal{L}_2}$, SSEH = SS_{ε_H}, SSEp = SS_{ε_p}, SSEHp= $SS_{\varepsilon_{Hp}}$. 165
6.5 Comparaison des erreurs produites par le state-space $SS_{\mathcal{L}_2}$ et les réalisations non state-space, par rapport à une implémentation double précision de référence. Légende : SSL2 = $SS_{\mathcal{L}_2}$, SIFrho = SIF_ρ, SIFlgs = SIF_{LGS}. 166
6.6 Comparaison des erreurs produites par les différentes réalisations par rapport à une implémentation double précision de référence. Légende : SSL2 = $SS_{\mathcal{L}_2}$, SSEH = SS_{ε_H}, SSEp = SS_{ε_p}, SSEHp= $SS_{\varepsilon_{Hp}}$, SIFrho = SIF_ρ, SIFlgs = SIF_{LGS}. 170

Liste des tableaux

2.1 Tailles des différentes composantes des deux principaux formats spécifiés par la norme IEEE754, et valeurs du biais pour les précisions simple et double [80]. 40
2.2 Tailles des différentes composantes des formats spécifiés par la norme IEEE754, et valeurs du biais pour les précisions simple et double [80]. 41
2.3 Moments d'ordre un et deux pour un bruit e pour un pas de quantification 2^ℓ et d bis pour le nombre de bits arrondis pour le bruit discret. 75
2.4 Bornes de l'intervalle d'erreur e pour un pas de quantification 2^ℓ et d bits pour le nombre de bits arrondis en précision finie. 76

6.1 Comparaison des temps de génération et des nombres d'opérations pour chaque réalisation. 163
6.2 Comparaison des intervalles d'erreur théoriques pour chaque réalisation. 164
6.3 Résultats retournés par l'optimisation du solveur `Bonmin`. 167
6.4 Résultats finaux après déduction des largeurs des coefficients à partir des résultats trouvés par `Bonmin`. 168

Notations et conventions

Notations

$\triangleq$	égal par définition
$\mathcal{H}$	le filtre
H	la fonction de transfert
h	la réponse impulsionnelle
$\langle\!\langle \mathcal{H} \rangle\!\rangle_{\text{wcpg}}$	matrice de *worst case peak gain* (gain dans le pire des cas) du filtre $\mathcal{H}$
$\boldsymbol{M}^\top$	la matrice transposée de la matrice $\boldsymbol{M}$ (notation également utilisée pour le transposé d'un vecteur)
$\text{tr}(\boldsymbol{M})$	la trace de la matrice $\boldsymbol{M}$
$\mathbb{B}$	ensemble des valeurs binaires : $\mathbb{B} \triangleq \{0, 1\}$
$\mathbb{N}$	ensemble des entiers naturels $\{0, 1, 2, 3, \ldots\}$
$\mathbb{Z}$	ensemble des entiers relatifs
$\lfloor x \rfloor$	arrondi à l'entier le plus proche par défaut
$\lceil x \rceil$	arrondi à l'entier le plus proche par excès
$\lfloor x \rceil$	arrondi à l'entier le plus proche
$\nabla_d(x)$	arrondi virgule fixe par troncature par rapport au d-ième bit $\nabla_d(x) \triangleq \lfloor \frac{x}{2^d} \rfloor \cdot 2^d$

$\circ_d(x)$	arrondi virgule fixe au plus proche par rapport au d-ième bit $\circ_d(x) \triangleq \lfloor \frac{x}{2^d} \rceil \cdot 2^d$
$\triangle_d(x)$	arrondi virgule fixe vers $+\infty$ par rapport au d-ième bit $\triangle_d(x) \triangleq \lceil \frac{x}{2^d} \rceil \cdot 2^d$
$[\underline{x}; \overline{x}]$	bornes inférieure/supérieure d'un intervalle réel $[\underline{x}; \overline{x}] = \{x \in \mathbb{R} \vert \underline{x} \leqslant x \leqslant \overline{x}\}$
$\langle x_m, x_r \rangle$	centre/rayon d'un intervalle réel $\langle x_m, x_r \rangle = \{x \in \mathbb{R} \vert \, \vert x - x_m \vert \leqslant x_r\}$ $\langle x_m, x_r \rangle = [x_m - x_r; x_m + x_r]$

Conventions

Scalaires / vecteurs / matrices : Sauf exceptions, une quantité scalaire sera notée en minuscule et sans graisse (par exemple x), un vecteur sera noté en minuscule et en gras (par exemple $\boldsymbol{x}$), et une matrice sera notée en majuscule et en gras (par exemple $\boldsymbol{X}$).

Remarque (Exceptions)**.** *Les notations usuelles d'un filtre, de sa réponse impulsionnelle et de sa fonction de transfert sont respectivement $\mathcal{H}$, h et H. Dans le cas d'un filtre à plusieurs entrées et plusieurs sorties, on notera $\boldsymbol{h}$ la matrice des réponses impulsionnelles et $\boldsymbol{H}$ la fonction de transfert matricielle.*

Intervalle vectoriel : Soit $\boldsymbol{u}$ un vecteur. On note $\langle \boldsymbol{u}_m, \boldsymbol{u}_r \rangle$ (resp. $[\underline{\boldsymbol{u}}; \overline{\boldsymbol{u}}]$) le vecteur des intervalles des composantes de $\boldsymbol{u}$ en notation centre/rayon (resp. bornes inf/sup), c'est-à-dire on a $\boldsymbol{u}_i \in \langle \boldsymbol{u}_m, \boldsymbol{u}_r \rangle_i \triangleq \langle \boldsymbol{u}_{m,i}, \boldsymbol{u}_{r,i} \rangle$ (resp. $\boldsymbol{u}_i \in [\underline{\boldsymbol{u}}; \overline{\boldsymbol{u}}]_i \triangleq [\underline{\boldsymbol{u}_i}; \overline{\boldsymbol{u}_i}]$) pour $1 \leqslant i \leqslant n_u$ avec n_u la taille de $\boldsymbol{u}$. De plus, on notera $\boldsymbol{u} \in \langle \boldsymbol{u}_m, \boldsymbol{u}_r \rangle$ (resp. $\boldsymbol{u} \in [\underline{\boldsymbol{u}}; \overline{\boldsymbol{u}}]$) l'appartenance terme à terme de $\boldsymbol{u}$ à l'intervalle $\langle \boldsymbol{u}_m, \boldsymbol{u}_r \rangle$ (resp. à l'intervalle $[\underline{\boldsymbol{u}}; \overline{\boldsymbol{u}}]$).

Écriture des nombres décimaux : Les nombres réels (ou flottants) seront représentés avec 6 chiffres significatifs uniquement, en considérant un arrondi au plus proche des chiffres supprimés. Pour les intervalles réels, la borne inférieure sera arrondi vers $-\infty$ et la borne supérieure vers $+\infty$, toujours sur 6 chiffres significatifs.

Introduction

Les travaux réalisés lors de cette thèse traitent de l'implémentation [1] de filtres linéaires en arithmétique virgule fixe, à destination des systèmes embarqués. Pour comprendre l'intérêt de ces travaux, il est tout d'abord nécessaire d'en situer le contexte.

Contexte

Ces dernières décennies, les appareils électroniques autonomes sont en pleine expansion dans notre quotidien, grâce aux progrès technologiques permettant de réduire toujours un peu plus la taille des processeurs. En effet, avec l'essor des *smartphones*, de l'électronique embarqué dans l'automobile, ou encore la mode plus récente des drones et des caméras miniatures, les systèmes embarqués sont plus que jamais au centre des préoccupations des constructeurs d'électronique. Ces systèmes sont de plus en plus connectés à des réseaux et possèdent des applications leur permettant de communiquer entre eux ou avec des systèmes non-embarqués, ou d'être localisés géographiquement, et ces applications effectuent des traitements de données, comme par exemple du traitement de signal ou d'image. Un smartphone possède de nombreuses applications de traitement du signal et de l'image : lors d'une communication, le bruit ambiant peut être diminué pour une meilleure qualité d'écoute entre les intervenants, les appareils permettent maintenant l'appel vidéo avec une transmission de l'image à l'interlocuteur et réception de son image, en temps réel, etc. Pour communiquer, ces appareils peuvent se connecter sur de longues distances via des réseaux de télécommunications (réseaux de deuxième, troisième ou quatrième génération), mais aussi sur des distances plus courtes via

1. Les termes *implanter* et *implémenter* sont parfois ambigus en français (le second étant un anglicisme, officiellement adopté dans la langue française en 2007), on fait donc la distinction suivante pour éviter toute confusion : *implanter* désigne l'action de coder sur une cible (*mettre le code à l'intérieur de la cible*), tandis qu'*implémenter* désigne l'action de décrire une suite d'opérations par un algorithme ou du code. Ainsi, la phase d'implémentation viendra nécessairement toujours avant la phase d'implantation.

des réseaux locaux sans fil (WiFi [2], Bluetooth, ZigBee, etc.), et utilisent alors des protocoles de communication avec les réseaux en "temps réel" afin d'assurer une connexion continue et la meilleure communication possible.

De par la diversité des applications des systèmes embarqués (on a parlé de la téléphonie et de l'automobile, mais on aurait pu également parler de la robotique, de l'aérospatiale, etc.) de leur automonie, et de la très forte demande, les systèmes embarqués sont sujets à diverses contraintes fortes :

- le coût : ces systèmes étant souvent fabriqués en grande série, il est nécessaire de réduire au maximum le coût de fabrication par système. Plus la surface occupée par le circuit et la mémoire est grande, plus le coût de fabrication est élevé, diminuer le coût passe donc par minimiser la surface du circuit et celle de la mémoire.
- l'autonomie, ou consommation d'énergie : c'est un enjeu majeur des systèmes embarqués, afin de maximiser le temps d'utilisation d'un appareil en autonomie il est nécessaire de minimiser la consommation d'énergie. Ce constat est particulièrement vrai et observable dans le contexte des réseaux de capteurs par exemple, où un capteur peut être lâché dans la nature (pour des applications militaires comme la détection de mouvement ou la reconnaissance terrestre de l'environnement, ou encore pour des applications environnementales comme la dispersion de thermo-capteurs pour prévenir les feux de forets, etc. [91]) et faire des transmissions de relevés locaux jusqu'à consommation totale de la batterie.
- le temps de développement : pour garantir la rentabilité d'un produit, il faut diminuer au maximum le temps avant sa mise sur le marché et donc son temps de développement.
- la fiabilité du système : un système doit répondre à certaines spécifications numériques, en terme de performance et de précision, il est donc nécessaire de rendre le système le plus fiable possible à ces spécifications.

Il existe principalement deux représentations des nombres utilisées dans les systèmes embarqués, la représentation *virgule fixe* (Fixed-Point, FxP) et la représentation *virgule flottante* (Floating-Point, FlP). La première utilise les nombres entiers comme une approximation des nombres réels par le biais d'un facteur d'échelle implicite fixé (le développeur sait où se situe chaque facteur d'échelle et écrit son algorithme avec des nombres entiers en faisant les alignements nécessaires), tandis que la seconde représentation utilise une mantisse et un facteur d'échelle explicite (codé dans la représentation) qui s'adapte au cours du traitement. Cette adaptation de l'exposant est gérée par le matériel et par conséquent les opérateurs FlP sont plus gros et plus lents que les opérateurs FxP, puisqu'ils ne sont rien d'autres que les opérateurs entiers classiques. Si les opérateurs flottants sont de plus en plus présents dans les systèmes embarqués, ils ne sont pas généralisés, alors qu'on trouve des

2. Wireless Fidelity

opérateurs FxP sur tous les systèmes embarqués. Avant d'avoir des opérateurs FlP, la seule façon de faire était d'utiliser des opérateurs FxP, les programmes de la mission Apollo par exemple, ont été codés en FxP [33].

Dans les systèmes embarqués du traitement du signal, l'arithmétique virgule fixe est généralement préférée. En effet, cette arithmétique est moins coûteuse (elle nécessite uniquement des unités de calculs en entiers), et permet d'utiliser des largeurs arbitraires. Si l'arithmétique FxP permet moins de modularité et de précision que l'arithmétique FlP, il faut cependant noter que les applications de traitement du signal sont généralement tolérantes à une précision relativement faible. Les contraintes des systèmes embarqués sont donc compatibles avec l'utilisation de cette arithmétique (ce point sera plus détaillé dans le chapitre 2).

Objectifs

L'un des désavantages de l'arithmétique FxP par rapport à l'arithmétique FlP, en plus de ceux déjà cités, est que la FxP nécessite un travail plus important de la part du développeur. En effet, celui-ci doit implémenter son algorithme en prenant en compte les différents facteurs d'échelle tandis qu'en FlP tout est géré par les opérateurs, le temps passé à implémenter en FxP est donc plus long qu'en FlP. En effet, pour implémenter un algorithme en FxP, puisque le facteur d'échelle (la position implicite de la virgule) est fixé une fois pour toutes lors de l'implémentation, il faut déterminer les facteurs d'échelle de chaque constante, chaque variable et surtout de chaque calcul intermédiaire. Or, nous avons vu qu'il était nécessaire de minimiser le temps de développement des systèmes embarqués. Le premier objectif est donc d'automatiser le plus possible ce processus d'implémentation en virgule fixe.

Un autre objectif est de minimiser, en fonction de la cible, les dégradations numériques subies par un filtre lors du passage en virgule fixe. Les sources de dégradations numériques en précision finie sont de deux types :

- la quantification des coefficients : les algorithmes de filtres sont spécifiés avec des coefficients réels, et il est donc nécessaire de quantifier (ou convertir) ces coefficients en virgule fixe sur des largeurs (la largeur est le nombre de bits représentant une valeur) qui sont soit fixées soit déterminées pour répondre à des contraintes de précision ;
- les arrondis dans les calculs : chaque valeur intermédiaire dans un calcul possède son propre facteur d'échelle, et il est par exemple nécessaire que deux valeurs possèdent le même facteur d'échelle pour être ajoutées ou soustraites. Cette opération d'*alignement* peut produire une dégradation.

Minimiser les erreurs de quantification revient à prendre des largeurs plus grandes pour représenter les coefficients (plus on a de bits, plus on est précis), mais cela va à l'encontre de la minimisation de la surface d'implantation et du coût matériel. Il est donc nécessaire de trouver un compromis entre minimiser

les erreurs de quantification et minimiser la surface et le coût. De plus, minimiser les arrondis de calculs revient à déterminer le schéma de calcul (l'ordre des opérations) qui va produire le moins d'arrondis de calcul.

On peut résumer l'objectif global par proposer une méthodologie et des outils pour implémenter de la meilleure façon possible (au sens de la précision du calcul et par rapport à la cible visée) un filtre en arithmétique virgule fixe. La section suivante présente cette méthodologie, dans laquelle s'insère les travaux présentés dans ce mémoire.

Méthodologie

Pour répondre à ces objectifs, une méthodologie a été mise en place, qui décrit le traitement d'un filtre jusqu'à la génération de code l'implémentant en virgule fixe. La figure 1 résume cette méthodologie. Étant donné un filtre linéaire, il existe une infinité de relations exprimant la sortie en fonction de l'entrée à chaque instant, appelée *réalisation*. La première étape est alors de déterminer quelle est la meilleure réalisation pour ce filtre, par rapport à des mesures de sensibilité. Ces mesures permettent d'estimer l'impact de la quantification des coefficients sur la fonction de transfert du filtre, la réalisation la moins sensible à ces quantifications est sélectionnée comme réalisation *optimale* (le terme est utilisé ici de façon abusive, en effet dans le cas multi-objectifs il n'y a pas de réalisation optimale). Cette étape se positionne en amont des travaux réalisés durant la présente thèse, mais consiste en un point essentiel de notre approche comme nous le verrons dans le premier chapitre.

À partir de cette réalisation, on peut distinguer deux différents cas, le cas d'une implémentation pour une cible logicielle (micro-contrôleur, DSP [3]), et le cas d'une implémentation pour une cible matérielle (FPGA [4], ASIC [5]). En *logiciel*, le processeur n'est pas programmable, il est vendu avec des opérateurs intégrés et les largeurs de ces opérateurs sont fixées, c'est donc au niveau algorithmique qu'on va pouvoir déterminer comment implémenter au mieux notre filtre, en fonction de ses opérateurs. Après avoir converti les coefficients du filtre en virgule fixe (sur les largeurs fixées par la cible), on considère alors tous les schémas d'évaluation possibles (en regardant les différents ordres d'addition) et on sélectionne le meilleur (par rapport à des critères comme la précision du calcul ou le degré de parallélisation). En *matériel*, c'est le développeur qui décrit son circuit en fonction d'un algorithme et de ses spécificités (largeurs des variables, des opérateurs, etc) dans un langage de description d'architecture matérielle. Dans ce contexte, puisque les largeurs sont à déterminer, on résout un problème d'optimisation combinatoire qui, à partir de contraintes sur les erreurs, donne l'ensemble optimal (ou sous-optimal si l'algorithme utilisé

3. Digital Signal Processor
4. Field-Programmable Gate Array
5. Application-Specific Integrated Circuit

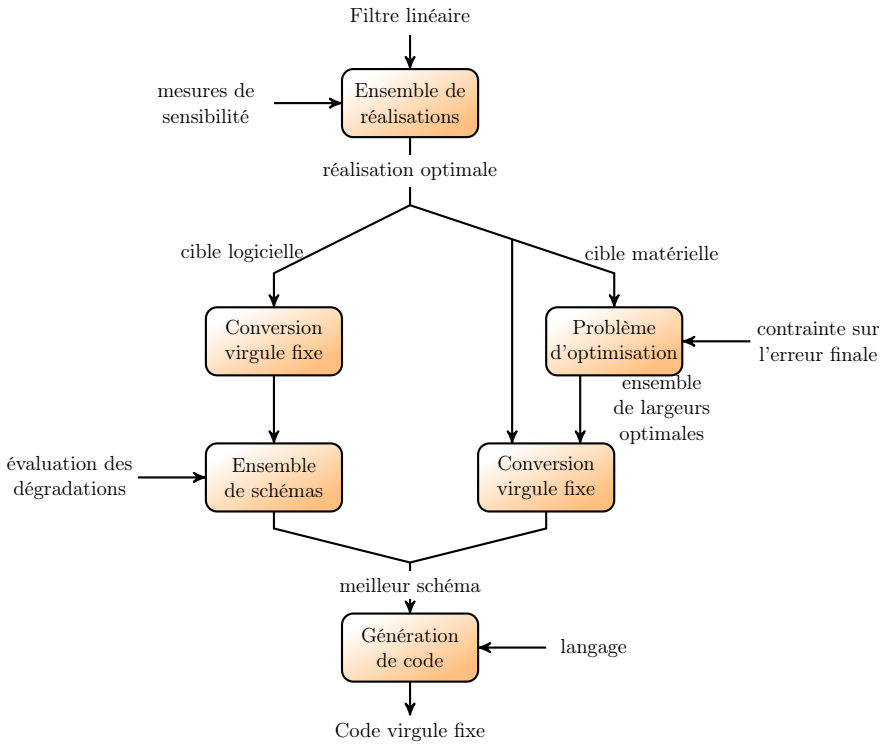


FIGURE 1 – Méthodologie du passage d'un filtre linéaire au code l'implémentation en virgule fixe.

n'est pas déterministe) des largeurs pour un objectif de coût défini. À partir de ces largeurs optimales, la réalisation est convertie et dans notre cas tous les schémas d'évaluation possibles sont équivalents en terme de dégradation numérique, donc n'importe quel schéma peut être sélectionné (ou celui offrant le plus grand degré de parallélisation). Enfin, un code virgule fixe est généré à partir du schéma d'évaluation sélectionné et du langage désiré.

Plan du mémoire

Ce mémoire se compose de six chapitres, articulés comme le montre la figure 2. Dans un premier temps, un chapitre sera consacré aux pré-requis concernant les signaux et les filtres linéaires. Ensuite, l'arithmétique virgule fixe sera introduite et formalisée dans un deuxième chapitre avec la notion de quantification. Une fois les bases des filtres et de la virgule fixe posées, le chapitre 3 analysera les filtres linéaires et leur comportement lors de l'implémentation en virgule fixe. Un quatrième chapitre proposera une méthode pour implémenter des filtres linéaires dans le cas d'une cible logicielle, en introduisant la notion de produits scalaires ordonnés. Le chapitre 5 quant à lui traitera à l'opposé le

cas matériel à travers l'optimisation des largeurs en proposant une méthode et un problème d'optimisation spécifiques à notre approche. L'outil développé pendant cette thèse fera l'objet du chapitre 6 dans lequel seront résumées les différentes méthodes à travers des exemples détaillés. Enfin, nous terminerons par une conclusion et présenterons les perspectives envisagées.

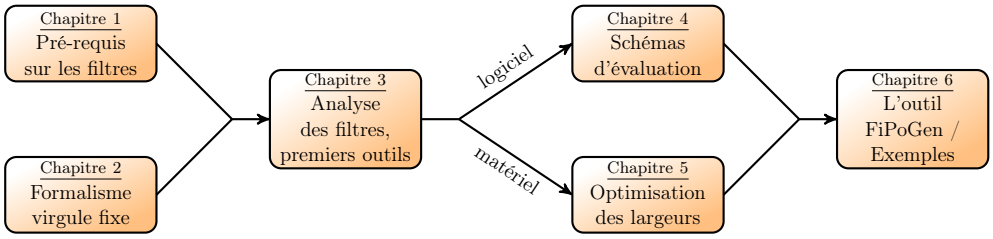


Figure 2 – Plan schématisé du mémoire.

Contributions de la thèse

La liste suivante présente les contributions revendiquées durant cette thèse :

- dans le chapitre 1, l'écriture sous forme SIF des structures LGS et LCW constitue une contribution, bien que mineure, à la généralisation des réalisations d'un filtre linéaire sous cette forme ;
- la formalisation de l'arithmétique virgule fixe proposée dans le chapitre 2 est une contribution dans le sens où, si cette arithmétique est couramment utilisée, on ne trouve pas dans la littérature de formalisation détaillée et complète des opérations usuelles ;
- dans le chapitre 3, la proposition 3.2 et la deuxième partie du corollaire 3.1 sont des contributions. En effet, le *worst-case-peak-gain*, bien que connu [6], n'avait jamais été utilisé pour évaluer l'intervalle d'erreur en sortie d'un filtre. L'analyse de l'erreur en sortie qui en découle, et surtout la formalisation de l'analyse de l'erreur considérant les erreurs de calculs ainsi que les erreurs paramétriques, constituent également des contributions ;
- dans ce même chapitre, une autre contribution est l'utilisation de la règle d'arithmétique modulaire en complément à deux pour une somme de plusieurs termes, décrite dans la section 3.3.2 ;
- la méthodologie d'implémentation logicielle proposée dans le chapitre 4, dont les deux algorithmes de générations des oSoPs, est une contribution ;
- dans le chapitre 5, la formalisation du formatage de bits (section 5.2), ainsi que celle du problème d'optimisation proposé et la méthodologie globale pour l'implémentation matérielle, sont des contributions ;

– l'outil `FiPoGen` développé durant cette thèse constitue une contribution mettant en œuvre les différentes contributions théoriques proposées.

Publications

Les travaux réalisés durant cette thèse ont fait l'objet de trois publications dans des conférences internationales avec comité de lecture :

[45] T. Hilaire et B. Lopez. "Reliable Implementation of Linear Filters with Fixed-Point Arithmetic". Dans : *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*. 2013.

[70] B. Lopez, T. Hilaire et L.-S. Didier. "Formatting Bits to Better Implement Signal Processing Algorithms". Dans : *PECCS 2014 - Proceedings of the 4th International Conference on Pervasive and Embedded Computing and Communication Systems, Lisbon, Portugal, 7-9 January*. 2014, p. 104–111.

[71] B. Lopez, T. Hilaire et L.-S. Didier. "Sum-of-products evaluation schemes with fixed-point arithmetic, and their application to IIR filter implementation". Dans : *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing, DASIP 2012, Karlsruhe, Germany, October 23-25*. 2012, p. 1–8.

Chapitre 1

Filtres linéaires

Ce premier chapitre rappelle quelques pré-requis sur les signaux et les filtres linéaires. Le but est d'introduire les notions qui seront utiles pour décrire et comprendre l'approche adoptée pour implémenter les filtres linéaires en virgule fixe.

Sauf exception, toutes les définitions sont données dans le domaine à temps discret, c'est-à-dire le domaine où les signaux sont échantillonnés à une certaine fréquence et donc représentés par une suite discrète de valeurs.

1.1 Signal discret et filtre linéaire

Cette première section rappelle les définitions de base des signaux et des filtres linéaires qui sont issues de [103] et [90].

1.1.1 Rappels sur les signaux

Définition 1.1 (Signal)**.** *Un signal est une variable temporelle, c'est-à-dire qu'à chaque instant k, x prend une valeur dans un intervalle de $\mathbb{R}$*[1]*. On note $x(k)$ la valeur du signal x à l'instant k, et $\{x(k)\}_{k\geqslant 0}$ l'ensemble des échantillons de x ou le signal x lui-même*[2]*.*

On peut voir x comme une suite d'éléments de $\mathbb{R}$ indexée par $\mathbb{N}$.

De plus on notera $\boldsymbol{x}$ un signal vecteur (composé de n signaux). $\boldsymbol{x}(k) \in \mathbb{R}^{n\times 1}$, la valeur du signal $\boldsymbol{x}$ à l'instant k, sera un vecteur de n valeurs.

Si dans la réalité un signal possède certaines propriétés (un signal peut par exemple suivre une variation sinusoïdale avec une certaine phase et une certaine

1. Dans un cadre plus général un signal x est défini sur l'ensemble $\mathbb{C}$ des nombres complexes mais nous considérerons dans cette thèse exclusivement des signaux réels.

2. S'il existe un signal y tel que $\{x(k)\}_{k\geqslant 0} = \{y(k)\}_{k\geqslant 0}$, c'est-à-dire $x(k) = y(k)$ pour tout $k \geqslant 0$, alors $x = y$, on peut donc utiliser la caractérisation $\{x(k)\}_{k\geqslant 0}$ pour désigner le signal x.

amplitude), lors d'un processus d'implémentation on suppose qu'aucune de ces propriétés n'est connue ni exploitée, mis à part son intervalle de définition. Ce signal pourra alors être interprété comme une variable aléatoire, c'est-à-dire qu'on considère qu'à chaque instant, x peut prendre n'importe quelle valeur de son intervalle de définition avec la même probabilité indépendamment du temps. À défaut de connaître sa distribution dans son intervalle d'existence, on fait donc l'hypothèse d'une distribution uniforme. On peut alors calculer les moments de cette variable (on considère ici uniquement les moments d'ordre un et deux).

Définition 1.2 (Moments d'un signal)**.** *Pour un signal $\boldsymbol{x}$ (avec $\boldsymbol{x}(k) \in \mathbb{R}^{n\times 1}$ pour $k \geqslant 0$), les moments d'ordre un ($\boldsymbol{\mu_x}$) et deux ($\boldsymbol{\sigma}_{\boldsymbol{x}}^2$ et $\boldsymbol{\psi_x}$) sont définis par :*

$$\begin{array}{lcll} \boldsymbol{\mu_x} & \triangleq & E\left\{\{\boldsymbol{x}(k)\}_{k\geqslant 0}\right\} & \in \mathbb{R}^n \\ \boldsymbol{\psi_x} & \triangleq & E\left\{\{(\boldsymbol{x}(k)-\boldsymbol{\mu_x})(\boldsymbol{x}(k)-\boldsymbol{\mu_x})^\top\}_{k\geqslant 0}\right\} & \in \mathbb{R}^{n\times n} \\ \boldsymbol{\sigma}_{\boldsymbol{x}}^2 & \triangleq & E\left\{\{(\boldsymbol{x}(k)-\boldsymbol{\mu_x})^\top(\boldsymbol{x}(k)-\boldsymbol{\mu_x})\}_{k\geqslant 0}\right\} = \operatorname{tr}(\boldsymbol{\psi_x}) & \in \mathbb{R} \end{array} \tag{1.1}$$

où $E\{X\}$ est l'espérance mathématique de la variable aléatoire X (ou du signal $\{X(k)\}_{k\geqslant 0}$ sur toute les valeurs de k).

On donne également les définitions de certaines normes d'un signal, nécessaires par la suite pour définir la norme d'un filtre.

Définition 1.3 (Norme ℓ^1)**.** *La norme ℓ^1 d'un signal scalaire x, notée $\|x\|_{\ell^1}$, est la somme des valeurs absolues de $x(k)$ aux différents instants k, c'est-à-dire :*

$$\|x\|_{\ell^1} \triangleq \sum_{k=0}^{+\infty} |x(k)|\,. \tag{1.2}$$

La norme ℓ^1 d'un signal n'existe que si celui-ci est ℓ^1-sommable, c.-à-d. si la somme de l'équation (1.2) converge.

Définition 1.4 (Norme ℓ^2)**.** *La norme ℓ^2 d'un signal scalaire x, notée $\|x\|_{\ell^2}$, s'exprime ainsi :*

$$\|x\|_{\ell^2} \triangleq \sqrt{\sum_{k=0}^{+\infty} x(k)^2}. \tag{1.3}$$

*L'*énergie *du signal x est définie comme $\|x\|_{\ell^2}^2$.*

La norme ℓ^2 d'un signal n'existe que si celui-ci est carré-sommable, c.-à-d. si la somme de l'équation (1.3) converge.

Définition 1.5 (Norme ℓ^∞)**.** *La norme ℓ^∞ d'un signal scalaire x, notée $\|x\|_{\ell^\infty}$, est le plus petit majorant de toutes les valeurs (en valeur absolue) prises par le signal x au cours du temps, c'est-à-dire :*

$$\|x\|_{\ell^\infty} \triangleq \sup_{k\in\mathbb{N}} |x(k)|\,. \tag{1.4}$$

1.1.2 Filtres linéaires invariants dans le temps

Un filtre $\mathcal{H}$ est une application qui consiste à transformer mathématiquement un signal $\boldsymbol{u}$ de dimension n_u en un signal $\boldsymbol{y} \triangleq \mathcal{H}(\boldsymbol{u})$ de taille n_y. Si $n_u = n_y = 1$, on parlera de filtre SISO (Single-Input-Single-Output[1]), dans le cas contraire, on parlera de filtre MIMO (Multiple-Input-Multiple-Output[1]).

À chaque entrée $\boldsymbol{u}(k)$ correspond une sortie $\boldsymbol{y}(k)$, comme le schématise la figure 1.1.

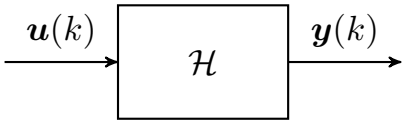


FIGURE 1.1 – Le filtre $\mathcal{H}$ calcule la sortie $\boldsymbol{y}(k)$ à partir de l'entrée $\boldsymbol{u}(k)$ à tout instant k.

Dans nos travaux, nous nous intéressons à une classe particulière de filtres, les filtres linéaires invariants dans le temps.

Définition 1.6 (Filtre linéaires invariants dans le temps)**.** *Un filtre $\mathcal{H}$ est dit* linéaire *si, pour tous signaux $\boldsymbol{u}_1$ et $\boldsymbol{u}_2$ et tous $\alpha, \beta \in \mathbb{R}$, on a :*

$$\mathcal{H}(\alpha \cdot \boldsymbol{u}_1 + \beta \cdot \boldsymbol{u}_2) = \alpha \cdot \mathcal{H}(\boldsymbol{u}_1) + \beta \cdot \mathcal{H}(\boldsymbol{u}_2). \tag{1.5}$$

De plus, soit $\boldsymbol{u}(k)$ la valeur d'un signal à l'instant k et $\boldsymbol{y}(k)$ la sortie de $\mathcal{H}$ pour l'entrée $\boldsymbol{u}(k)$. Le filtre $\mathcal{H}$ est dit invariant dans le temps *si pour tout décalage temporel $k_0 \in \mathbb{N}$, on a :*

$$\{\mathcal{H}(\boldsymbol{u})(k - k_0)\}_{k \geqslant 0} = \mathcal{H}\left(\{\boldsymbol{u}(k - k_0)\}_{k \geqslant 0}\right) \tag{1.6}$$

c'est-à-dire le filtre $\mathcal{H}$ effectue le même calcul sur l'entrée à chaque instant.

*Un filtre linéaire invariant dans le temps (*Linear Time Invariant filter*, ou filtre LTI) est donc un filtre qui respecte à la fois la condition de linéarité et la condition d'invariance dans le temps.*

Remarque 1.1. *Par la suite, tous nos filtres sont considérés BIBO stables (*Bounded-Input, Bounded-Output*), c'est-à-dire que si l'entrée est bornée, alors la sortie l'est aussi. Cette propriété permet de garantir que les sommes infinies considérées par la suite convergent [102].*

1.1.2.1 Réponse impulsionnelle

Un filtre LTI SISO $\mathcal{H}$ peut être caractérisé par sa *réponse impulsionnelle* noté h, qui est la réponse de $\mathcal{H}$ à une impulsion de Dirac δ, définie par :

$$\delta(k) \triangleq \begin{cases} 1 & \text{si } k = 0 \\ 0 & \text{sinon} \end{cases}. \tag{1.7}$$

1. Les abbreviations anglaises étant plus répandues, elles seront utilisées dans ce mémoire.

En effet, toute entrée u peut s'écrire comme la somme pondérée d'impulsions de Dirac :

$$u = \sum_{\ell \geqslant 0} u(\ell)\delta_\ell, \tag{1.8}$$

où δ_ℓ est l'impulsion de Dirac centrée en ℓ, c'est-à-dire :

$$\delta_\ell(k) \triangleq \begin{cases} 1 & \text{si } k = \ell \\ 0 & \text{sinon} \end{cases}. \tag{1.9}$$

La condition de linéarité de $\mathcal{H}$ donne $y = \mathcal{H}(u) = \sum_{\ell \geqslant 0} u(\ell)\mathcal{H}(\delta_\ell)$, et l'invariance dans le temps donne $\mathcal{H}(\delta_\ell)(k) = h(k-\ell)$, et donc :

$$y(k) = \sum_{\ell \geqslant 0} u(\ell)h(k-\ell) = \sum_{\ell=0}^{k} u(k)h(k-\ell), \tag{1.10}$$

qui correspond à la définition du produit de convolution de u et h, que l'on note $y = h * u$.

Dans le cas MIMO, on notera $\boldsymbol{h} \in \mathbb{R}^{n_y \times n_u}$ la réponse impulsionnelle du filtre $\mathcal{H}$, où $\boldsymbol{h}_{i,j}$ est la réponse sur la i-ème sortie à une impulsion de Dirac sur la j-ème entrée seulement (la j-ème colonne de $\boldsymbol{h}$, notée $\boldsymbol{h}_{\cdot,j}$ est obtenue à partir de l'entrée $\boldsymbol{u}$ avec $\boldsymbol{u}_j = \delta$ et $\boldsymbol{u}_i$ est un signal nul pour tout $i \neq j$). De la même manière, on obtient :

$$\boldsymbol{y}_i(k) = \sum_{j=1}^{n_u} \sum_{\ell=0}^{k} \boldsymbol{h}_{i,j}(k-\ell)\boldsymbol{u}_j(\ell), \quad \forall 1 \leqslant i \leqslant n_y, \tag{1.11}$$

que l'on note $\boldsymbol{y} = \boldsymbol{h} * \boldsymbol{u}$.

1.1.2.2 Filtres FIR et IIR

Il existe deux types de filtres LTI, les filtres à *Réponse Impulsionnelle Finie* (*Finite Impulse Response* filter, ou filtre FIR) et les filtres à *Réponse Impulsionnelle Infinie* (*Infinite Impulse Response* filter, ou filtre IIR). Un filtre FIR a, comme son nom l'indique, une réponse impulsionnelle finie, c'est-à-dire qu'il existe un entier n tel que

$$\forall k \geqslant n, \quad h(k) = 0. \tag{1.12}$$

Le plus petit entier n qui vérifie (1.12) est appelé l'ordre du filtre.

Un filtre FIR d'ordre n est caractérisé par la relation temporelle suivante entre l'entrée u et la sortie y à l'instant k :

$$y(k) = \sum_{i=0}^{n} b_i u(k-i). \tag{1.13}$$

La sortie à un instant k n'est fonction que des entrées à l'instant k et aux n instants précédents. En utilisant l'équation (1.10), on note qu'on a la correspondance suivante entre les coefficients du filtre et les valeurs de la réponse impulsionnelle finie :

$$b_i = h(i), \forall 1 \leqslant i \leqslant n.$$

Un filtre IIR d'ordre n est caractérisé quant à lui par la relation temporelle suivante entre l'entrée u et la sortie y à l'instant k, appelée *équation aux différences* :

$$y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=1}^{n} a_i y(k-i). \tag{1.14}$$

Comme pour un filtre FIR, la sortie à un instant k est fonction des entrées à l'instant k et aux n instants précédents mais également des sorties aux n instants précédents, on parle alors de *rebouclage*. Il est possible de déduire de l'équation (1.14) les valeurs de la réponse impulsionnelle, en remplaçant $y(k-1)$ par l'équation (1.14) décalée d'un instant, puis $y(k-2)$ par l'équation (1.14) décalée de deux instants, etc. Il est alors possible de déduire une définition récursive de $h(k)$:

$$h(k) = \begin{cases} 0 & \text{si } k < 0 \\ b_k - \sum\limits_{\ell=1}^{n} a_\ell h(k-\ell) & \text{si } 0 \leqslant k \leqslant n \\ \sum\limits_{\ell=1}^{n} a_\ell h(k-\ell) & \text{si } n < k \end{cases}. \tag{1.15}$$

Dans les travaux décrits dans cette thèse, nous nous intéressons uniquement aux filtres IIR (classe qui englobe les filtres FIR, il suffit de poser $a_i = 0$ pour tout $1 \leqslant i \leqslant n$), la notion de rebouclage jouant un rôle important dans l'approche présentée. Nous désignerons alors par filtre LTI (ou parfois seulement par filtre) un filtre IIR, sauf précision contraire.

1.1.2.3 Fonction de transfert

Un filtre LTI $\mathcal{H}$ peut également être caractérisé par sa fonction de transfert $H : \mathbb{C} \to \mathbb{C}$ dans le domaine fréquentiel. Si le signal d'entrée de $\mathcal{H}$ est une sinusoïde de pulsation ω, alors le signal de sortie est une sinusoïde de pulsation ω, d'amplitude augmentée de $\left|H(e^{j\omega})\right|$ et déphasée de $\mathrm{Arg}(H(e^{j\omega}))$.

Pour passer du domaine temporel au domaine fréquentiel, on utilise la transformée[3] en z. La transformée en z est linéaire, et si x et x' sont deux signaux tels que $x'(k-k_0) = x(k)$ pour tout $k \geqslant 0$ (x' est le signal x décalé

3. La transformation en z, notée $\mathcal{Z}\{\cdot\}$, est une application qui consiste à transformer un signal x en une fonction X d'une variable complexe z qui exprime l'évolution fréquentielle du signal. On note $X(z)$ la transformée en z d'un signal x et on a $X(z) = \mathcal{Z}\{x\} \triangleq \sum\limits_{k=0}^{+\infty} x(k) z^{-k}$.[94]

de k_0), alors la transformée en z vérifie $\mathcal{Z}\{x'\} = \mathcal{Z}\{x\} \cdot z^{-k_0}$. Ainsi, si on note U et Y les transformées en z de l'entrée u et de la sortie y respectivement, l'équation (1.14) devient, par linéarité :

$$Y(z) = \sum_{i=0}^{n} b_i U(z) z^{-i} - \sum_{i=1}^{n} a_i Y(z) z^{-i}. \tag{1.16}$$

En notant $H(z)$ le rapport des transformées en z de la sortie et de l'entrée, on a :

$$H(z) = \frac{Y(z)}{U(z)} = \frac{\sum\limits_{i=0}^{n} b_i z^{-i}}{1 + \sum\limits_{i=1}^{n} a_i z^{-i}}, \quad \forall z \in \mathbb{C}. \tag{1.17}$$

La fonction H est appelée fonction de transfert du filtre $\mathcal{H}$ et donne l'expression du filtre dans le domaine fréquentiel. Elle correspond aussi à la transformée en z de la réponse impulsionnelle h.

Dans le cas MIMO, la fonction de transfert $\boldsymbol{H}$ est une matrice de taille $n_y \times n_u$, où $\boldsymbol{H}_{i,j}$ exprime le transfert entre la j-ème entrée et la i-ème sortie.

1.1.2.4 Réalisations

L'équation (1.14) permet de décrire la relation temporelle algorithmique entre les sorties et les entrées pour un filtre, mais n'est pas la seule façon de décrire cette relation [7, 29, 42, 68]. Ces différents algorithmes de calcul sont appelés des *réalisations* d'un filtre. La section 1.3 présentera quelques réalisations connues.

Définition 1.7 (Réalisation)**.** *On peut définir une* réalisation *comme un* algorithme *détaillant comment calculer les sorties en fonction des entrées (en utilisant possiblement des variables dites d'état ou intermédiaires),* ***mais*** *ne décrivant pas comment sont faites les opérations (sur quel format, dans quel ordre, avec quel mode d'arrondi, etc.).*

Si toutes les réalisations sont équivalentes mathématiquement en précision infinie, elles ne le sont plus en précision finie, il y a donc un véritable enjeu dans le choix d'une réalisation pour un filtre donné [29, 58, 89]. La section 1.3 présente quelques une de ces réalisations.

1.1.2.5 Normes de filtres

On définit dans cette section quelques normes de filtres qui nous seront utiles dans notre approche, comme nous le verrons dans le chapitre 3. On commence par introduire la norme de Frobenius afin de définir par la suite la norme $\mathcal{L}_2$.

Définition 1.8 (Norme de Frobenius)**.** *Soit une matrice complexe* $\boldsymbol{M} \in \mathbb{C}^{m\times n}$*. La norme de Frobenius* $\|\boldsymbol{M}\|_F$ *est le nombre réel défini par*

$$\|\boldsymbol{M}\|_F \triangleq \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|\boldsymbol{M}_{ij}|^2}. \tag{1.18}$$

Définition 1.9 (Norme $\mathcal{L}_2$)**.** *La norme* $\mathcal{L}_2$ *d'un filtre* $\mathcal{H}$*, notée* $\|\mathcal{H}\|_2$*, est donnée par l'équation suivante :*

$$\|\mathcal{H}\|_2 \triangleq \left(\frac{1}{2\pi}\int_0^{2\pi}\left\|\boldsymbol{H}(e^{j\omega})\right\|_F^2 d\omega\right)^{\frac{1}{2}}. \tag{1.19}$$

En effet, $\boldsymbol{H}(e^{j\omega})$ *donne le gain du filtre pour une sinusoïde de pulsation* ω*, on intègre donc le carré de* $\left|\boldsymbol{H}(e^{j\omega})\right|$ *sur toutes les pulsations, de 0 à* 2π*.*

Remarque 1.2. *Dans la définition précédente,* j *désigne l'unité imaginaire, c'est-à-dire le nombre complexe qui vérifie* $j^2 = -1$*. Cette notation est utilisée à la place du* i *usuel car plus répandue dans la communauté du traitement du signal.*

La norme $\mathcal{L}_2$ sera utilisée par la proposition 3.1.

Définition 1.10 (Worst Case Peak Gain)**.** *Le gain dans le pire des cas (en anglais* Worst Case Peak Gain*, WCPG) d'un filtre* $\mathcal{H}$ *correspond à la plus grande amplification que peut subir un signal d'entrée à travers ce filtre. Le WCPG de* $\mathcal{H}$*, noté* $\|\mathcal{H}\|_{\text{wcpg}}$*, s'exprime ainsi :*

$$\|\mathcal{H}\|_{\text{wcpg}} \triangleq \sup_{u\neq 0}\frac{\|h * u\|_{\ell^\infty}}{\|u\|_{\ell^\infty}} \tag{1.20}$$

où h *est la réponse impulsionnelle du filtre* $\mathcal{H}$*,* u *est le signal d'entrée du filtre, et* $h * u$ *est le produit de convolution entre* h *et* u *(*i.e. *le signal de sortie du filtre* $\mathcal{H}$ *pour l'entrée* u*).*

Le WCPG de $\mathcal{H}$ *peut se calculer à partir de la norme* ℓ_1 *de sa réponse impulsionnelle* h *:*

$$\|\mathcal{H}\|_{\text{wcpg}} = \sum_{k\geqslant 0}|h(k)|, \tag{1.21}$$

La proposition 3.2 ainsi que sa preuve C.2 démontreront cette affirmation. On calcule généralement le WCPG d'après cette somme sur un nombre fini de termes, il a en effet été montré que le WCPG peut être calculé à une précision arbitraire sur un nombre fini de termes de la somme [6].

Dans le cas d'un filtre MIMO, on notera $\langle\!\langle\mathcal{H}\rangle\!\rangle_{\text{wcpg}}$ *la matrice des WCPG de chaque sous-système, c.-à-d.* $(\langle\!\langle\mathcal{H}\rangle\!\rangle_{\text{wcpg}})_{i,j} \triangleq \|\mathcal{H}_{i,j}\|_{\text{wcpg}}$*.*

Le WCPG sera utilisé par la proposition 3.2.

On définit également dans cette section la notion de DC-Gain, car bien que n'étant pas une norme, le DC-Gain est utilisé, dans notre approche, conjointement avec le WCPG ou avec la norme $\mathcal{L}_2$.

Définition 1.11 (DC-Gain)**.** *Le DC-gain, ou gain statique, d'un filtre $\mathcal{H}$ correspond au gain à fréquence nulle. Il est noté $|\mathcal{H}|_{\mathrm{DC}}$ et s'exprime ainsi :*

$$|\mathcal{H}|_{\mathrm{DC}} \triangleq H(e^{j\omega})_{|\omega=0} = H(1) \tag{1.22}$$

où H est la fonction de transfert du filtre $\mathcal{H}$.

Dans le cas d'un filtre MIMO, on notera $\langle\!\langle\mathcal{H}\rangle\!\rangle_{\mathrm{DC}}$ la matrice des DC-gain de chaque terme.

Nous verrons dans la section 3.1 que ces normes permettent de mettre en avant des propriétés intéressantes des filtres linéaires, elles permettent par exemple d'exprimer la transformation subie par un intervalle ou une variable aléatoire à travers un filtre linéaire.

1.2 Représentation d'état

La première réalisation décrite ici, généralement utilisée par la communauté de l'*automatique* et beaucoup moins par la communauté du *traitement du signal*, est la représentation d'état (on utilisera généralement dans cette thèse le terme anglais, *state-space*, et son abréviation *SS*). Cette réalisation consiste à considérer l'état instantané du système et à exprimer l'évolution de cet état dans le temps. Cette évolution est dans le cas continu exprimée par des équations différentielles du premier ordre, mais dans le cas discret cette évolution s'exprime par une récurrence, en effet l'état à l'instant $k+1$ est fonction de l'état à l'instant k et de l'entrée à l'instant k. Généralement, cet état est représenté par un vecteur, appelé *vecteur d'état*, et se note $\boldsymbol{x}$. Ainsi, une représentation d'état s'écrit de la façon suivante :

$$\begin{cases} \boldsymbol{x}(k+1) &= \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}\boldsymbol{u}(k) \\ \boldsymbol{y}(k) &= \boldsymbol{C}\boldsymbol{x}(k) + \boldsymbol{D}\boldsymbol{u}(k) \end{cases} \tag{1.23}$$

où $\boldsymbol{x}(k) \in \mathbb{R}^{n_x}$ est le vecteur d'état, $\boldsymbol{u}(k) \in \mathbb{R}^{n_u}$ le vecteur d'entrées et $\boldsymbol{y}(k) \in \mathbb{R}^{n_y}$ le vecteur de sorties, à l'instant k. Les matrices $\boldsymbol{A} \in \mathbb{R}^{n_x \times n_x}$, $\boldsymbol{B} \in \mathbb{R}^{n_x \times n_u}$, $\boldsymbol{C} \in \mathbb{R}^{n_y \times n_x}$, $\boldsymbol{D} \in \mathbb{R}^{n_y \times n_u}$ et la valeur $\boldsymbol{x}(0)$ suffisent à caractériser un filtre LTI. La représentation d'état est donnée ici dans le cas d'un filtre MIMO, dans le cas SISO on a $n_u = n_y = 1$ et donc les matrices $\boldsymbol{C}$ et $\boldsymbol{B}$ deviennent des vecteurs et la matrice $\boldsymbol{D}$ devient un scalaire.

Le signal $\boldsymbol{x}(k)$, tout comme le signal d'entrée $\boldsymbol{u}(k)$, est supposé nul pour tout $k < 0$.

La fonction de transfert d'un filtre décrit par l'équation (1.23) peut s'obtenir en exprimant la transformée en z de cette équation :

$$\boldsymbol{H}(z) \triangleq \boldsymbol{C}(z\boldsymbol{I}_{n_x} - \boldsymbol{A})^{-1}\boldsymbol{B} + \boldsymbol{D}, \quad \forall z \in \mathbb{C}. \tag{1.24}$$

L'expression sous forme de représentation d'état d'un filtre n'est pas unique. En effet, si le filtre $\mathcal{H}$ est décrit d'après les matrices $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$, alors tout filtre défini par les matrices $(\boldsymbol{T}^{-1}\boldsymbol{A}\boldsymbol{T}, \boldsymbol{T}^{-1}\boldsymbol{B}, \boldsymbol{C}\boldsymbol{T}, \boldsymbol{D})$, avec $\boldsymbol{T} \in \mathbb{R}^{n_x \times n_x}$ non singulière, est mathématiquement équivalent au filtre $\mathcal{H}$ de l'équation (1.23). Un ensemble $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ définissant la représentation d'état d'un filtre $\mathcal{H}$ est appelée une *paramétrisation* de $\mathcal{H}$ [29].

On définit également la notion de *grammiens* de *commandabilité* et d'*observabilité* ainsi que la forme équilibrée.

Définition 1.12 (Grammiens, Forme équilibrée)**.** *Soit une représentation d'état définie par la paramétrisation* $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$*. Les grammiens de commandabilité et d'observabilité, respectivement notés* $\boldsymbol{W}_c$ *et* $\boldsymbol{W}_o$*, sont définis par :*

$$\boldsymbol{W}_c \triangleq \sum_{k=0}^{+\infty} \boldsymbol{A}^k \boldsymbol{B}\boldsymbol{B}^\top \left(\boldsymbol{A}^\top\right)^k, \tag{1.25}$$

$$\boldsymbol{W}_o \triangleq \sum_{k=0}^{+\infty} \left(\boldsymbol{A}^\top\right)^k \boldsymbol{C}^\top \boldsymbol{C}\boldsymbol{A}^k. \tag{1.26}$$

On appelle forme équilibrée une représentation d'état telle que $\boldsymbol{W}_c = \boldsymbol{W}_o$*. Dans le cas d'une forme équilibrée, les grammiens* $\boldsymbol{W}_c$ *et* $\boldsymbol{W}_o$ *sont des matrices diagonales.*

Les grammiens $\boldsymbol{W}_c$ et $\boldsymbol{W}_o$ peuvent se calculer dans la pratique en résolvant les équations de Lyapunov suivantes :

$$\boldsymbol{W}_c = \boldsymbol{A}\boldsymbol{W}_c\boldsymbol{A}^\top + \boldsymbol{B}\boldsymbol{B}^\top, \tag{1.27}$$

$$\boldsymbol{W}_o = \boldsymbol{A}^\top\boldsymbol{W}_o\boldsymbol{A} + \boldsymbol{C}^\top\boldsymbol{C}. \tag{1.28}$$

Laub *et al.* ont proposé un algorithme pour calculer, à partir d'une représentation d'état $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$, la matrice non singulière $\boldsymbol{T}$ telle que

$$(\boldsymbol{T}^{-1}\boldsymbol{A}\boldsymbol{T}, \boldsymbol{T}^{-1}\boldsymbol{B}, \boldsymbol{C}\boldsymbol{T}, \boldsymbol{D})$$

est une forme équilibrée [61].

1.2.1 Le calcul de normes par la représentation d'état

La représentation d'état permet en effet de calculer les normes définies précédemment pour les filtres de façon plus simple. Pour les calculs présentés ici, on suppose que $\mathcal{H}$ est un filtre MIMO décrit dans la représentation d'état avec comme paramétrisation $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$.

La norme $\mathcal{L}_2$ peut se calculer par :

$$\|\mathcal{H}\|_2^2 = \mathrm{tr}(\boldsymbol{D}\boldsymbol{D}^\top + \boldsymbol{C}\boldsymbol{W}_c\boldsymbol{C}^\top), \tag{1.29}$$

ou bien par :

$$\|\mathcal{H}\|_2^2 = \mathrm{tr}(\boldsymbol{D}^\top\boldsymbol{D} + \boldsymbol{B}^\top\boldsymbol{W}_o\boldsymbol{B}). \tag{1.30}$$

Le WCPG quant à lui se calcule par :

$$\langle\!\langle\mathcal{H}\rangle\!\rangle_{\mathrm{wcpg}} = \sum_{k\geqslant 0} |\boldsymbol{C}\boldsymbol{A}^k\boldsymbol{B}| + |\boldsymbol{D}|, \tag{1.31}$$

où la valeur absolue d'une matrice est la matrice des valeurs absolues de ses composantes. En effet on peut montrer que l'on a :

$$\boldsymbol{h}(k) = \begin{cases} \boldsymbol{C}\boldsymbol{A}^{k-1}\boldsymbol{B} & \text{si } k > 0 \\ \boldsymbol{D} & \text{si } k = 0 \\ \boldsymbol{0} & \text{sinon} \end{cases}. \tag{1.32}$$

Enfin, le DC-gain peut être obtenu, d'après la définition et l'équation (1.24), en calculant :

$$\langle\!\langle\mathcal{H}\rangle\!\rangle_{\mathrm{DC}} = \boldsymbol{C}(\boldsymbol{I}_n - \boldsymbol{A})^{-1}\boldsymbol{B} + \boldsymbol{D}. \tag{1.33}$$

Remarque 1.3. *Pour une représentation d'état, la propriété de BIBO stabilité d'un filtre s'exprime par le fait que toutes les valeurs propres de la matrice* $\boldsymbol{A}$ *ont un module strictement plus petit que 1. Ainsi, les sommes infinies précédentes convergent, et de plus on a* $\sum_{k\geqslant 0} \boldsymbol{A}^k = (\boldsymbol{I}_n - \boldsymbol{A})^{-1}$.

1.2.2 Calcul de sensibilité

L'implémentation en virgule fixe d'un filtre amène une dégradation sur chaque coefficient due à leur quantification. Cette quantification des coefficients modifie directement la fonction de transfert. Or on a vu que pour un filtre donné décrit par la représentation d'état, il existe une infinité de paramétrisations possibles, toutes équivalentes en précision infinie mais plus en précision finie. Tavsanoglu et Thiele [100] se sont les premiers intéressés à définir une mesure statistique qui permettrait d'évaluer la déviation subie par la fonction de transfert lors de la quantification des coefficients, et ainsi de décrire le problème de trouver la réalisation optimale par rapport à cette mesure. Cette mesure, appelée *mesure de sensibilité* est basée sur la sensibilité de la fonction de transfert par rapport à la matrice et aux vecteurs de coefficients (pour plus de détails, se reporter à [100]). Les auteurs montrent qu'une forme équilibrée minimise leur mesure.

Gevers et Li [29] ont proposé une amélioration plus naturelle de cette mesure de sensibilité, appelée $\mathcal{L}_2$-*sensibilité*. En effet, la mesure de Tavsanoglu et

Thiele utilise deux normes, la norme $\mathcal{L}_1$ (non définie dans ce mémoire) et la norme $\mathcal{L}_2$, tandis que la mesure de Gevers et Li est basée uniquement sur la norme $\mathcal{L}_2$. Les auteurs expliquent qu'utiliser seulement la norme $\mathcal{L}_2$ répond à une interprétation physique naturelle. Si $\boldsymbol{T}$ est une transformation permettant de changer de paramétrisation, alors la réalisation qui minimise la $\mathcal{L}_2$-sensibilité est obtenue par une transformation $\boldsymbol{T}$ telle que $\boldsymbol{T}\boldsymbol{T}^\top$ est singulière. Ce problème peut se résoudre en utilisant un algorithme de type gradient. Pour plus d'informations sur la $\mathcal{L}_2$-*sensibilité*, se référer à [29, 99].

Dans [46], Hinamoto *et al.* introduisent une nouvelle mesure, appelée *erreur sur la fonction de transfert*, dans le cas où tous les coefficients quantifiés en virgule fixe ont la même position de la virgule. Hilaire généralise la mesure dans le cas où chaque coefficient à sa propre position de la virgule [40, 41]. Cette mesure, contrairement à la mesure de la $\mathcal{L}_2$-sensibilité, tient compte de la quantification des coefficients. En fait les deux mesures sont équivalentes dans le cas où les coefficients partagent la même position de la virgule, mais l'erreur sur la fonction de transfert reste pertinente quand chaque coefficient à sa propre position de la virgule, contrairement à la mesure de Gevers et Li. Une solution non-optimale du problème d'optimisation consistant à trouver la meilleure réalisation par rapport à l'erreur sur la fonction de transfert peut être obtenue en utilisant un algorithme de Recuit Simulé Adaptatif [49]. Cet algorithme converge vers un optimum global si on le laisse chercher indéfiniment, on peut donc obtenir une solution relativement proche de l'optimale sur un nombre fini d'itérations.

Les pôles d'une fonction de transfert peuvent également des indicateurs de la stabilité du système [66]. Ainsi, dans [41], une sensibilité par rapport aux pôles de la fonction de transfert d'un filtre est décrite. De plus, un compromis entre cette mesure et l'erreur sur la fonction de transfert est proposé dans cet article (équation (98) de [41]).

Ces mesures vont nous permettre, notamment dans la section suivante, de donner une idée de la *sensibilité* de la fonction de transfert à la quantification en virgule fixe.

1.3 Différentes réalisations

Nous avons déjà vu qu'un filtre LTI peut se caractériser par la relation temporelle qui donne la sortie en fonction de l'entrée, comme la relation de l'équation (1.14). Cette relation est appelée Forme Directe I du fait qu'elle reprend directement l'équation aux différences. Nous avons également vu la représentation d'état, faisant intervenir une variable conservant l'état du système d'une itération à l'autre, et qu'il y a, par un changement de variable $\boldsymbol{T}$, une infinité de réalisations en représentation d'état. Nous allons voir qu'il existe d'autres formes directes, mais également d'autres réalisations, d'un filtre LTI. L'intérêt de cette diversité des réalisations est que, si elles sont toutes équiva-

lentes en précision infinie, certaines sont mieux conditionnées que d'autres par rapport aux dégradations dues à l'implémentation en précision finie.

Le but n'est pas d'être exhaustif tant il y a de réalisations possibles pour un filtre donné, mais au moins de montrer les principales réalisations, et certaines plus méconnues (comme les structures LGS et LCW) auxquelles nous nous sommes intéressés dans nos travaux.

Pour plus de commodité de lecture, les formes directes sont présentées ici dans le cas SISO. La généralisation au cas MIMO est faisable mais alourdit les notations.

Graphe flot de données : En traitement du signal, les réalisations de filtres LTI sont souvent représentés par des *Graphes Flot de Données* (GFD). Toute réalisation de filtre LTI exprimée par un GFD nécessite uniquement trois types d'opérateurs :

- l'opérateur retard (ou délai) qui effectue un décalage temporel d'une unité sur un signal,
- l'opérateur d'addition qui effectue la somme des signaux entrants et retourne un signal,
- l'opérateur de multiplication par une constante.

La réciproque est également vraie, toute réalisation d'un filtre LTI s'exprime par un graphe composé uniquement de ces trois types d'opérateurs. Ces opérateurs sont représentés par la figure 1.2.

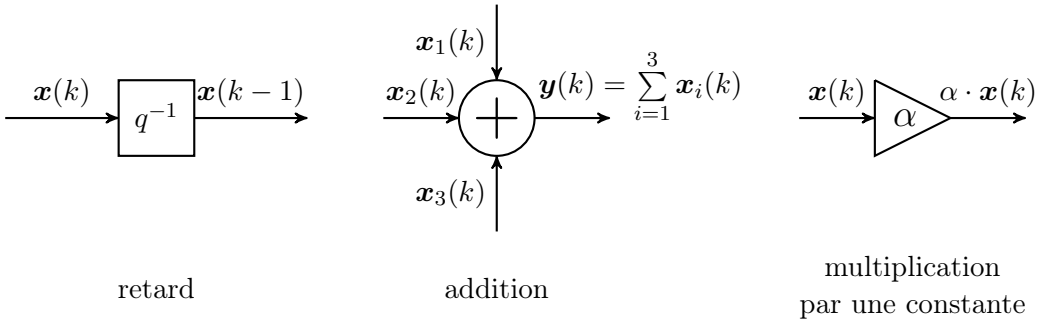


Figure 1.2 – Les trois types d'opérateurs d'un GFD d'un filtre LTI.

L'opérateur retard est représenté par la notation q^{-1} car q est la notation usuelle de l'opérateur *avance* défini par :

$$q[\boldsymbol{x}(k)] \triangleq \boldsymbol{x}(k+1). \tag{1.34}$$

Ainsi, l'opérateur retard q^{-1} effectue l'opération inverse, c'est-à-dire $q^{-1}[\boldsymbol{x}(k)] = \boldsymbol{x}(k-1)$.

La figure 1.3 illustre le graphe flot de données pour la représentation d'état.

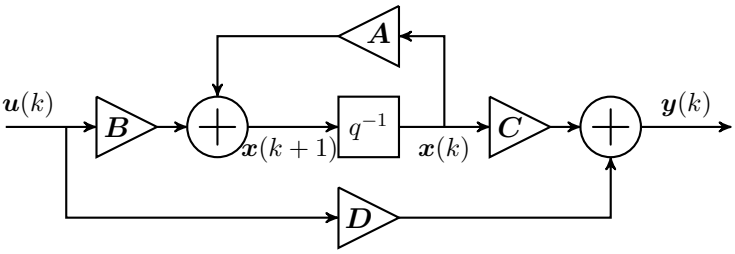


FIGURE 1.3 – Graphe flot de données pour la représentation d'état.

1.3.1 Formes Directes

Si la communauté de l'*automatique* utilise couramment la représentation d'état, les réalisations les plus couramment utilisées par la communauté du *signal* sont les formes directes. Leurs noms vient du fait que leurs coefficients sont directement ceux de la fonction de transfert définie par l'équation (1.17).

1.3.1.1 Forme Directe I

La forme directe I (DFI) est la forme directe la plus courante, car elle correspond tout simplement à l'équation aux différences, et est donnée par l'équation (1.14), rappelée ici :

$$y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=1}^{n} a_i y(k-i),$$

où n est l'ordre du filtre.

Bien qu'elle soit la plus utilisée, cette réalisation peut avoir une sensibilité élevée et nécessite de mémoriser à chaque instant $2n-1$ valeurs ($u(k-i)$ pour $0 \leqslant i \leqslant n$ et $y(k-i)$ pour $1 \leqslant i \leqslant n$) [29].

Le GFD de cette réalisation est illustré par la figure 1.4.

1.3.1.2 Forme Directe II

La forme directe II (DFII) permet d'écrire un filtre sous une forme plus compacte que la DFI en mutualisant les opérateurs retard, comme le montre le GFD de la réalisation dans la figure 1.5.

La DFII est décrite par l'algorithme suivant :

$$\begin{cases} e(k) &= u(k) - \sum_{i=1}^{n} a_i e(k-i) \\ y(k) &= \sum_{i=0}^{n} b_i e(k-i) \end{cases} \tag{1.35}$$

Dans cette réalisation, il n'y a plus que $n+1$ valeurs à mémoriser à l'instant k ($e(k-i)$ pour $0 \leqslant i \leqslant n$).

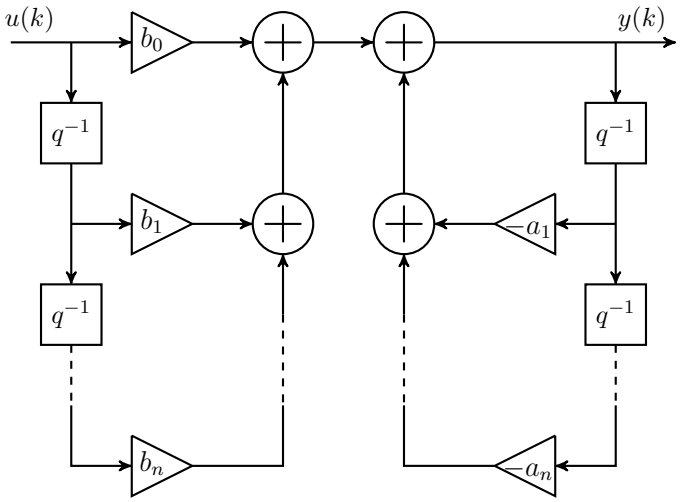


Figure 1.4 – Graphe flot de données pour la Forme Directe I.

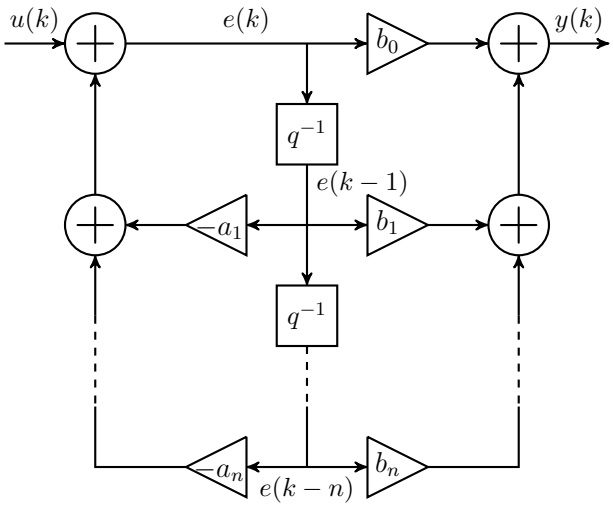


Figure 1.5 – Graphe flot de données pour la Forme Directe II.

1.3.1.3 Formes Directes Transposées

Il existe également, pour les les formes directes I et II, une forme transposée consistant à considérer le transposé du graphe flot de données[4] de la forme directe souhaitée. Par exemple, pour la DFII, le GFD de la forme directe II transposée (DFIIt) est illustré par la figure 1.6.

4. Le graphe transposé d'un GFD est obtenu en inversant le sens de chaque arête du graphe de départ et en échangeant l'entrée et la sortie. Ainsi, les nœuds *addition* deviennent des redistributions (des fourches) et les redistributions deviennent des nœuds *addition*.

Remarque 1.4. *Le GFD d'un graphe et son transposé définissent la même fonction de transfert. En effet, la* règle de Mason *[74] permet d'obtenir une fonction de transfert à partir d'un GFD, et on peut montrer que les fonctions de transfert issues d'un GFD et de son transposé sont les mêmes.*

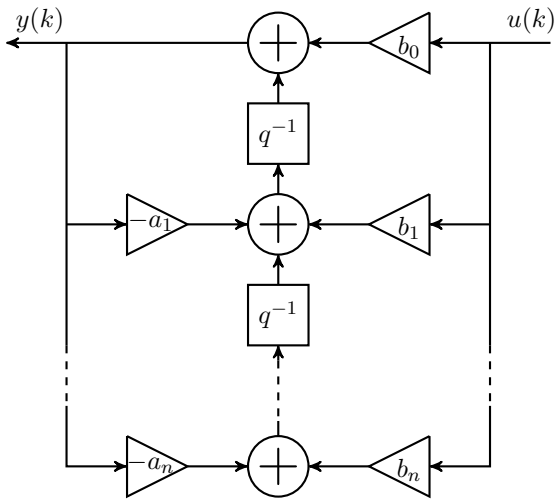


FIGURE 1.6 – Graphe flot de données pour la Forme Directe II transposée.

Le nombre de valeurs mémorisées et la sensibilité d'une forme directe et de sa transposée sont les mêmes, mais la propagation des erreurs de calcul est différente.

1.3.1.4 ρDFIIt

La forme directe II transposée avec opérateur ρ (ρDFIIt) est une modification de la DFIIt proposée par Li et Zhao [69, 108–110]. La modification consiste à remplacer l'opérateur délai q^{-1} par des opérateurs ρ_k^{-1}, définis par :

$$\rho_k(z) \triangleq \frac{z - \gamma_k}{\Delta_k}, \tag{1.36}$$

pour $1 \leqslant k \leqslant n$, où les coefficients γ_k et Δ_k sont choisis de manière à optimiser la sensibilité de la réalisation. On obtient ainsi une re-paramétrisation de la fonction de transfert sous la forme :

$$H(z) = \frac{\sum\limits_{i=0}^{n} \beta_i \varrho_i(z)}{1 + \sum\limits_{i=1}^{n} a_i \varrho_i}, \quad \forall z \in \mathbb{C}, \tag{1.37}$$

avec

$$\varrho_i(z) \triangleq \prod_{k=i+1}^{n} \rho_k(z), \quad \forall 0 \leqslant i \leqslant n. \tag{1.38}$$

Cette réalisation possède plus de coefficients que la DFI ($3n + 1$ contre $2n - 1$) mais a une $\mathcal{L}_2$-sensibilité bien meilleure que celle de la DFIIt comme l'ont montré Li et Zhao [69].

La figure 1.7 décrit un ρ_k^{-1} en tant que nœud d'un graphe flot de données.

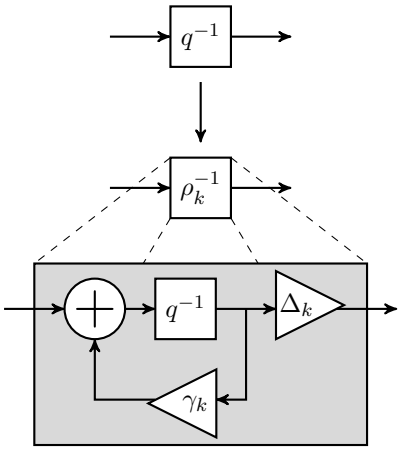


Figure 1.7 – La ρDFIIt par d'une DFIIt et change tous les q^{-1} en ρ_k^{-1}.

1.3.2 Structure LGS

Cette structure, proposée par Li, Gevers et Sun (LGS) [68], s'inspire d'une paramétrisation (on rappelle qu'une paramétrisation est l'ensemble des quatre matrices caractérisant un state-space donné) proposée par Johns, Snelgrove et Sedra (JSS) pour un filtre dans le domaine continu [52]. La paramétrisation JSS a l'avantage d'utiliser des matrices très creuses et donc de diminuer fortement le nombre de calculs (cette paramétrisation est décrite dans l'annexe A.1), mais possède également une sensibilité très faible par rapport à la quantification des coefficients. Ainsi, Li *et al.* ont eu l'idée de décrire la paramétrisation JSS discrète (paramétrisation DJSS) puisque celle-ci possède également une sensibilité très faible (en effet, il a été montré que si une paramétrisation avait une mesure de sensibilité faible dans le domaine continu, alors son équivalent dans le domaine discret avait aussi une mesure de sensibilité faible [73]). Cependant, le passage du domaine continu au domaine discret implique une inversion de matrice et donc une matrice creuse devient une matrice dense. Li *et al.* proposent alors la structure LGS, qui consiste à factoriser la matrice des coefficients en un produit de matrices creuses. Le terme paramétristation n'est pas applicable pour la réalisation proposée par Li *et al.* car elle n'a plus la forme d'un state-space.

La construction de cette structure, décrite dans [68], est détaillée dans

l'annexe A.2. On donne néanmoins ici la structure LGS :

$$\begin{cases} \boldsymbol{x}^{(0)}(k) &= \boldsymbol{x}(k) \\ \boldsymbol{x}^{(m)}(k) &= \boldsymbol{A}^{(m)}\boldsymbol{x}^{(m-1)}(k), \quad m = 1, 2, \ldots, N \\ \boldsymbol{x}(k+1) &= \boldsymbol{x}^{(N)}(k) + \boldsymbol{b}_{in}u(k) \\ y(k) &= \boldsymbol{c}_{in}\boldsymbol{x}(k) + du(k) \end{cases}$$

où les vecteurs $\boldsymbol{b}_{in}$ et $\boldsymbol{c}_{in}$ sont issus de la paramétrisation DJSS, les matrices $\boldsymbol{A}^{(m)}$ pour $1 \leqslant m \leqslant N$ sont issues de la factorisation de $\boldsymbol{A}_{in}$ (étant elle-même la matrice carrée dans la paramétrisation DJSS), les $\boldsymbol{x}^{(m)}(k)$ pour $1 \leqslant m \leqslant N$ sont des variables intermédiaires, et $N = 3(n-1)+1$ où n est l'ordre du filtre.

Cette structure possède une sensibilité $\mathcal{L}_2$ très faible (proche de celle de la paramétrisation DJSS), nécessite $7n-3$ multiplications et $6n-3$ additions (un state-space classique dense nécessite $(n+1)^2$ multiplications et $n(n+1)$ additions si tous les coefficients sont non nuls), et est caractérisée par $5(n-1)$ coefficients non triviaux ($(n+1)^2$ pour un state-space dense).

1.3.3 Structure LCW

Cette structure, proposée par Li, Chu et Wu (LCW) [67] est une amélioration de la structure LGS. En effet, cette structure se base elle aussi sur la paramétrisation DJSS et propose une autre factorisation de la matrice $\boldsymbol{A}_{in}$ (voir notations de la structure LGS). La description de la structure est donnée dans l'annexe A.3 et la structure est donnée par :

$$\begin{cases} \boldsymbol{x}^{(0)}(k) &= 2\boldsymbol{x}(k) \\ \boldsymbol{x}^{(m)}(k) &= \tilde{\boldsymbol{A}}_m\boldsymbol{x}^{(m-1)}(k), \\ \boldsymbol{x}(k+1) &= \boldsymbol{x}^{(N)}(k) - \boldsymbol{x}(k) + \tilde{\boldsymbol{b}}u(k), \\ y(k) &= \tilde{\boldsymbol{c}}\boldsymbol{x}(k) + du(k), \end{cases}$$

Les matrices $\tilde{\boldsymbol{A}}_m$ sont très creuses et il en est de même pour le vecteur $\tilde{\boldsymbol{c}}$.

La structure LCW offre une meilleure sensibilité que la structure LGS, et diminue encore le nombres d'opérations. En effet, cette structure nécessite $4n-1$ multiplications, contre $7n-3$ pour la structure LGS et $(n+1)^2$ pour un state-space pleinement paramétré, et $4n-1$ additions contre $6n-3$ pour la structure LGS et $n(n+1)$ pour un state-space pleinement paramétré.

1.3.4 Décompositions en sous-filtres

1.3.4.1 Structure en cascade

On dit d'un filtre qu'il a une *structure en cascade* s'il s'écrit comme une succession de sous-filtres, c'est-à-dire chaque sortie d'un sous-filtre est l'entrée de son successeur (sauf pour le dernier dont la sortie est la sortie du filtre complet). La figure 1.8 illustre une telle décomposition.

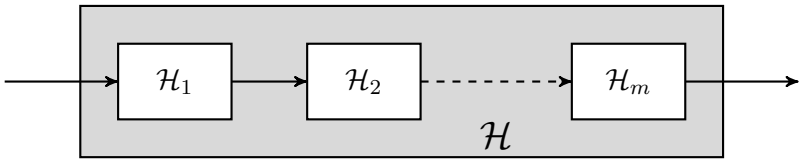


Figure 1.8 – Décomposition en cascade d'un filtre.

Pour déterminer une structure en cascade pour un filtre LTI, il suffit de factoriser la fonction de transfert H :

$$H(z) = \prod_{i=1}^{m} H_i(z) \quad z \in \mathbb{C}, \tag{1.39}$$

où H_i est la fonction de transfert d'un sous-filtre $\mathcal{H}_i$, pour $1 \leqslant i \leqslant m$. Chaque sous-filtre $\mathcal{H}_i$ peut être réalisé par n'importe quelle réalisation, indépendamment des autres. Classiquement le filtre $\mathcal{H}$ est décomposé en sous-filtres d'ordre 2.

Cette décomposition ne peut généralement pas être appliquée au cas MIMO.

1.3.4.2 Structure parallèle

Une *structure parallèle* pour la décomposition d'un filtre en sous-filtres est une décomposition telle que l'entrée de chaque sous-filtre est l'entrée du filtre complet et telle que la sortie du filtre complet est la somme des sortie de tous les sous-filtres (voir figure 1.9).

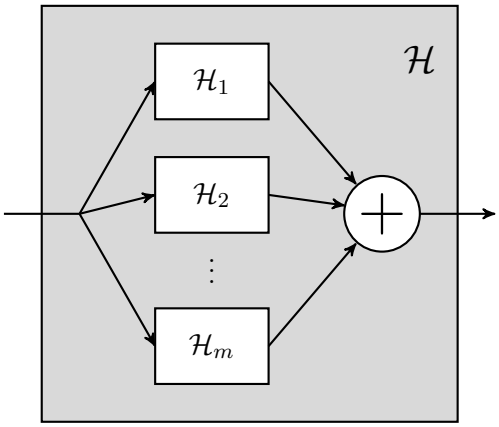


Figure 1.9 – Décomposition parallèle d'un filtre.

Déterminer une telle décomposition revient donc à décomposer la fonction de transfert H du filtre en somme de fractions rationnelles, généralement du second ordre.

Comme pour la structure en cascade, chaque sous-filtre peut être réalisé par n'importe quelle réalisation.

1.4 Forme implicite

La forme implicite (**S**pecialized **I**mplicit **F**orm, noté SIF), introduite dans [43] et décrite également dans [39, 44], est un formalisme sous forme de calcul matriciel permettant d'exprimer toutes les réalisations de filtres LTI. L'utilité d'une telle unification dans la représentation est de proposer un flot d'implémentations unifié et d'écrire, une fois pour toutes, les expressions des différentes mesures de dégradation, au lieu de devoir les redévelopper pour chaque nouvelle structure, comme c'était le cas jusqu'alors [67, 68, 110]. Par conséquent, l'étude de la SIF'importe qu'elle réalisation de la plus simple (FDI) au dans le cas général va nous permettre de traiter nx plus compliquées (LGS, LCW). Elle permet également d'expliciter tous les calculs, en décorrélant deux mêmes signaux à des instants différents (par exemple lorsque le calcul de y à l'instant k dépend de y aux instants $k-i$). C'est-à-dire en séparant, d'un côté, les calculs faits à un instant donné et, d'un autre côté, la propagation de ces calculs.

Cette structure est un state-space implicite [65] pour lequel le terme de gauche (voir équation (1.40)) a une forme particulière.

Comme dans un state-space, les variables $\boldsymbol{x}_i$ sont des variables d'état, tandis que les variables $\boldsymbol{t}_i$ sont des variables intermédiaires.

La SIF s'exprime ainsi :

$$\begin{pmatrix} \boldsymbol{J} & \boldsymbol{0} & \boldsymbol{0} \\ -\boldsymbol{K} & \boldsymbol{I}_{n_x} & \boldsymbol{0} \\ -\boldsymbol{L} & \boldsymbol{0} & \boldsymbol{I}_{n_y} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k+1) \\ \boldsymbol{x}(k+1) \\ \boldsymbol{y}(k) \end{pmatrix} = \begin{pmatrix} \boldsymbol{0} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{0} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{0} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k) \\ \boldsymbol{x}(k) \\ \boldsymbol{u}(k) \end{pmatrix} \tag{1.40}$$

On notera alors n_t, n_x, n_y et n_u, les tailles des vecteurs $\boldsymbol{t}$, $\boldsymbol{x}$, $\boldsymbol{y}$ et $\boldsymbol{u}$, respectivement. La matrice $\boldsymbol{J}$ est une matrice triangulaire inférieure avec des 1 sur sa diagonale. On a les dimensions suivantes pour les différentes matrices de coefficients :

$$\begin{array}{llllllll} \boldsymbol{J} & \in & \mathbb{R}^{n_t \times n_t}, & \boldsymbol{M} & \in & \mathbb{R}^{n_t \times n_x}, & \boldsymbol{N} & \in & \mathbb{R}^{n_t \times n_u}, \\ \boldsymbol{K} & \in & \mathbb{R}^{n_x \times n_t}, & \boldsymbol{P} & \in & \mathbb{R}^{n_x \times n_x}, & \boldsymbol{Q} & \in & \mathbb{R}^{n_x \times n_u}, \\ \boldsymbol{L} & \in & \mathbb{R}^{n_y \times n_t}, & \boldsymbol{R} & \in & \mathbb{R}^{n_y \times n_x}, & \boldsymbol{S} & \in & \mathbb{R}^{n_y \times n_u}. \end{array} \tag{1.41}$$

Exemple 1.1. *On considère un exemple, avec des notations simplifiées, où on veut exprimer le calcul* $\boldsymbol{y}(k) = \boldsymbol{M}_1 \cdot \boldsymbol{M}_2 \cdot \boldsymbol{x}(k)$ *sous la forme* $\boldsymbol{y}(k) = \boldsymbol{M}_1 \cdot (\boldsymbol{M}_2 \cdot \boldsymbol{x}(k))$ *(ce qui peut être intéressant par exemple si les matrices* $\boldsymbol{M}_1$ *et* $\boldsymbol{M}_2$ *sont creuses mais que le produit* $\boldsymbol{M}_1 \cdot \boldsymbol{M}_2$ *ne l'est pas).*

On utilise alors la variable intermédiaire $\boldsymbol{t}$ *pour décomposer le calcul :*

$$\begin{aligned} \boldsymbol{t}(k+1) &= \boldsymbol{M}_2 \cdot \boldsymbol{x}(k) \\ \boldsymbol{y}(k) &= \boldsymbol{M}_1 \cdot \boldsymbol{t}(k+1) \end{aligned}$$

que l'on peut aussi écrire :

$$\begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} \\ -\boldsymbol{M}_1 & \boldsymbol{I} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k+1) \\ \boldsymbol{y}(k) \end{pmatrix} = \begin{pmatrix} \boldsymbol{M}_2 \\ \boldsymbol{0} \end{pmatrix} \cdot \boldsymbol{x}.$$

Les valeurs du vecteurs de variables intermédiaires $\boldsymbol{t}(k+1)$ sont calculées et utilisées dans la même itération, et ne sont donc pas stockées, ce qui se caractérise par la colonne de 0 dans la matrice suivante :

$$\begin{pmatrix} \boldsymbol{0} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{0} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{0} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix}. \tag{1.42}$$

On notera également $\boldsymbol{Z}$ la matrice de tous les coefficients de la SIF, de la façon suivante :

$$\boldsymbol{Z} \triangleq \begin{pmatrix} -\boldsymbol{J} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} \tag{1.43}$$

Dans cette matrice on utilise $-\boldsymbol{J}$ plutôt que $\boldsymbol{J}$ pour des raisons de simplicité dans les calculs ultérieurs [39, 44].

La fonction de transfert $\boldsymbol{H}$ du système décrit par la SIF caractérisée par la matrice $\boldsymbol{Z}$ est donnée par :

$$\boldsymbol{H} : z \mapsto \boldsymbol{C_Z}(z\boldsymbol{I}_n - \boldsymbol{A_Z})^{-1}\boldsymbol{B_Z} + \boldsymbol{D_Z}, \quad \forall z \in \mathbb{C} \tag{1.44}$$

avec

$$\begin{aligned} \boldsymbol{A_Z} &= \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{P}, & \boldsymbol{B_Z} &= \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{Q}, \\ \boldsymbol{C_Z} &= \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{R}, & \boldsymbol{D_Z} &= \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{S}. \end{aligned} \tag{1.45}$$

On a alors

$$\begin{aligned} \boldsymbol{A_Z} &\in \mathbb{R}^{n_x \times n_x}, & \boldsymbol{B_Z} &\in \mathbb{R}^{n_x \times n_u}, \\ \boldsymbol{C_Z} &\in \mathbb{R}^{n_y \times n_x}, & \boldsymbol{D_Z} &\in \mathbb{R}^{n_y \times n_u}. \end{aligned} \tag{1.46}$$

À la SIF correspond alors l'algorithme de calcul suivant (en effectuant simplement le produit matrice/vecteur de l'équation (1.40)) :

$$\begin{aligned} \boldsymbol{J}\boldsymbol{t}(k+1) &\leftarrow \boldsymbol{M}\boldsymbol{x}(k) + \boldsymbol{N}\boldsymbol{u}(k) \\ \boldsymbol{x}(k+1) &\leftarrow \boldsymbol{K}\boldsymbol{t}(k+1) + \boldsymbol{P}\boldsymbol{x}(k) + \boldsymbol{Q}\boldsymbol{u}(k) \\ \boldsymbol{y}(k) &\leftarrow \boldsymbol{L}\boldsymbol{t}(k+1) + \boldsymbol{R}\boldsymbol{x}(k) + \boldsymbol{S}\boldsymbol{u}(k) \end{aligned}$$

Complètement développé, l'algorithme précédent devient :

$$\boldsymbol{t}_i(k+1) \leftarrow -\sum_{j<i} \boldsymbol{J}_{ij}\boldsymbol{t}_j(k+1) + \sum_{j=1}^{n_x} \boldsymbol{M}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{N}_{ij}\boldsymbol{u}_j(k) \quad \forall 1 \leqslant i \leqslant n_t \tag{1.47a}$$

$$\boldsymbol{x}_i(k+1) \leftarrow \sum_{j=1}^{n_t} \boldsymbol{K}_{ij}\boldsymbol{t}_j(k+1) + \sum_{j=1}^{n_x} \boldsymbol{P}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{Q}_{ij}\boldsymbol{u}_j(k) \quad \forall 1 \leqslant i \leqslant n_x \tag{1.47b}$$

$$\boldsymbol{y}_i(k) \leftarrow \sum_{j=1}^{n_t} \boldsymbol{L}_{ij}\boldsymbol{t}_j(k+1) + \sum_{j=1}^{n_x} \boldsymbol{R}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{S}_{ij}\boldsymbol{u}_j(k) \quad \forall 1 \leqslant i \leqslant n_y \tag{1.47c}$$

Remarque 1.5. *Dans les faits, les matrices $\boldsymbol{Z}$ des SIF sont généralement composées en majorité de 0 et de 1, donc les calculs sont plus simples que ce que les équations (1.47) laissent penser.*

Il est important de noter ce qu'implique la forme particulière de la matrice $\boldsymbol{J}$. En effet, $\boldsymbol{t}_i$, pour $1 \leqslant i \leqslant n_t$, est calculé à partir des $\boldsymbol{t}_j$ pour $1 \leqslant j < i$ (et des vecteurs de variables $\boldsymbol{x}$ et $\boldsymbol{u}$), on doit alors calculer dans l'ordre, $\boldsymbol{t}_1$, puis $\boldsymbol{t}_2$ à partir de $\boldsymbol{t}_1$, etc. En considérant cette information, on peut écrire l'algorithme de la façon suivante :

$$\boldsymbol{t}_i(k+1) = -\boldsymbol{J}'_i\boldsymbol{t}(k+1) + \boldsymbol{M}_i\boldsymbol{x}(k) + \boldsymbol{N}_i\boldsymbol{u}(k) \quad \forall 1 \leqslant i \leqslant n_t \tag{1.48a}$$

$$\boldsymbol{x}_i(k+1) = \boldsymbol{K}_i\boldsymbol{t}(k+1) + \boldsymbol{P}_i\boldsymbol{x}(k) + \boldsymbol{Q}_i\boldsymbol{u}(k) \quad \forall 1 \leqslant i \leqslant n_x \tag{1.48b}$$

$$\boldsymbol{y}_i(k) = \boldsymbol{L}_i\boldsymbol{t}(k+1) + \boldsymbol{R}_i\boldsymbol{x}(k) + \boldsymbol{S}_i\boldsymbol{u}(k) \quad \forall 1 \leqslant i \leqslant n_y \tag{1.48c}$$

où $\boldsymbol{J}' \triangleq \boldsymbol{J} - \boldsymbol{I}_{n_t}$ et $\boldsymbol{C}_i$ est la i-ème ligne de la matrice $\boldsymbol{C}$ pour

$$\boldsymbol{C} \in \{\boldsymbol{J}', \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{K}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{L}, \boldsymbol{R}, \boldsymbol{S}\}.$$

Cette réécriture nous permet d'exprimer la SIF sous forme d'une liste de sommes de produits (voir chapitre 3).

1.4.1 Exemple : Structures LGS et LCW

Structure LGS : On se propose ici de décrire la structure LGS sous forme d'une SIF. On rappelle pour cela l'expression de la structure LGS :

$$\begin{cases} \boldsymbol{x}^{(0)}(k) &= \boldsymbol{x}(k) \\ \boldsymbol{x}^{(m)}(k) &= \boldsymbol{A}^{(m)}\boldsymbol{x}^{(m-1)}(k), \quad m = 1, 2, \ldots, N \\ \boldsymbol{x}(k+1) &= \boldsymbol{x}^{(N)}(k) + \boldsymbol{b}_{in}u(k) \\ y(k) &= \boldsymbol{c}_{in}\boldsymbol{x}(k) + du(k) \end{cases}$$

dont les notations sont décrites dans l'annexe A.2.

Il faut bien remarquer ici que le produit des variables intermédiaires $\boldsymbol{x}^{(m)}$ se calcule dans un ordre précis. En effet, puisque, selon les notations utilisées dans la description de la structure, $\boldsymbol{A}_{in} = \prod_{m=1}^{N} \boldsymbol{A}^{(m)} = \boldsymbol{A}^{(N)}\boldsymbol{A}^{(N-1)}\dots\boldsymbol{A}^{(2)}\boldsymbol{A}^{(1)}$, alors le produit des $\boldsymbol{x}^{(m)}$ se calcule selon l'ordre suivant :

$$\begin{aligned}
\boldsymbol{x}^{(0)}(k) &= \boldsymbol{x}(k), && (1.49)\\
\boldsymbol{x}^{(1)}(k) &= \boldsymbol{A}^{(1)}\boldsymbol{x}^{(0)}(k), && (1.50)\\
\dots & && (1.51)\\
\boldsymbol{x}^{(N)}(k) &= \boldsymbol{A}^{(N)}\boldsymbol{x}^{(N-1)}(k). && (1.52)
\end{aligned}$$

On a alors la même notion d'ordre que dans le cas de la SIF avec le calcul du vecteur de variables intermédiaires $\boldsymbol{t}$. La correspondance se fait donc naturellement entre les variables $\boldsymbol{x}^{(m)}$ et les variables $\boldsymbol{t}_i$ (en supprimant le terme initial $\boldsymbol{x}^{(0)}(k)$) :

$$\begin{cases}
\boldsymbol{t}_1(k+1) &= \boldsymbol{A}^{(1)}\boldsymbol{x}(k)\\
\boldsymbol{t}_2(k+1) &= \boldsymbol{A}^{(2)}\boldsymbol{t}_1(k+1)\\
& \vdots\\
\boldsymbol{t}_N(k+1) &= \boldsymbol{A}^{(N)}\boldsymbol{t}_{N-1}(k+1)
\end{cases}$$

Remarque 1.6. *On pourra noter qu'ici un élément $\boldsymbol{t}_i$, avec $1 \leqslant i \leqslant N$, est un vecteur de taille n (l'ordre du filtre, on rappelle qu'on a $N = 3(n-1)+1$), alors que dans la description générale $\boldsymbol{t}_i$ est un scalaire, même si cela ne change rien à la mise en œuvre de la SIF.*

On a donc, pour tout $2 \leqslant i \leqslant N$:

$$\boldsymbol{t}_i(k+1) = -\sum_{j<i} \boldsymbol{J}_{ij}\boldsymbol{t}_j(k+1) \quad \text{avec} \quad \begin{cases} \boldsymbol{J}_{ij} = \boldsymbol{0}_n & \text{pour} \quad j < i-1,\\ \boldsymbol{J}_{ij} = -\boldsymbol{A}^{(i)} & \text{pour} \quad j = i-1, \end{cases} \tag{1.53}$$

où $\boldsymbol{0}_n$ désigne la matrice carrée composée uniquement de 0. L'équation (1.53), par analogie avec l'équation (1.47a), donne la matrice $\boldsymbol{J}$ suivante :

$$\boldsymbol{J} \triangleq \begin{pmatrix} \boldsymbol{I}_n & & & \text{\Large 0} \\ -\boldsymbol{A}^{(2)} & \boldsymbol{I}_n & & \\ & \ddots & \ddots & \\ \text{\Large 0} & & -\boldsymbol{A}^{(N)} & \boldsymbol{I}_n \end{pmatrix}. \tag{1.54}$$

De plus, on a $\boldsymbol{t}_1(k+1) = \boldsymbol{A}^{(1)}\boldsymbol{x}(k)$, dont on peut déduire, toujours par analogie avec l'équation (1.47a) la matrice $\boldsymbol{M}$ et le vecteur $\boldsymbol{n}$ (puisque $u(k)$ n'intervient

pas dans le calcul des $\boldsymbol{t}_i$) :

$$\boldsymbol{M} \triangleq \begin{pmatrix} \boldsymbol{A}^{(1)} \\ \boldsymbol{0} \\ \vdots \\ \boldsymbol{0} \end{pmatrix}, \quad \boldsymbol{n} \triangleq \boldsymbol{0}_{1\times nN}, \tag{1.55}$$

où nN correspond à la taille du vecteur $\boldsymbol{t}$ ($\boldsymbol{t}_i$ est un vecteur de taille n pour tout $1 \leqslant i \leqslant N$).

Le reste de la structure s'écrit alors :

$$\boldsymbol{x}(k+1) = \boldsymbol{t}_N(k+1) + \boldsymbol{b}_{in}u(k) \tag{1.56}$$

$$y(k) = \boldsymbol{c}_{in}\boldsymbol{x}(k) + du(k) \tag{1.57}$$

Le calcul de $\boldsymbol{x}(k+1)$ dépend uniquement de la variable $\boldsymbol{t}_N$ et de l'entrée $u(k)$, pas des $\boldsymbol{t}_i$ pour $i < N$ ni de l'état $\boldsymbol{x}(k)$, on peut donc en déduire les matrices $\boldsymbol{K}$ et $\boldsymbol{P}$ et le vecteur $\boldsymbol{Q}$:

$$\boldsymbol{K} \triangleq \begin{pmatrix} 0 & \dots & 0 & \boldsymbol{I}_{n\times n} \end{pmatrix}, \quad \boldsymbol{P} \triangleq \boldsymbol{0}_{n\times n}, \quad \text{et} \quad \boldsymbol{Q} \triangleq \boldsymbol{b}_{in}. \tag{1.58}$$

Le calcul de la sortie $y(k)$ ne dépend pas des variables intermédiaires $\boldsymbol{t}_i$ pour $1 \leqslant i \leqslant N$, on peut donc directement déduire $\boldsymbol{L}$ et $\boldsymbol{R}$ et $\boldsymbol{S}$:

$$\boldsymbol{L} \triangleq \boldsymbol{0}_{nN\times 1}, \quad \boldsymbol{R} \triangleq \boldsymbol{c}_{in}, \quad \text{et} \quad \boldsymbol{S} \triangleq d. \tag{1.59}$$

Finalement, on a la matrice des coefficients $\boldsymbol{Z}$:

$$\boldsymbol{Z} = \begin{pmatrix} -\boldsymbol{J} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} = \left(\begin{array}{cccc:c:c} -\boldsymbol{I}_n & & & \bigcirc & \boldsymbol{A}^{(1)} & 0 \\ \boldsymbol{A}^{(2)} & -\boldsymbol{I}_n & & & \boldsymbol{0} & 0 \\ & \ddots & \ddots & & \vdots & \vdots \\ \bigcirc & & \boldsymbol{A}^{(N)} & -\boldsymbol{I}_n & \boldsymbol{0} & 0 \\ \hdashline \boldsymbol{0} & \dots & \boldsymbol{0} & \boldsymbol{I}_n & \boldsymbol{0} & \boldsymbol{b}_{in} \\ \hdashline 0 & \dots & \dots & 0 & \boldsymbol{c}_{in} & d \end{array}\right). \tag{1.60}$$

Les tailles des vecteurs $\boldsymbol{t}$ et $\boldsymbol{x}$ sont respectivement $n_t = nN = n(3n-2)$ et $n_x = n$, avec n l'ordre du filtre, et puisqu'on est dans le cas SISO, on a $n_u = n_y = 1$.

Structure LCW : Exprimons maintenant la structure LCW sous forme de SIF. La structure LCW est donnée par :

$$\begin{cases} \boldsymbol{x}^{(0)}(k) & = 2\boldsymbol{x}(k) \\ \boldsymbol{x}^{(m)}(k) & = \tilde{\boldsymbol{A}}_m \boldsymbol{x}^{(m-1)}(k), \\ \boldsymbol{x}(k+1) & = \boldsymbol{x}^{(N)}(k) - \boldsymbol{x}(k) + \tilde{\boldsymbol{b}}u(k), \\ y(k) & = \tilde{\boldsymbol{c}}\boldsymbol{x}(k) + du(k), \end{cases}$$

avec les notations décrites dans l'annexe A.3. Attention, ici on a $N = 3(n-1)$, contrairement au $3(n-1)+1$ de la structure LGS, car la factorisation utilisée pour la structure LCW possède une matrice en moins que celle de la structure LGS.

On peut alors, de la même façon que pour la structure LGS, trouver une ressemblance entre les $\boldsymbol{t}_i$ impliqués dans la SIF et les $\boldsymbol{x}^{(m)}(k)$ impliqués dans la structure LCW. On pose alors :

$$\begin{cases} \boldsymbol{t}_1(k+1) &= 2\tilde{\boldsymbol{A}}_1\boldsymbol{x}(k) \\ \boldsymbol{t}_2(k+1) &= \tilde{\boldsymbol{A}}_2\boldsymbol{t}_1(k+1) \\ & \vdots \\ \boldsymbol{t}_N(k+1) &= \tilde{\boldsymbol{A}}_N\boldsymbol{t}_{N-1}(k+1) \end{cases}$$

On peut alors réécrire l'équation (1.53) dans le cas de la structure LCW, c'est-à-dire en changeant uniquement $-\boldsymbol{A}^{(i)}$ par $-\tilde{\boldsymbol{A}}_i$. On en déduit la matrice $\boldsymbol{J}$ suivante :

$$\boldsymbol{J} \triangleq \begin{pmatrix} \boldsymbol{I}_n & & & \text{\Large 0} \\ -\tilde{\boldsymbol{A}}_2 & \boldsymbol{I}_n & & \\ & \ddots & \ddots & \\ \text{\Large 0} & & -\tilde{\boldsymbol{A}}_N & \boldsymbol{I}_n \end{pmatrix}. \tag{1.61}$$

De plus, toujours par analogie avec la construction de la SIF pour la structure LGS, on peut déduire les matrices $\boldsymbol{M}$ et le vecteur $\boldsymbol{N}$:

$$\boldsymbol{M} \triangleq \begin{pmatrix} 2\tilde{\boldsymbol{A}}_1 \\ \boldsymbol{0} \\ \vdots \\ \boldsymbol{0} \end{pmatrix}, \quad \boldsymbol{N} \triangleq \boldsymbol{0}_{1\times nN}, \tag{1.62}$$

où de la même façon nN correspond à la taille du vecteur $\boldsymbol{t}$ ($\boldsymbol{t}_i$ est un vecteur de taille n pour tout $1 \leqslant i \leqslant N$).

Le reste de la structure s'écrit :

$$\boldsymbol{x}(k+1) = \boldsymbol{t}_N(k) - \boldsymbol{x}(k) + \tilde{\boldsymbol{b}}u(k) \tag{1.63}$$

$$y(k) = \tilde{\boldsymbol{c}}\boldsymbol{x}(k) + du(k) \tag{1.64}$$

Le calcul de $\boldsymbol{x}(k+1)$ dépend uniquement de la variable $\boldsymbol{t}_N$, de l'état $\boldsymbol{x}(k)$ et de l'entrée $u(k)$, pas des $\boldsymbol{t}_i$ pour $i < N$, on peut donc en déduire les matrices $\boldsymbol{K}$ et $\boldsymbol{P}$ et le vecteur $\boldsymbol{Q}$:

$$\boldsymbol{K} \triangleq \begin{pmatrix} 0 & \dots & 0 & \boldsymbol{I}_{n\times n} \end{pmatrix}, \quad \boldsymbol{P} \triangleq -\boldsymbol{I}_{n\times n}, \quad \text{et} \quad \boldsymbol{Q} \triangleq \tilde{\boldsymbol{b}}. \tag{1.65}$$

Le calcul de la sortie $y(k)$ ne dépend pas, comme pour la structure LGS, des variables intermédiaires $\boldsymbol{t}_i$ pour $1 \leqslant i \leqslant N$, on peut donc directement déduire $\boldsymbol{L}$, $\boldsymbol{R}$ et $\boldsymbol{S}$:

$$\boldsymbol{L} \triangleq \boldsymbol{0}_{nN\times 1}, \quad \boldsymbol{R} \triangleq \tilde{\boldsymbol{c}}, \quad \text{et} \quad \boldsymbol{S} \triangleq d. \tag{1.66}$$

Finalement, on a la matrice des coefficients $\boldsymbol{Z}$:

$$\boldsymbol{Z} = \begin{pmatrix} -\boldsymbol{J} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} = \left(\begin{array}{cccc:c:c} -\boldsymbol{I}_n & & & \text{\Large 0} & 2\tilde{\boldsymbol{A}}_1 & 0 \\ \tilde{\boldsymbol{A}}_2 & -\boldsymbol{I}_n & & & \boldsymbol{0} & 0 \\ & \ddots & \ddots & & \vdots & \vdots \\ \text{\Large 0} & & \tilde{\boldsymbol{A}}_N & -\boldsymbol{I}_n & \boldsymbol{0} & 0 \\ \hdashline \boldsymbol{0} & \dots & \boldsymbol{0} & \boldsymbol{I}_n & -\boldsymbol{I}_n & \tilde{\boldsymbol{b}} \\ \hdashline 0 & \dots & \dots & 0 & \tilde{\boldsymbol{c}} & d \end{array}\right). \tag{1.67}$$

Les tailles des vecteurs $\boldsymbol{t}$ et $\boldsymbol{x}$ sont respectivement $n_t = nN = 3n(n-1)$ et $n_x = n$, avec n l'ordre du filtre, et puisqu'on est dans le cas SISO, on a $n_u = n_y = 1$.

1.5 Conclusion

Dans ce premier chapitre, nous avons introduit les bases de traitement du signal nécessaires à notre approche. Après avoir rappelé les définitions d'un signal et d'un filtre linéaire invariant dans le temps, nous avons vu quelques propriétés de cette classe de filtre, comme le calcul des normes et la représentation de ces filtres par différentes réalisations.

La forme implicite SIF permet de représenter toutes ces réalisations sous un format unique. Une telle écriture nous permettra, par la suite (chapitre 3), de décrire notre approche une seule fois en la rendant applicable à toutes les réalisations.

Le prochain chapitre introduit l'arithmétique virgule fixe et propose un formalisme des opérations usuelles telles que la conversion, l'addition, la multiplication par une constante, et la quantification.

Chapitre 2

Arithmétique virgule fixe

"*La virgule flottante, c'est pour les fainéants,*
parce que c'est le FPU qui fait tout le boulot."
- Thibault Hilaire

L'objectif de ce chapitre est de présenter la notion d'arithmétique virgule fixe (FxP, pour **Fix**ed-**P**oint) en dressant un état de l'art rapide des différentes façons de représenter des nombres (entiers et réels) en informatique.

Après l'introduction des nombres FxP, nous formaliserons la conversion d'un nombre réel en un nombre FxP ainsi que les opérations usuelles en virgule fixe, à savoir les opérations de formatages sur un format connu, l'addition et la multiplication. Cette formalisation constitue à elle seule une première contribution. En effet, il est difficile de trouver un formalisme pleinement détaillé de l'arithmétique virgule fixe dans la littérature, chacun ayant sa propre représentation, qui n'est, à notre connaissance, détaillée nulle part. Dans [26], les auteurs présentent un nombre virgule fixe comme une simple extension des entiers et se limitent par conséquent à l'arithmétique des nombres entiers, sans préciser l'enjeu et la particularité du facteur d'échelle dans l'arithmétique virgule fixe. Dans [77], un début d'arithmétique virgule fixe est décrite mais est très orientée FPGA, or notre but est de décrire l'arithmétique indépendamment d'une cible particulière. Il était également important dans notre approche de prendre en compte des cas particuliers, qui ont une infime probabilité de se produire mais qu'il est nécessaire de considérer pour proposer un formalisme le plus complet possible.

Enfin, l'impact numérique des conversions et des calculs sera discuté à la fin de ce chapitre.

2.1 Représentations binaires des nombres

En machine, que ce soit sur un ordinateur généraliste ou sur des systèmes embarqués, les nombres sont généralement représentés en base 2 (représenta-

tion binaire), par des mots finis composés de chiffres valant 0 ou 1 appelés *bits*. Cependant, il existe des représentations décimales des nombres, notamment une représentation décimale pour les nombres flottants dans le standard IEEE754 [48] (il est d'ailleurs possible de convertir un flottant en représentation décimale en un flottant en représentation binaire et vice-versa [57]) ou le *Binary-Coded Decimal* qui utilise la représentation binaire pour représenter les chiffres de 0 à 9 [4], mais les représentations binaires sont généralement préférées, ne serait-ce que pour la plus grande simplicité des calculs.

Dans cette partie nous introduirons différentes représentations des nombres et ce en deux groupes, tout d'abord les nombres entiers relatifs et ensuite les nombres réels.

2.1.1 Nombres entiers

Il existe de nombreuses représentations binaires des nombres entiers relatifs [8, 79] mais nous ne montrerons ici que trois représentations, la représentation Signe - Valeur Absolue (SVA) et les représentations en complément à un et à deux. Dans le cas des nombres entiers, nous considérerons la représentation binaire sur w bits, allant du bit de poids 1 (position 0) jusqu'au bit le plus significatif de poids 2^{w-1} (position $w-1$).

La représentation binaire usuelle d'un nombre entier $X \in \mathbb{N}$ consiste à décomposer X en base 2 :

$$X = \sum_{i=0}^{w-1} x_i 2^i, \tag{2.1}$$

où les $(x_i)_{0 \leqslant i \leqslant w-1}$ sont les bits de la décomposition binaire de X. En effet, X en base 2 s'écrit comme une séquence ordonnée des $(x_i)_{0 \leqslant i \leqslant w-1}$ (voir figure 2.1). Cette représentation, non signée (aucun bit n'est alloué spécifiquement au signe du nombre X), permet de représenter les entiers de l'ensemble $[0; 2^w - 1]$.

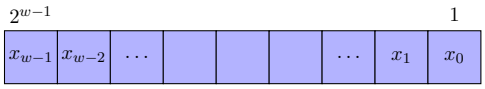


Figure 2.1 – Décomposition binaire d'un entier naturel X.

Intéressons-nous maintenant aux représentations binaires *signées* des entiers relatifs.

2.1.1.1 Représentation Signe - Valeur Absolue

Cette représentation est constituée d'un bit de signe situé sur le bit de poids fort, et d'une mantisse de $w-1$ bits correspondant à la valeur absolue du nombre. En fait cette représentation peut être vue comme une extension de la représentation binaire usuelle pour les nombres entiers naturels, à laquelle

on ajoute un bit de signe pour préciser si le nombre est positif (bit de poids fort égal à 0) ou négatif (bit de poids fort égal à 1). La représentation est symétrique et comporte donc deux écritures pour 0, l'une positive et l'autre négative.

Un nombre entier $X \in \mathbb{Z}$ s'écrira dans cette représentation, sur w bits :

$$X = (-1)^{x_{w-1}} \times \sum_{i=0}^{w-2} x_i 2^i. \tag{2.2}$$

De plus, l'ensemble des nombres représentables sur w bits est

$$[-2^{w-1} + 1; 2^{w-1} - 1] \cap \mathbb{Z}.$$

La Figure 2.2 illustre cette représentation pour l'ensemble des nombres entiers utilisant 4 bits, *i.e.* les nombres dans l'ensemble $[-7; 7] \cap \mathbb{Z}$.

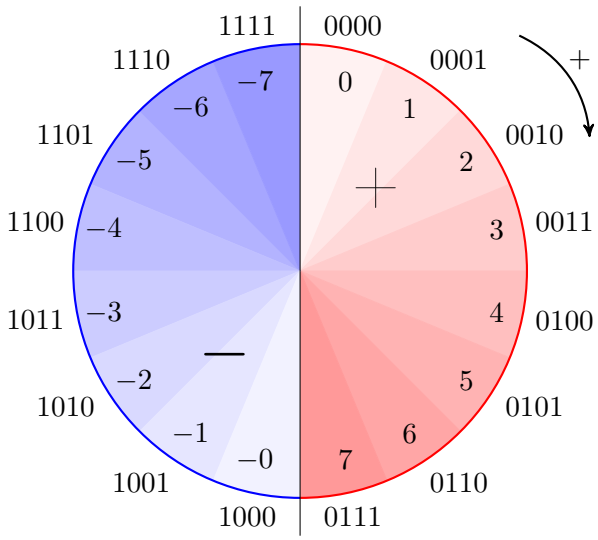


FIGURE 2.2 – Représentation Signe-Valeur Absolue des entiers de 4 bits

Comme on peut le remarquer dans (2.2), l'opposé d'un nombre s'obtient en changeant uniquement le bit de signe. Ceci est dû à la double représentation du zéro, ce qui rend la représentation symétrique.

2.1.1.2 Représentation en complément à un

Les représentations en complément sont basées sur les représentations dites biaisées [1] appliquées uniquement aux nombres négatifs. La représentation en

1. Les représentations biaisées utilisent un nombre entier R, que l'on ajoute au nombre $X \in \mathbb{Z}$ que l'on souhaite représenter, de sorte que $X + R$ soit toujours positif. Généralement R est choisi comme l'opposé du plus petit nombre négatif à représenter (voir [8]).

complément à un permet de représenter les mêmes valeurs que la représentation SVA pour une longueur de mots binaires donnée.

Un entier $X \in \mathbb{Z}$ s'écrira dans cette représentation, sur w bits :

$$X = -x_{w-1} \cdot (2^{w-1} - 1) + \sum_{i=0}^{w-2} x_i 2^i. \tag{2.3}$$

L'opposé d'un nombre en représentation en complément à un s'obtient en inversant chacun des bits de la représentation binaire, comme on peut le voir sur la Figure 2.3, illustrant la représentation pour l'ensemble des nombres entiers utilisant 4 bits (comme précisé plus haut, l'ensemble des valeurs représentables est le même que pour la représentation SVA).

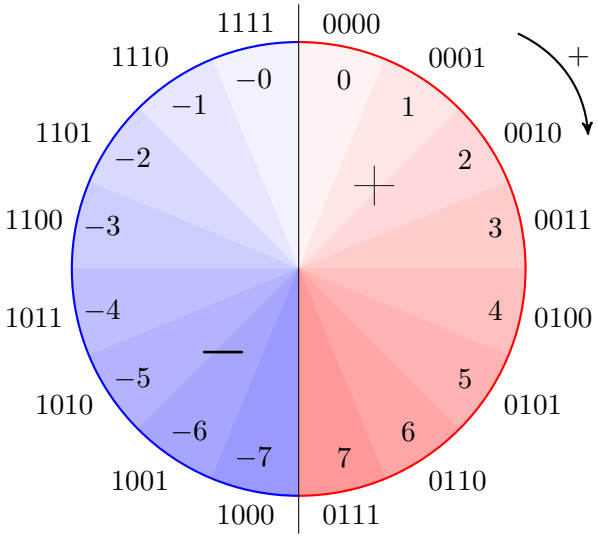


Figure 2.3 – Représentation en complément à un des entiers de 4 bits

2.1.1.3 Représentation en complément à deux

La dernière représentation discutée ici, dite en complément à deux, diffère des deux précédentes représentations par son unique représentation du 0.

En effet, un entier $X \in \mathbb{Z}$ s'écrira dans cette représentation, sur w bits :

$$X = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i. \tag{2.4}$$

De plus, l'ensemble des nombres représentables sur w bits est

$$[-2^{w-1}; 2^{w-1} - 1] \cap \mathbb{Z}.$$

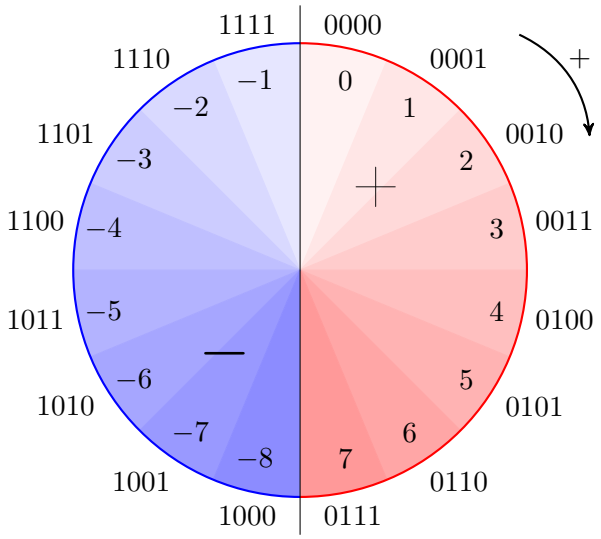


FIGURE 2.4 – Représentation en complément à deux des entiers de 4 bits

La Figure 2.4 illustre cette représentation pour l'ensemble des nombres entiers utilisant 4 bits, *i.e.* les nombres dans l'ensemble $[-8; 7] \cap \mathbb{Z}$.

On constate que contrairement aux deux précédentes représentations, la représentation en complément à deux n'est pas symétrique. En effet, si on a besoin de w bits pour représenter -2^{w-1}, il faut $w+1$ bits pour représenter son opposé 2^{w-1}.

2.1.1.4 Comparaison des représentations pour les nombres entiers

La double représentation du 0, dans les représentations SVA et en complément à un, compliquent les opérateurs matériels pour les opérations arithmétiques de base et la comparaison entre deux nombres. En comparaison, la représentation en complément à deux permet d'effectuer les opérations arithmétiques de base de façon simple [8]. Par exemple, l'addition de deux nombres dans cette représentation s'effectue bit à bit en partant du bits de poids le plus faible vers le bit de poids le plus fort, modulo 2^w, c'est-à-dire comme pour les nombres binaires non signés.

2.1.2 Nombres réels

Il existe plusieurs façons de représenter et d'approcher les nombres réels en machine, mais nous nous intéresserons uniquement aux deux principales représentations, les nombres à virgule flottante et les nombres à virgule fixe. Pour les nombres à virgule flottante, nous présenterons uniquement la norme IEEE754.

2.1.2.1 Virgule Flottante (Norme IEEE754)

La norme IEEE754 [48, 80], ou *IEEE Standard for Binary Floating-Point Arithmetic*, est le standard pour la représentation binaire des nombres à virgule flottante. Cette représentation se compose d'un signe s, d'un exposant e et d'une mantisse m (voir Figure 2.5). L'exposant e, stocké explicitement dans la représentation, est écrit dans une représentation biaisée afin d'être toujours positif. Ce biais, noté b, est égal à $2^{w_e-1} - 1$, où w_e est le nombre de bits alloués à l'exposant dans la représentation (on dira également que w_e est la largeur de e). Le tableau 2.1 donne les largeurs des différentes composantes s, e et m, ainsi que la valeur du biais b, pour les précisions simple et double dans la norme IEEE754. L'exposant biaisé e prend donc ses valeurs dans l'ensemble $[0; 2^{w_e} - 1] \cap \mathbb{N}$, où les deux bornes 0 et $2^{w_e} - 1$ sont réservées à certains cas particuliers.

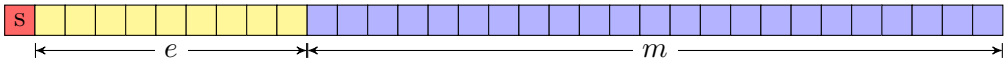


Figure 2.5 – Représentation de la répartition des bits pour un nombre en virgule flottante simple précision dans la norme IEEE754.

précision	largeur totale	signe	exposant	mantisse	biais
simple	32	1	8	23	127
double	64	1	11	52	1023

Table 2.1 – Tailles des différentes composantes des deux principaux formats spécifiés par la norme IEEE754, et valeurs du biais pour les précisions simple et double [80].

De plus, afin de fournir l'unicité de la représentation d'un nombre réel $x \in \mathbb{R}$ dans la norme IEEE754, x est normalisé, c'est-à-dire qu'on écrit x sous la forme

$$x = (-1)^s \cdot m \cdot 2^e,$$

où $m \in [1; 2[$ (e étant ici l'exposant non biaisé) est la mantisse normalisée de x que l'on veut stocker dans notre représentation. Le premier bit d'une mantisse normalisée étant toujours 1, il n'est pas stocké dans la représentation et est donc implicite.

Il peut arriver que x ne puisse pas être normalisé (si $x < 2^{e_{min}}$, où e_{min} est le plus petit exposant non biaisé possible [2] pour représenter un réel x), x est alors dit *sous-normalisé* (ou *dénormalisé*). Pour un nombre sous-normalisé,

2. Pour un réel x on a vu que l'exposant biaisé e était dans l'intervalle $]0; 2^{w_e} - 1[\cap\mathbb{N}$, donc au minimum 1. En soustrayant le biais b on obtient $e_{min} = -2^{w_e-1} + 2$.

l'exposant biaisé e et le bit implicite de poids fort de la mantisse valent tous deux 0.

Il existe des exceptions à ces règles permettant de coder les valeurs ± 0, $\pm\infty$ et `NaN` (*Not a Number*, pour les erreurs de calculs telles que le résultat d'une division par 0). Le tableau 2.2 illustre les différentes valeurs représentables à partir de l'exposant biaisé e et de la mantisse stockée m (dans ce tableau, w_m indique la largeur de m).

exposant biaisé e	mantisse m (sans le bit implicite) $m_{w_m-1} \dots m_1 m_0$	valeur représentée
$11\dots 11$	$\neq 00\dots 00$	NaN
$11\dots 11$	$00\dots 00$	$(-1)^s \times \infty$
$00\dots 00$	$00\dots 00$	$(-1)^s \times 0$
$00\dots 00$	$\neq 00\dots 00$	$(-1)^s \times 0.m_{w_m-1} \dots m_1 m_0 \times 2^{e_{min}}$
$0 < e < 2^{w_e} - 1$	$0 \leqslant m \leqslant 2^{w_m} - 1$	$(-1)^s \times 1.m_{w_m-1} \dots m_1 m_0 \times 2^{e-b}$

TABLE 2.2 – Tailles des différentes composantes des formats spécifiés par la norme IEEE754, et valeurs du biais pour les précisions simple et double [80].

Exemple 2.1. *Écrivons la représentation approchée du réel π, notée $\tilde{\pi}$, dans la norme IEEE754. Tout d'abord, la décomposition binaire (infinie) de π s'écrit :*

$$\pi =_2 11.00100100001111110110101010001000100001011010001100 0\dots$$

On normalise cette représentation :

$$\pi =_2 1.10010010000111111011010101000100010000 1\dots \times 2^1$$

La mantisse obtenue est normale, *le bit implicite vaut 1, et la mantisse m stockée correspond aux 23 bits de poids fort situés immédiatement après la virgule (on considère l'arrondi au plus proche pour le bit de poids le plus faible). L'exposant non biaisé vaut 1 et l'exposant biaisé e s'obtient en ajoutant le biais b égal à 127, soit $e = 128$. On obtient donc la représentation de $\tilde{\pi}$ illustrée par la figure 2.6.*

0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	1

FIGURE 2.6 – Représentation de $\tilde{\pi}$ en flottant simple précision dans la norme IEEE754.

2.1.2.2 Virgule Fixe

La seconde représentation des nombres réels discutée ici est la représentation virgule fixe, qui consiste en un nombre entier (signé ou non) multiplié par un facteur d'échelle. Contrairement à l'exposant en virgule flottante, ce facteur d'échelle est implicite dans la notation d'un nombre en virgule fixe. Le nombre entier peut être exprimé dans les différentes représentations des nombres entiers discutées précédemment, mais usuellement, et pour des raisons de simplicité d'écriture des opérateurs arithmétiques, la représentation signée en complément à deux est préférée.

Dans cette thèse, les nombres en représentation virgule fixe seront toujours exprimés en utilisant la représentation signée en complément à deux.

Soit x un nombre en virgule fixe, si on note X la *mantisse* de x (l'entier stocké dans la représentation), x s'écrit alors :

$$x = X \times 2^{\ell}, \tag{2.5}$$

où 2^{ℓ} est le facteur d'échelle avec ℓ la position du bit le moins significatif (***l**east **s**ignificant **b**it, lsb*). En utilisant (2.4), on remplace X par sa représentation en complément à deux signée :

$$x = \left(-X_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} X_i 2^i \right) \times 2^{\ell}, \tag{2.6}$$

où w est la largeur (en bits) du nombre X. En effectuant le changement de variable $x_i = X_{i+\ell}$ provenant de l'équation (2.5), et en notant $m \triangleq w + \ell - 1$, on obtient :

$$x = -x_m \cdot 2^m + \sum_{i=\ell}^{m-1} x_i 2^i. \tag{2.7}$$

x_i est alors le i-ème bit de x et m la position du bit le plus significatif de x (***m**ost **s**ignificant **b**it, msb*). La figure 2.7 montre la représentation d'un nombre virgule fixe avec la position (implicite dans le nombre stocké) de la virgule (la virgule étant représentée par un point sur le schéma).

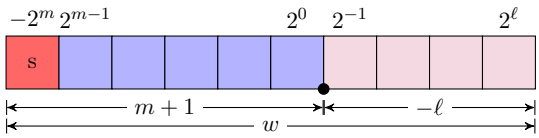


Figure 2.7 – Représentation d'un nombre en arithmétique virgule fixe.

Définition 2.1 (Format virgule fixe)**.** *Le format virgule fixe (ou Fixed-Point Format en anglais, nommé dans la suite par l'acronyme FPF) d'un nombre*

virgule fixe est déterminé par les positions des bits de poids fort m et de poids faible ℓ. Ce format sera par la suite noté (m, ℓ).

De plus, la largeur w de ce nombre peut-être déterminée à partir de m et ℓ par :

$$w \triangleq m - \ell + 1. \tag{2.8}$$

On peut noter enfin qu'on a toujours $m > \ell$ dans le cas non signé et $m > \ell + 1$ dans le cas signé.

Remarque 2.1. *Il existe de nombreuses définitions et notations différentes pour le FPF d'un nombre. Souvent le FPF d'un nombre est défini à partir du nombre de bits pour la partie entière, bit de signe exclu (noté i pour* Integer part*) et du nombre de bits pour la partie fractionnaire (noté f pour* Fractional part*) et est noté $Q_{i,f}$. On a alors les relations suivante :*

$$i = m \quad \text{et} \quad f = -\ell.$$

Cette notation, appelée notation en Q*, est la notation de Texas Instuments pour les nombres virgule fixe signés en complément à deux*[3]*. Cette notation trouve ses limites par exemple quand ℓ est positif. En effet, f est alors négatif et cela revient à dire qu'un nombre de bits est négatif, ce qui n'a pas de sens. C'est pourquoi nous préférerons parler de position du bit de poids le plus fort et de position du bit de poids le plus faible.*

Remarque 2.2. *Si usuellement on considère (m, ℓ) tel que $m > 0$ et $\ell < 0$, il peut arriver fréquemment d'avoir $m \leqslant 0$ (si le nombre est plus petit que 1 en valeur absolue), ou d'avoir $\ell \geqslant 0$ (si le nombre est un multiple d'une puissance positive de 2, ou si le nombre est très grand et que la largeur est assez petite pour imposer un ℓ positif). La figure 2.8 illustre de tels cas.*

Il n'existe pas de formalisation de la représentation virgule fixe (au sens de la norme IEEE pour la virgule flottante), on peut alors, et c'est tout l'enjeu de cette représentation, choisir la largeur que l'on veut, ainsi que le format (m, ℓ), pour un nombre virgule fixe.

2.1.2.3 Comparaison entre virgule flottante et virgule fixe

Dans [75], D. Ménard compare les codages en virgule flottante et en virgule fixe selon deux critères, celui de l'analyse de la dynamique et le rapport signal à bruit de quantification (RSBQ).

3. Le Q viendrait de Quotient, car à la base la notation en Q n'était suivie que d'un seul nombre, f. Cette première notation était ambiguë car on ne pouvait pas en déduire la largeur du nombre représenté. Afin de lever toute ambiguité, la notation $Q_{i,f}$ est apparue, et on peut trouver dans certains documents récents de Texas Instruments la notation $IiQf$, I pour Integer suivi du nombre de bits pour la partie entière, et Q pour Quotient, suivi du nombre de bits pour la partie fractionnaire.

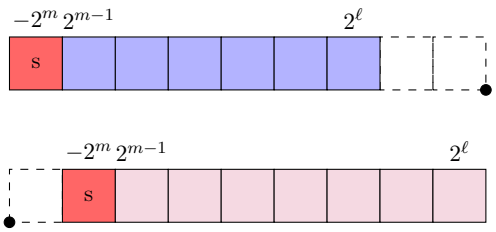


Figure 2.8 – D'autres représentations de nombres en arithmétique virgule fixe. Dans le premier cas ℓ est positif et dans le second m est négatif.

Le niveau de dynamique correspond au rapport logarithmique entre la plus grande valeur et la plus petite valeur représentables sur un nombre de bits donné. Le RSBQ, quant à lui, est une mesure définie comme le rapport entre la puissance du signal et la puissance de l'erreur de quantification. Ces deux mesures mettent en avant que l'exposant explicite du codage en virgule flottante permet une plus grande dynamique des données représentables et permet d'avoir un RSBQ quasi-constant pour la virgule flottante quand celui de la virgule fixe est fonction du signal d'entrée.

Le codage virgule fixe reste cependant une solution préférable pour l'implantation sur des systèmes embarqués. En effet, tout processeur embarqué possède des unités de calcul en entier mais pas forcément d'unité de calcul en flottant. Les calculs en virgule fixe sont plus simples à mettre en œuvre car basés sur le calcul en entiers (calcul bit-à-bit), alors que les calculs en flottant demandent plus de manipulations (dénormalisation des opérandes et normalisation du résultat). De plus, la flexibilité des tailles des données en arithmétique virgule fixe, par rapport à l'arithmétique flottante dans la norme IEEE754, permet d'avoir des processeurs plus petits et donc moins gourmands en énergie. On peut également citer le coût financier de la production de processeurs sans unité de calculs flottants qui est bien moins élevé que celui de processeurs avec unités de calculs flottants. La contrepartie de l'arithmétique virgule fixe est que c'est au développeur d'un algorithme virgule fixe de s'assurer que les opérations ont un sens (mêmes positions de la virgule pour les opérandes) et de gérer les problèmes de dynamiques (débordements, choix de la position de la virgule).

C'est pour ces raisons que l'arithmétique virgule fixe est préféré pour les applications embarquées, et que les besoins en méthodes et outils pour aider à une implémentation fiable sont grands. Les prochaines parties présentent alors les méthodes de conversion et de calcul dans cette arithmétique.

2.2 Conversion en virgule fixe

Cette section décrit la conversion d'une constante réelle en une constante virgule fixe, selon plusieurs cas de figure (largeur fixée, non fixée, format fixé). Cette conversion peut aussi s'appliquer à la conversion d'une constante flottante ou à la conversion d'une constante déjà en virgule fixe (on peut alors parler de raffinement du format virgule fixe), on considérant ces dernières comme des constantes réelles à précision infinie.

On présentera également comment convertir un intervalle réel en un intervalle virgule fixe, pour déterminer les bornes d'une variable réelle définie sur un intervalle et convertie en virgule fixe.

Par ailleurs, on ne considérera que la conversion d'un nombre réel non nul. En effet, puisque tous les bits de la décomposition binaire (infinie) de 0 sont nuls, on ne peut pas déterminer de positions de bit de poids fort ou de bit de poids faible. Ainsi, on admettra que 0 peut prendre n'importe quel format virgule fixe.

2.2.1 Conversion à largeur connue

2.2.1.1 Format virgule fixe inconnu

La première étape de la conversion d'un nombre réel non nul $c \in \mathbb{R}^*$ en un nombre virgule fixe où seule la largeur w du nombre virgule fixe est connue, est le calcul de la position de la virgule, qui est déterminée par la position du bit le plus significatif, m. En effet, si m est surestimé, on aura des répétitions du bit de signe (voir Figure 2.9), et si m est sous-estimé, alors c ne sera pas représentable dans le format calculé, on parlera alors de *débordement* (voir partie 2.3.1.4).

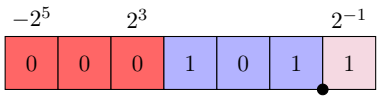


FIGURE 2.9 – Représentation virgule fixe du nombre 5.5 où la position du *msb* a été surestimé. En effet le FPF est $(5, -1)$ alors que $(3, -1)$ était suffisant, et donc il y a deux bits de signes inutiles dans ce cas là.

Il est donc important d'estimer au plus juste la position de ce bit, ce qui revient donc à trouver la puissance de 2 immédiatement supérieure à c (ou immédiatement inférieure si $c < 0$).

On veut alors déterminer le plus petit entier naturel m tel que, pour une constante réelle $c \in \mathbb{R}^*$ on a l'écriture (potentiellement infinie) en complément à deux suivante :

$$c = -c_m \cdot 2^m + \sum_{i=-\infty}^{m-1} c_i 2^i, \tag{2.9}$$

c'est-à-dire telle que, si $c > 0$, alors $c_m = 0$ et $c_{m-1} = 1$, et si $c < 0$, alors $c_m = 1$ et $c_{m-1} = 0$.

Compte tenu que la représentation en complément à deux n'est pas symétrique, on peut alors différencier le cas positif et le cas négatif.

1er cas - c est strictement positif : D'après (2.9), on a $c_m = 0$ et $c_{m-1} = 1$, donc $c = \sum_{i=-\infty}^{m-1} c_i 2^i$, par conséquent :

$$\begin{array}{rcl} 2^{m-1} \leqslant & c & < 2^m \\ m-1 \leqslant & \log_2(c) & < m \end{array}$$

et on en déduit

$$m = \lfloor \log_2(c) \rfloor + 1. \tag{2.10}$$

Remarque 2.3. *En fait, même avec nos hypothèses sur m, on peut avoir $c = 2^m$, dans le cas où $c_i = 1$ pour tout $-\infty < i \leqslant m-1$. Ce cas particulier est exclu des hypothèses ici, on peut en effet remplacer un tel nombre par c' avec $c'_m = 1$ et $c'_i = 0$ pour $i \leqslant m-1$ (et on aurait $c = c'$). On a le cas analogue en base 10, où les nombres $1.000\ldots$ et $0.999\ldots$ sont égaux.*

2ème cas - c est strictement négatif : Toujours d'après (2.9), on a $c_m = 1$ et $c_{m-1} = 0$, donc $c = -2^m + \sum_{i=-\infty}^{m-1} c_i 2^i$, ce qui nous donne :

$$\begin{array}{rcl} -2^m \leqslant & c & < -2^{m-1} \\ m-1 < & \log_2(-c) & \leqslant m \end{array}$$

et on en déduit

$$m = \lceil \log_2(-c) \rceil. \tag{2.11}$$

Les deux cas précédents peuvent être regroupés par la formule suivante :

$$m = \begin{cases} \lfloor \log_2(c) \rfloor + 1 & \text{si } c > 0 \\ \lceil \log_2(-c) \rceil & \text{si } c < 0 \end{cases} \tag{2.12}$$

Il s'agit en fait ici, dans les deux cas, d'une première estimation de m. En effet, nous allons voir dans la suite que m peut nécessiter un réajustement dans des cas particuliers d'arrondi de la mantisse.

Après cette étape, on connait la position du bit le plus significatif m et la largeur w dont on dispose pour convertir c en virgule fixe, on peut donc en déduire la position du bit le moins significatif ℓ d'après (2.8) par $\ell = m - w + 1$. Par conséquent, on peut calculer la valeur de la mantisse $C \in \mathbb{Z}$ correspondant aux w bits les plus significatifs de c en complément à deux. Comme c est un nombre réel, dans le cas général w bits ne suffiront pas pour représenter

exactement c, on va donc estimer une valeur approchée au plus proche de c sur w bits. La mantisse C est donc donnée par :

$$C = \left\lfloor c.2^{-\ell} \right\rceil \tag{2.13}$$

où $\lfloor \cdot \rceil$ est l'opérateur d'arrondi au plus proche. Une fois la mantisse C calculée, on obtient la valeur virgule fixe de c sur w, notée $\tilde{c}$, par :

$$\tilde{c} = C \cdot 2^{\ell}. \tag{2.14}$$

Considérons maintenant un exemple pour mettre en évidence certains cas particuliers.

Exemple 2.2. *Soient une constante $c = 127.6$ et une largeur $w = 8$, on souhaite convertir c sur w bits. La décomposition binaire de c s'écrit :*

$$01111111.10011001100110011001100110011001100110011000\ldots$$

On commence par calculer m à partir de (2.10) (car $c > 0$) et on obtient $m = 7$. Donc, d'après (2.8) et (2.13), on a :

$$C = \lfloor c.2^0 \rceil = \lfloor c \rceil = 128,$$

et de (2.14) on déduit $\tilde{c} = C = 128$.

On remarque alors que le format virgule fixe calculé, $(7, 0)$, ne permet pas de représenter $\tilde{c}$, car C doit appartenir à l'intervalle $[-128; 127]$. En effet, sur 8 bits, le format permettant de représenter $\tilde{c}$ au plus juste est $(8, 1)$.

On peut observer un phénomène similaire pour les nombres négatifs si on essaye de convertir par exemple -64.4 sur 8 bits. En effet, puisque $-64.4 < -64 = -2^6$, la formule (2.11) donnera $m = 7$ et donc le format virgule fixe $(7, 0)$, alors que (2.14) donnera $\tilde{c} = -64$, qui est représentable sur le format $(6, 1)$. A priori le problème est moindre dans ce cas là, on a juste *un bit de signe répété inutilement, mais en regardant de plus près, si on recalcule C et $\tilde{c}$ avec le format $(6, 1)$ on obtient $\tilde{c} = -64.5$ qui est un résultat plus précis.*

Nous parlerons alors de *cas limites* car ces deux cas se produisent quand le nombre à convertir est *proche* de la puissance de 2 immédiatement supérieure. Il est donc nécessaire, une fois que m et C ont été estimés, de vérifier que nous ne sommes pas dans un cas limite.

- Si $C = 2^{w-1}$ alors
  - on ajoute 1 à m
  - on réévalue $C = \left\lfloor c.2^{-\ell} \right\rceil$, avec $\ell = m - w + 1$
- Si $C = -2^{w-2}$ alors
  - on retranche 1 à m
  - on réévalue $C = \left\lfloor c.2^{-\ell} \right\rceil$, avec $\ell = m - w + 1$

Enfin, on calcule $\tilde{c}$ d'après (2.14), ce qui achève la conversion de c sur w bits. La méthode de conversion complète est décrite par l'algorithme 1.

Algorithme 1: Conversion d'une constante c sur w bits, $c \neq 0$

1ère estimation de m
Si $c > 0$ **Alors**
  $m \leftarrow \lfloor \log_2(c) \rfloor + 1$;
Sinon
  $m \leftarrow \lceil \log_2(-c) \rceil$;
Fin

Calcul de C
$C \leftarrow \lfloor c.2^{w-m-1} \rceil$;

Vérification des cas limites
Si $C = 2^{w-1}$ **Alors**
  $m \leftarrow m + 1$;
  $C \leftarrow \lfloor c.2^{w-m-1} \rceil$;
Fin
Si $C = -2^{w-2}$ **Alors**
  $m \leftarrow m - 1$;
  $C \leftarrow \lfloor c.2^{w-m-1} \rceil$;
Fin

Calcul de ℓ et $\tilde{c}$
$\ell \leftarrow w - m - 1$;
$\tilde{c} \leftarrow C \cdot 2^{\ell}$;

2.2.1.2 Format virgule fixe connu

Il peut arriver lors d'une conversion d'un réel en virgule fixe que le format virgule fixe soit imposé. Soient $c \in \mathbb{R}$ un nombre réel à convertir et (m', ℓ') le format imposé pour la conversion. La largeur w est déterminée par (2.8) à partir de (m', ℓ'), on peut donc appliquer l'algorithme 1 pour déterminer le FPF idéal de c sur w bits, noté (m, ℓ). Il y a alors deux cas à considérer, le cas $m < m'$ et le cas $m > m'$ (le cas $m = m'$ étant trivial).

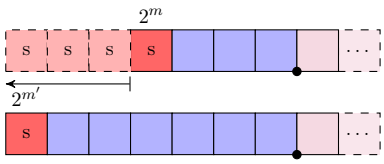


Figure 2.10 – Extension du bit de signe dans le cas $m' > m$.

Cas $m < m'$: Cela signifie que $|c| < 2^{m'}$ donc c est représentable dans le format imposé. On fait alors une extension du bit de signe, c'est-à-dire qu'on aura $c_i = c_m$ pour $m + 1 \leqslant i \leqslant m'$ (voir figure 2.10).

La conversion se fait ensuite sur le format (m', ℓ') en calculant $C = \lfloor c.2^{-\ell'} \rceil$ et $\tilde{c} = C \cdot 2^{\ell'}$. Les cas limites ne sont pas testés puisque le format est imposé et ne peut donc pas être changé.

Cas $m' < m$: Puisque m est estimé au plus juste pour représenter c, ce cas signifie que c n'est pas représentable sur le format imposé, on parle alors de débordement (l'anglicisme *overflow* peut aussi être utilisé). Le traitement des débordements est décrit partie 2.3.1.4 et se fait soit par arithmétique modulaire, soit par saturation. Dans les deux cas, les bits en positions m à $m' + 1$ sont supprimés et le traitement effectué sur les bits restants dépend du choix du mode de débordement.

2.2.2 Conversion à largeur inconnnue

Il peut arriver, comme c'est le cas dans les problèmes d'optimisation des largeurs, que la largeur ne soit pas connue au moment de la conversion. Cette partie a uniquement pour but, quand la largeur n'est pas fixée, de préciser comment éviter les cas limites. En effet, on calculera toujours m d'après (2.12), et on pourra ensuite estimer le nombre de bits nécessaires pour éviter les cas limites décrits précédemment.

Pour cela mettons en évidence ce qu'il se produit lors d'un cas limite.

Cas limite avec $c > 0$: La constante c s'écrit $c = \sum_{i=-\infty}^{m-1} c_i 2^i$ et m est calculé par (2.12). Supposons alors que l'on choisisse une largeur w pour pouvoir calculer C (et par déduction $\tilde{c}$), cette largeur nous permet de déterminer ℓ. Ensuite, d'après (2.13) et puisqu'on est par hypothèse dans un cas limite, on a :

$$C = \left\lfloor \left(\sum_{i=-\infty}^{m-1} c_i 2^i \right) .2^{-\ell} \right\rceil = 2^{w-1}, \tag{2.15}$$

et donc $\tilde{c} = 2^m$, alors que par définition $c < 2^m$. L'explication vient du fait que

$$c_i = 1 \quad \forall \ell - 1 \leqslant i \leqslant m - 1,$$

c'est-à-dire que les $w + 1$ bits de poids fort de c (excepté le bit de signe qui vaut 0), valent 1, et on peut alors vérifier qu'on a :

$$c.2^{-\ell} \geqslant 2^{w-1} - 1 + 2^{-1} = 2^{w-1} - 2^{-1} \tag{2.16}$$

et on obtient bien que $\lfloor c.2^{-\ell} \rceil = 2^{w-1}$.

La Figure 2.11 illustre cette explication.

Cela signifie donc qu'on peut éviter les cas limites en choisissant w plus grand (et donc ℓ plus petit) de sorte que $c_{\ell-1}$ soit le premier bits à 0 (Figure 2.11).

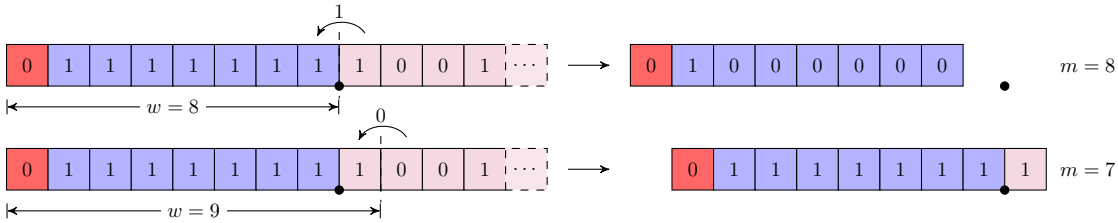


Figure 2.11 – Illustration d'un cas limite avec la conversion de $c = 127.6$ sur 8 bits (cas limite) et sur 9 bits (cas non limite). Sur 8 bits, la retenue de l'arrondi au plus proche de c engendre un changement de la position du bit le plus significatif.

Cas limite avec $c < 0$: Dans ce cas-ci on a $c = -2^m + \sum_{i=-\infty}^{m-1} c_i 2^i$ et on obtient, avec un w trop petit, $\tilde{c} = -2^{m-1}$. En effet on a $c_m = 1$, $c_{m-1} = 0$ puis une suite de bits à 1. Si on choisit w tel que la position de ℓ est dans cette suite de 1, alors l'arrondi au plus proche propagera une retenue valant 1 jusqu'à la position $m-1$ et on aura $c_m = c_{m-1} = 1$, et donc une répétition du bit de signe inutile. En choisissant w tel que ℓ est la position du dernier bit de la suite de 1 (comme dans le cas précédent), alors on évite ce cas limite.

2.2.3 Conversion à partir d'un intervalle

Nous avons vu jusqu'ici comment convertir une constante réelle en un nombre virgule fixe, mais quid des variables définies sur des intervalles ?

N.B. : Dans ces travaux, nous ne ferons pas la distinction entre une variable et l'intervalle dans lequel elle est définie.

Soit une variable réelle $x \in [\underline{x}; \overline{x}] \subset \mathbb{R}$, on veut savoir comment représenter x sur w bits. On commence par définir la notion d'intervalle virgule fixe.

Définition 2.2 (Intervalle virgule fixe)**.** *Un intervalle virgule fixe est l'ensemble des valeurs comprises entre deux bornes et ayant toutes un unique format commun. Si on note $[\underline{x}; \overline{x}]_{(m,\ell)}$ l'intervalle virgule fixe comprenant l'ensemble des valeurs virgule fixe au format (m, ℓ) entre $\underline{x}$ et $\overline{x}$, alors on a :*

$$[\underline{x}; \overline{x}]_{(m,\ell)} = \{x | \underline{x} \leqslant x \leqslant \overline{x} \text{ et } x \text{ en virgule fixe au format } (m, \ell)\}. \tag{2.17}$$

Puisque (m, ℓ) est le format commun et qu'on a $\underline{x} = \underline{X}.2^\ell$ et $\overline{x} = \overline{X}.2^\ell$ avec $\underline{X}$ et $\overline{X}$ les mantisses respectives de $\underline{x}$ et $\overline{x}$, alors on peut écrire :

$$[\underline{x}; \overline{x}]_{(m,\ell)} = \left\{ X \cdot 2^\ell | X \in [\underline{X}; \overline{X}] \cap \mathbb{Z} \right\} \tag{2.18}$$

On peut alors dire que 2^ℓ est le pas de quantification de l'intervalle $[\underline{x}; \overline{x}]_{(m,\ell)}$. De plus, on préfèrera généralement noter l'intervalle virgule fixe x en sous-entendant que les bornes de l'intervalle sont $\underline{x}$ et $\overline{x}$.

On en déduit la définition d'une variable virgule fixe.

Définition 2.3 (Variable virgule fixe)**.** *Une variable en virgule fixe se définit comme un nombre virgule fixe qui peut prendre n'importe quelle valeur à l'intérieur d'un intervalle virgule fixe.*

Pour représenter $x \in \mathbb{R}$ sur w bits il faut déterminer le format commun à toute valeur que x peut prendre. Pour cela, il suffit de déterminer le format de $\underline{x}$ et $\overline{x}$.

On considère donc $\underline{x}$ et $\overline{x}$ comme deux constantes, on applique l'algorithme de conversion (Algorithme 1) sur chacune d'elle. On obtient ainsi, pour $\underline{x}$, sa valeur FxP convertie notée $\underline{\tilde{x}}$ et son format $(m_{\underline{x}}, \ell_{\underline{x}})$ et de la même manière pour $\overline{x}$ on obtient sa valeur FxP convertie $\overline{\tilde{x}}$ et son format $(m_{\overline{x}}, \ell_{\overline{x}})$. Le FPF ayant le plus grand m entre $(m_{\underline{x}}, \ell_{\underline{x}})$ et $(m_{\overline{x}}, \ell_{\overline{x}})$ est le format commun qui permet de représenter toutes les valeurs de la variable x. L'intervalle de la variable virgule fixe $\tilde{x}$ est donné par $[\underline{x}'; \overline{x}']_{(m,\ell)}$ où $\underline{x}'$ est $\underline{x}$ converti sur le format (m, ℓ), $\overline{x}'$ est $\overline{x}$ converti sur le format (m, ℓ) et (m, ℓ) est le format commun.

2.3 Opérations virgule fixe

Dans cette partie sont décrites les deux opérations usuelles d'arithmétique utilisées dans nos travaux, l'addition virgule fixe entre deux variables et la multiplication virgule fixe entre une variable et une constante. Il y a, dans la description des deux opérations, deux étapes à différencier :

- la façon dont on détermine le format virgule fixe du résultat à partir du format, des intervalles (dans le cas de variables) ou valeurs (dans le cas de constantes) des opérandes et des autres données (la largeur du résultat peut être fixée),
- et la façon dont est effectivement calculée l'opération une fois le format connu.

Dans le cas d'une addition, puisque le but est d'implémenter des algorithmes avec uniquement des entiers (les différents facteurs d'échelles sont implicites), il faut faire en sorte que les facteurs d'échelle correspondant aux deux opérandes d'une même opération soient les mêmes, on dit alors qu'on *aligne* les virgules des deux nombres. Cette phase d'alignement se fait en utilisant des formatages, ces derniers sont donc décrits dans la première sous-partie, avant de passer à la description des opérations.

2.3.1 Formatage

On appelle formatage d'un nombre virgule fixe l'opération consistant à modifier le format de ce nombre pour le représenter sur un nouveau format donné. Considérons un nombre FxP c au format (m, ℓ), ainsi que (m', ℓ'), le

format dans lequel on veut représenter c. On rappelle que C désigne la mantisse de c, c.-à-d. l'entier tel que $c = C \cdot 2^{\ell}$. Pour passer du format (m, ℓ) au format (m', ℓ'), il y a plusieurs cas à considérer en comparant m à m' et ℓ à ℓ', mais on peut également différencier le cas d'une cible matérielle et le cas d'une cible logicielle. Puisque dans certains cas, formater c revient à décaler C, on commence donc par définir le décalage sur un entier.

2.3.1.1 Décalage sur un entier

Le décalage est une opération consistant à modifier la représentation binaire d'un nombre en ajoutant (décalage à gauche) ou en supprimant (décalage à droite) des bits à droite du nombre.

Soient un nombre entier $C \in \mathbb{Z}$ et $d \in \mathbb{Z}$ le nombre de bits dont on veut décaler C. Pour un nombre entier, l'opérateur de décalage à gauche se note $\ll$ et peut être défini comme suit :

$$C \ll d \triangleq C \times 2^{d} \tag{2.19}$$

Pour l'opérateur de décalage à droite, noté $\gg$, on a :

$$C \gg d \triangleq \left\lfloor C \times 2^{-d} \right\rfloor \tag{2.20}$$

(en considérant la représentation binaire en complément à 2).

À un décalage à droite correspond donc un arrondi de la valeur rationnelle décalée. Or dans nos travaux nous considérons deux types d'arrondis, par troncature et au plus proche (voir 2.4). Par abus de langage et de notation, nous considérerons donc que le décalage à droite peut s'effectuer avec un arrondi par troncature ou un arrondi au plus proche :

$$C \gg d \triangleq \begin{cases} \lfloor C \times 2^{-d} \rfloor & \text{si arrondi par troncature} \\ \lfloor C \times 2^{-d} \rceil & \text{si arrondi au plus proche} \end{cases} . \tag{2.21}$$

Comme nous l'avons vu, le décalage à droite supprime des bits, donc une erreur d'arrondi est produite. Dans la section 2.4.2, nous verrons comment évaluer cette erreur de calcul selon le type d'arrondi.

2.3.1.2 Formatage sur une cible matérielle

Dans les cibles matérielles, c'est l'utilisateur qui conçoit ses opérateurs et donc décide de leurs largeurs selon ses besoins. Ayant connaissance des formats de départ et d'arrivée, l'utilisateur peut donc soit réduire la largeur d'une variable, soit l'augmenter en lui allouant plus de bits.

- $m' \geqslant m$: le bit en position m, c.-à-d. le bit de signe de c, est étendu (c.-à-d. recopié) vers la gauche jusqu'à la position m'.
- $m' < m$: les bits de poids en positions $m, m-1, \ldots m'+1$ sont supprimés, il y a alors débordement (voir 2.3.1.4).

- $\ell' \leqslant \ell$: le format est étendu vers la droite en rajoutant $\ell - \ell'$ bits à 0.
- $\ell' > \ell$: les $\ell' - \ell$ bits de poids faible de c sont supprimés et il y a donc un arrondi. Or dans nos travaux nous considérons deux types d'arrondis, par troncature et au plus proche (voir 2.4). L'arrondi par troncature consiste simplement à supprimer les bits tandis que l'arrondis au plus proche consiste à ajouter le bit supprimé de poids le plus fort (en position $\ell' - 1$) en position ℓ' et à tronquer les $\ell' - \ell$ bits de poids faible.

Remarque 2.4. *Dans le cas $\ell' > \ell$ avec arrondi au plus proche, on peut tomber sur un cas limite et donc m devra être réajusté.*

En matériel, tous ces traitements sont gratuits (en considérant un formatage constant au cours du temps), mis à part le cas $\ell' > \ell$ en considérant un arrondi au plus proche, qui recourt à une addition.

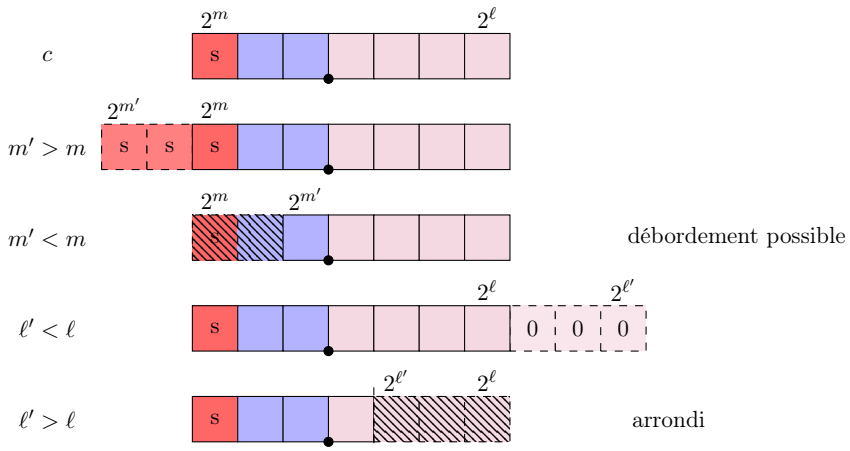


FIGURE 2.12 – Illustration des différents cas de formatage pour une constante c en matériel.

2.3.1.3 Formatage sur une cible logicielle

Dans le cas d'une cible logicielle, il faut considérer une condition supplémentaire. En effet, les opérateurs et registres ayant des largeurs fixées et non-modifiables, il est nécessaire que la largeur de départ de c, ainsi que la largeur d'arrivée, soient inférieures ou égales à celle de l'opérateur, notée w_L, c.-à-d. on a :

$$m - \ell + 1 \leqslant w_L \tag{2.22}$$
$$m' - \ell' + 1 \leqslant w_L \tag{2.23}$$

- $m' \geqslant m$: on peut considérer deux cas :

- $m' - \ell + 1 \leqslant w_L$: aucun traitement particulier n'est à effectuer, le bit de signe est déjà étendu.
- $m' - \ell + 1 > w_L$: d'après la condition (2.23) on a $\ell' > \ell$ et donc on applique sur C un décalage à droite de $\ell' - \ell$ bits (figure 2.13). L'instruction de décalage insère $\ell' - \ell$ bits de signe à gauche pour conserver la même largeur.

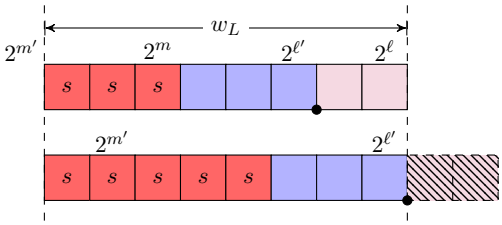


Figure 2.13 – Décalage à droite de C dans le cas où $m' \geqslant m$ avec $\ell' > \ell$.

- $m' < m$: le bit en position m' devient le nouveau bit de signe du nombre et il est répété aux positions supérieures. Pour ce faire, on décale C à gauche de $d = w_L - (m' - \ell + 1)$ bits (pour placer le bit en position m' tout à gauche du registre et ainsi supprimer les bits supérieurs) puis on re-décale C à droite du même nombre de bits pour les replacer tout à droite et ainsi insérer des bits de signes sur les bits de poids fort.

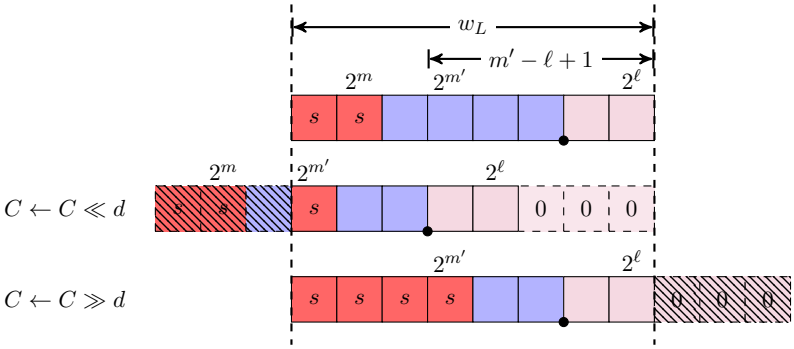


Figure 2.14 – Cas $m' < m$, on calcule $(C \ll d) \gg d$.

- $\ell' \geqslant \ell$: C est décalé à droite de $\ell' - \ell$ bits (on se retrouve sur un cas précédent illustré par la figure 2.13).
- $\ell' < \ell$: C est décalé à gauche de $\ell - \ell'$ bits.

2.3.1.4 Débordement

Cette partie décrit ce qu'il se passe lorsqu'on veut représenter une valeur dans un format trop petit pour la représenter (par exemple quand le format

d'une somme est fixé). Il y a en effet deux types de débordements possibles, celui induit par l'arithmétique modulaire et celui dit de saturation.

Arithmétique modulaire : Le débordement induit par l'arithmétique modulaire (lui-même dû à l'utilisation de la représentation en complément à deux) consiste simplement à prendre le nombre induit par les bits représentables sur la largeur donnée. Soit une valeur virgule fixe v au format (m, ℓ) que l'on souhaite représenter sur le format (m', ℓ) avec $m' < m$. On a par définition de v :

$$v = -v_m \cdot 2^m + \sum_{i=\ell}^{m-1} v_i 2^i \tag{2.24}$$

et le nombre virgule fixe représentant v sur le format (m', ℓ), noté $\tilde{v}$, en considérant le débordement induit par l'arithmétique modulaire est donné par :

$$\tilde{v} \triangleq -v_{m'} \cdot 2^{m'} + \sum_{i=\ell}^{m'-1} v_i 2^i. \tag{2.25}$$

Le bit $v_{m'}$ est alors interprété comme nouveau bit de signe à la place de v_m.

L'équation mathématique donnant $\tilde{v}$ à partir de v et du format (m', ℓ) est la suivante :

$$\tilde{v} = \left(\left(\left((v + 2^{m'}) \cdot 2^{-\ell}\right) \mod 2^{m'-\ell+1}\right) \cdot 2^{\ell}\right) - 2^{m'} \tag{2.26}$$

Cette égalité se vérifie en appliquant les opérations successives sur le développement binaire en complément à deux de v de la formule (2.24), on retrouve alors bien sur le développement binaire en complément à deux de $\tilde{v}$ de l'équation (2.25).

Remarque 2.5. *La réinterprétation du bit en position m' comme bit de signe nécessite de convertir v en entier positif pour appliquer l'opération modulaire avant d'appliquer la conversion inverse, ce qui explique le $+2^{m'}$ et le $-2^{m'}$ dans l'équation (2.26).*

Exemple 2.3. *Soit un nombre $c = 36$ au format $(6, 0)$, et on souhaite le représenter sur le format $(5, 0)$. En binaire c s'écrit :*

$$c =_2 0100100$$

On supprime le bit le plus significatif, on obtient alors la mantisse 100100. Le nombre virgule fixe associé est obtenu en réinterprétant le bit le plus significatif égal à 1, ce qui veut dire que le nombre est négatif. En appliquant la formule ci-dessus, le nombre obtenu ainsi est $\tilde{c} = -28$ $(-32 + 4)$.

On peut, par analogie avec la figure 2.4, construire un cercle avec les valeurs du format $(5, 0)$, c'est-à-dire les entiers de -32 à 31. Ainsi, pour obtenir la

représentation de 36*, on part de* 0 *et on "avance" sur le cercle* 36 *fois. Après* 31 *itérations, on se trouve sur la plus grande valeur positive représentable sur le format* $(5, 0)$*, et à l'étape suivante on passe donc de* 31 *à* -32*. Les quatre dernières itérations amène logiquement sur la valeur* -28*.*

Cette opération modulaire consistant à passer de la plus grande valeur représentable à la plus petite valeur représentable (ou inversement) est donc source d'une grande erreur.

Dans la section 3.3.2 nous verrons une application de l'arithmétique modulaire dans le cas d'une somme de plusieurs termes en virgule fixe, qui autorise les débordements intermédiaires tant que le résultat final est représentable sur le format imposé.

Saturation : Le mode de débordement par saturation consiste quant à lui à considérer la plus grande valeur représentable sur le format donné et de même signe que la valeur à représenter, ou en d'autres termes la valeur représentable la plus proche de la valeur à représenter.

Soit une valeur v virgule fixe au format (m, ℓ) que l'on souhaite représenter sur le format (m', ℓ) avec $m' < m$. Le nombre virgule fixe représentant v sur le format (m', ℓ), noté $\tilde{v}$, en considérant le débordement par saturation est donné par :

$$\tilde{v} \triangleq \begin{cases} 2^{m'} - 2^{\ell} & \text{si } v > 0 \\ -2^{m'} & \text{sinon} \end{cases} . \tag{2.27}$$

Remarque 2.6. *L'opération de saturation implique l'utilisation de matériel supplémentaire et fausse tous les calculs en complément à deux (notamment en n'autorisant pas le calcul d'une somme de plusieurs termes par arithmétique modulaire).*

Exemple 2.4. *On considère le même exemple, soit* $c = 36$ *au format* $(6, 0)$*, et on souhaite le représenter sur le format* $(5, 0)$*.* c *est positif donc le nombre obtenu par saturation est* $2^m - 2^{\ell} = 31$*.*

2.3.2 Addition virgule fixe

On décrira ici la somme de deux variables virgule fixe, c'est-à-dire des variables définies sur des intervalles virgule fixe. En effet, le but de nos travaux est de convertir un algorithme réel de filtre en un algorithme virgule fixe, or au moment de cette implémentation on a uniquement connaissance des intervalles dans lesquels les signaux d'entrées prendront leurs valeurs à l'exécution. Par conséquent la description de nos calculs se fera à partir de ces intervalles.

Selon l'additionneur que l'on considère (matériel ou logiciel), la largeur du résultat peut être fixée ou non. De plus, en matériel, nous verrons par la suite que nos applications d'implémentations de filtres (et surtout le rebouclage des filtres IIR) peuvent fixer le format de virgule fixe du résultat d'une somme.

Par conséquent la description de l'addition en virgule fixe se fera selon ces différents cas : largeur non fixée, largeur fixée, format virgule fixe du résultat fixé.

Lors d'une addition virgule fixe en complément à deux, il est important d'aligner les deux opérandes sur le même format (généralement celui du résultat) pour pouvoir ajouter les mantisses bit à bit (en propageant les retenues). La première étape consiste alors à déterminer le format commun des opérandes (sauf si celui-ci est fixé), pour ensuite formater ces derniers sur celui-ci, avant de calculer l'addition en utilisant l'arithmétique d'intervalle.

Notations : Dans cette partie, nous considérerons deux opérandes x et y. L'opérande x (resp. y) est une variable définie sur l'intervalle FxP $[\underline{x};\overline{x}]_{(m_x,\ell_x)}$ (resp. $[\underline{y};\overline{y}]_{(m_y,\ell_y)}$). Les variables $\underline{X}$ et $\overline{X}$ représentent les mantisses respectives de $\underline{x}$ et $\overline{x}$. De la même façon, le résultat de l'addition est noté z, l'intervalle associé à z est $[\underline{z};\overline{z}]_{(m_z,\ell_z)}$ et les variables $\underline{Z}$ et $\overline{Z}$ représentent les mantisses respectives de $\underline{z}$ et $\overline{z}$.

2.3.2.1 Choix du format

Dans l'optique d'aligner les opérandes sur un format commun, la première étape est de déterminer ce format commun (excepté si celui-ci est fixé). Cette étape est décrite ci-dessous dans les trois cas cités en introduction.

Largeur du résultat non fixée (cas matériel) : En matériel, on peut décrire l'additionneur de façon à ce qu'il effectue le calcul sur la largeur optimale, notée w_{opt}, c'est à dire la largeur minimale qui effectue l'addition exacte des deux opérandes (figure 2.15a).

Pour ce faire, il suffit de calculer, en utilisant l'arithmétique d'intervalle, la somme des intervalles des variables sans considérer les formats :

$$[\underline{z};\overline{z}] \triangleq [\underline{x};\overline{x}] + [\underline{y};\overline{y}]. \tag{2.28}$$

On rappelle qu'en arithmétique d'intervalle, la somme de deux intervalles réels $[\underline{a};\overline{a}]$ et $[\underline{b};\overline{b}]$ est définie par :

$$[\underline{a};\overline{a}] + [\underline{b};\overline{b}] \triangleq [\underline{a}+\underline{b};\overline{a}+\overline{b}]. \tag{2.29}$$

Ensuite, m_z est déterminé par le plus grand msb entre celui de $\underline{z}$ et celui de $\overline{z}$, eux-mêmes déterminés par l'algorithme 1, c.-à-d. on a :

$$m_z \triangleq \max(m_{\underline{z}}, m_{\overline{z}}). \tag{2.30}$$

Pour finir, puisqu'on veut effectuer une somme sans perte d'information (sans suppression de bit), ℓ_z est le plus petit lsb des deux opérandes, on a alors :

$$\ell_z \triangleq \min(\ell_x, \ell_y), \tag{2.31}$$

et on en déduit

$$w_{opt} = m_z - \ell_z + 1. \tag{2.32}$$

Largeur du résultat fixée mais format non fixé : En logiciel, la largeur d'un additionneur est fixée, et est la même pour le résultat et pour les opérandes (on peut noter ce principe par le schéma $w+w \to w$, la somme de deux éléments de même largeur w donne un élément de largeur w). Le bit de retenue existe mais est stocké à part, généralement dans un registre de configuration. De plus, en matériel, on peut également adopter ce même schéma ou bien décider de conserver le bit de retenue (schéma $w+w \to w+1$). On considère dans le cas présent (illustré par la figure 2.15b) une largeur fixée w_{fix} plus petite que la largeur optimale déterminée précédemment (le cas $w_{fix} \geqslant w_{opt}$ ne changeant rien à la méthode d'évaluation ci-dessus). L'exemple suivant vise à montrer pourquoi la méthode précédente ne fonctionne pas dans le cas w_{fix} fixée avec $w_z < w_{opt}$.

Exemple 2.5. *Soient $x \in [-64;100]_{(7,0)}$, $y \in [-32;28]_{(5,0)}$, et la largeur fixée $w_{fix} = 5$. En utilisant la méthode précédente, la somme des intervalles donne :*

$$[\underline{z};\overline{z}] \triangleq [-96;128], \tag{2.33}$$

D'après (2.30), $m_z = m_{\overline{z}} = 8$, et donc puisque w_z est fixée on a, d'après (2.8), $\ell_z = m_z - w_{fix} + 1 = 4$. Anticipons maintenant un peu sur l'étape suivante d'une somme, à savoir la mise au format commun, ici $(8,4)$. Sur ce format, l'intervalle de x devient $[-64;96]_{(8,4)}$ et celui de y devient $[-32;16]_{(8,4)}$ (on précise que le mode d'arrondi considéré est l'arrondi par troncature). Et donc on obtient l'intervalle FxP de z par :

$$[\underline{z};\overline{z}]_{(8,4)} \triangleq [-64;96]_{(8,4)} + [-32;16]_{(8,4)} = [-96;112]_{(8,4)}. \tag{2.34}$$

On s'aperçoit que les deux bornes de z sont représentables sur le format $(7,3)$.

On pouvait faire plus fin en considérant une autre méthode. En effet, posons dans un premier temps $\widetilde{m}_z \triangleq \max(m_x, m_y) = 7$ en tant que première estimation de m_z. On peut alors déterminer $\widetilde{\ell}_z$ à partir de $\widetilde{m}_z$ et w_{fix} par $\widetilde{\ell}_z = \widetilde{m}_z - w_{fix} + 1 = 3$ et on a un format commun $(\widetilde{m}_z, \widetilde{\ell}_z) = (7,3)$. On obtient alors, une fois les intervalles formatés, $x \in [-64;96]_{(7,3)}$, et $y \in [-32;24]_{(7,3)}$, et finalement :

$$[\underline{z};\overline{z}]_{(7,3)} \triangleq [-64;96]_{(7,3)} + [-32;24]_{(7,3)} = [-96;120]_{(7,3)}. \tag{2.35}$$

L'intervalle est alors plus grand et surtout le pas de quantification est divisé par deux.

Une fois le formatage effectué sur le format déterminé a priori, *il est possible d'avoir une retenue (si on avait prit la largeur optimale on aurait eu $\overline{z} = 128$ qui n'est pas représentable sur le format $(7,3)$) et donc il faut dans ce cas là passer au format $(8,4)$ et recalculer la somme des intervalles.*

Cet exemple montre bien qu'il est nécessaire d'adopter une nouvelle approche. Décrivons alors formellement la seconde méthode utilisée dans cet

exemple. Lors d'une addition, on sait que le *msb* du résultat sera égal au *msb* du plus grand opérande, plus un s'il y a une retenue. Cette retenue ne peut être déterminée qu'en ajoutant les bornes formatées des opérandes deux-à-deux (puisqu'on a vu qu'une fois formatées, l'addition des bornes peut donner un *msb* différent). Or pour formater les variables, il faut connaître le format commun et donc connaître m_z qui dépend de la présence d'une retenue ou non. Pour contourner ce cercle vicieux, nous proposons l'algorithme 2. Cet algorithme consiste dans un premier temps à déterminer un format commun *a priori* où le *msb* commun est le plus grand *msb* des deux opérandes. Ensuite, les bornes des intervalles des opérandes sont arrondies sur ce format commun et on en déduit une première estimation des bornes de l'intervalle somme. Pour finir, une évaluation *a posteriori* du format idéal de la somme est effectuée.

Algorithme 2: Choix du format commun pour l'addition $z = x + y$

Calcul *a priori* du format commun

$\widetilde{m}_z \leftarrow \max(m_x, m_y)$;
$\widetilde{\ell}_z \leftarrow \min(\ell_x, \ell_y)$;
$\widetilde{w}_z \leftarrow \widetilde{m}_z - \widetilde{\ell}_z + 1$;
Si $\widetilde{w}_z > w_{fix}$ **Alors**
  $\widetilde{\ell}_z \leftarrow \widetilde{m}_z - w_{fix} + 1$;
  $\widetilde{w}_z \leftarrow \widetilde{m}_z - \widetilde{\ell}_z + 1$;
Fin

Arrondi (au plus proche ou par troncature) sur le $\widetilde{\ell}_z$-ième bit des bornes de x et de y, en utilisant l'opérateur $\diamond_{\widetilde{\ell}_z}(\cdot)$ (voir 2.4.1 pour les arrondis virgule fixe)

$\underline{x}' \leftarrow \diamond_{\widetilde{\ell}_z}(\underline{x})$;
$\overline{x}' \leftarrow \diamond_{\widetilde{\ell}_z}(\overline{x})$;
$\underline{y}' \leftarrow \diamond_{\widetilde{\ell}_z}(\underline{y})$;
$\overline{y}' \leftarrow \diamond_{\widetilde{\ell}_z}(\overline{y})$;

Calcul *a priori* des bornes de z

$\underline{z} \leftarrow \underline{x}' + \underline{y}'$;
$\overline{z} \leftarrow \overline{x}' + \overline{y}'$;

Calcul du format commun *a posteriori*. $MSB(x)$ est l'opération consistant à déterminer le *msb* de x.

$m_z \leftarrow \max\left(MSB(\underline{z}), MSB(\overline{z})\right)$;
Si $m_z - \widetilde{\ell}_z + 1 \leqslant w_{fix}$ **Alors**
  $\ell_z \leftarrow \widetilde{\ell}_z$;
Sinon
  $\ell_z \leftarrow m_z - w_{fix} + 1$;
Fin

À la fin de l'algorithme on constate qu'il y a une condition pour le calcul *a posteriori* de ℓ_z. En effet, si on calcule ℓ_z par $\ell_z = m_z - w_{fix} + 1$ dans tous les cas, et si le format calculé est plus petit que la largeur fixée (par exemple si la somme exacte peut se faire sur 12 bits alors que l'opérateur a une largeur fixée à 16), on calculera la somme uniquement sur les bits de poids fort (le calcul de la somme sera "collé" à gauche) et les bits de poids faible seront inutilisés (et auront pour valeur 0). Si dans ce cas de figure on calcule ℓ_z par $\ell_z = \min(\ell_x, \ell_y)$, alors l'opérateur utilisera uniquement les bits de poids faible (le calcul de la somme sera "collé" à droite) et la potentielle retenue sera représentable sur le reste des bits disponibles.

L'estimation de $\widetilde{m}_z$ à l'étape 1 nous donne la borne minimale de m_z, en fait on a $m_z \in \{\widetilde{m}_z, \widetilde{m}_z + 1\}$ selon s'il y a une retenue ou non.

Dans le cas d'un schéma d'addition à largeur fixée $w+w \rightarrow w$ (cas logiciel), si $m_z = \widetilde{m}_z$ alors le format final est (m_z, ℓ_z), et si $m_z = \widetilde{m}_z + 1$ alors le format commun des opérandes est (m_z, ℓ_z) mais le format du résultat dépend de sa largeur $w_z = m_z - \ell_z + 1$:

- si $w_z < w_{fix}$: la retenue est représentable sur w_{fix} et le résultat a pour format $(m_z + 1, \ell_z)$,
- si $w_z = w_{fix}$: la retenue n'est pas représentable sur w_{fix} et le résultat a pour format (m_z, ℓ_z). La potentielle retenue significative est alors envoyée dans un registre à part et peut être utilisée si l'architecture le permet.

En matériel, dans le cas d'un schéma $w + w \rightarrow w + 1$, si $m_z = \widetilde{m}_z + 1$ alors le résultat prend en compte la retenue sans arrondir le bit en position ℓ_z, c.-à-d. l'additionneur retourne $w_z + 1$ bits.

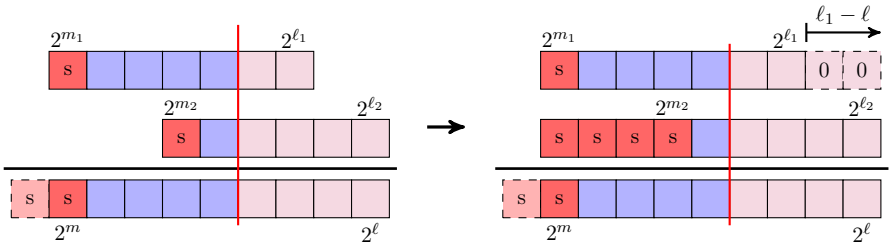
Largeur et format du résultat fixés : Nous avons vu dans le chapitre 1 qu'il est possible de connaître l'intervalle de sortie d'un filtre, et donc le format virgule fixe de cette sortie. Nous verrons comment cette information va imposer non pas une largeur mais un format sur le résultat de nos additions (figure 2.15c). Dans ce cas là, il n'y a rien à faire à cette étape, la question sera de savoir si le résultat de la somme est représentable ou non sur le format fixé.

2.3.2.2 Mise au format commun des opérandes

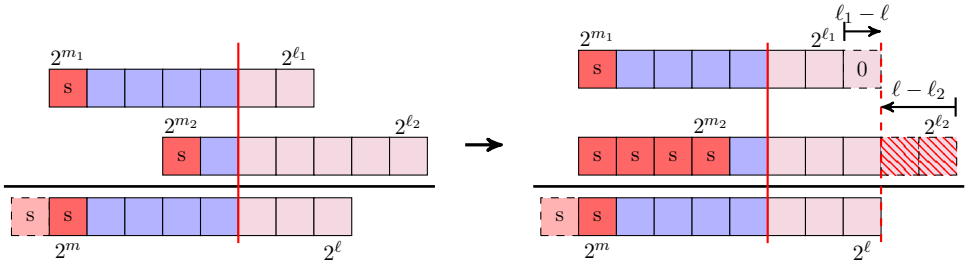
Une fois que l'on connait le format commun, il faut aligner les deux opérandes sur ce format. Pour cela, il suffit de formater chaque opérande sur ce format, selon la cible (matérielle ou logicielle) et le mode d'arrondi. Le formatage sur un format virgule fixe a été décrit à la section 5.2 pour une constante, mais peut s'appliquer à une variable x. En effet, x est définie sur un intervalle virgule fixe avec un format connu, mais les valeurs que prendra x dans cet intervalle ne seront connues que lors de l'exécution du programme. Formater x sur un format donné revient alors à décrire les opérations de formatage qui

seront effectuées sur x à l'exécution, et ces opérations sont les mêmes pour toute valeur de l'intervalle de x.

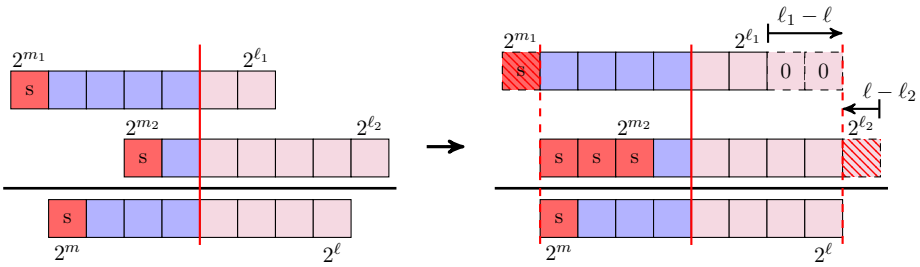
On a vu que ces opérations de formatage sur les opérandes peuvent produire des arrondis, et donc des erreurs, dont on pourra borner la valeur (voir 2.4.2).



(a) Largeur optimale (dans cette illustration on suppose $m = m_1$).



(b) Largeur fixée (dans cette illustration on suppose $m = m_1$).



(c) FPF fixé.

FIGURE 2.15 – Illustration de la somme de deux opérandes dans trois différents cas.

2.3.2.3 Calcul de la somme

Après avoir déterminé le format commun pour les opérandes et avoir formaté ceux-ci sur ce format, il n'y a plus qu'à calculer la somme en utilisant l'arithmétique d'intervalle. En effet, on rappelle que nos opérandes sont définis sur des intervalles et que notre but est de décrire un algorithme virgule fixe, c'est-à-dire de préciser les formats de chaque opération intermédiaire de notre algorithme. Par conséquent, puisque le résultat de l'addition qu'on souhaite

décrire peut être un opérande d'une autre opération, il est nécessaire de calculer l'intervalle résultat de notre addition pour décrire l'opération suivante, en utilisant l'équation (2.29).

En matériel, et en adoptant un schéma d'addition $w + w \rightarrow w + 1$ (ou plus généralement avec trois largeurs différentes), l'addition des intervalles des variables x et y se déroule exactement comme dans l'équation (2.29) et on obtient :

$$z \in [\underline{x} + \underline{y}; \overline{x} + \overline{y}] \tag{2.36}$$

sur le format (m_z, ℓ_z) (ceci est vrai car les variables sont considérées comme formatées sur le FPF commun).

En logiciel, sur un schéma d'addition $w + w \rightarrow w$, l'addition va se dérouler de la même façon sur le format commun des deux opérandes, mais s'il y a une retenue seuls les w_z bits les moins significatifs (c'est-à-dire tous sauf le bit de retenue) seront renvoyés, la retenue étant quant à elle renvoyée dans un registre à part.

Enfin, si le FPF est imposé, alors si au moins un des opérandes est formaté sur ses bits les plus significatifs (par exemple si $m_x > m_z$), alors la probabilité d'une retenue significative n'est pas négligeable et donc le résultat ne sera pas représentable sur le format imposé. La solution est alors de formater le résultat sur le format final pour assurer sa représentabilité sur ce dernier.

Considérons maintenant un exemple de somme selon nos différents cas.

Exemple 2.6. *Dans cet exemple l'arrondi par troncature est considéré. Soient deux variables virgule fixe x et y telles que le FPF de x est $(m_x, \ell_x) = (6, -3)$ et le FPF de y est $(m_y, \ell_y) = (3, -6)$. Les variables sont supposées définies sur l'intervalle induit par leurs FPF respectifs, c'est-à-dire $[\underline{x}; \overline{x}] = [-64, 63.875]$ et $[\underline{y}; \overline{y}] = [-8, 7.984375]$. On considère alors trois cas différents, le premier où la largeur du résultat n'est pas fixée par l'opérateur, le deuxième où la largeur est fixée mais pas le FPF, et le dernier où le FPF du résultat est fixé.*

- *largeur non fixée : on détermine alors le format optimal d'après (2.30) et (2.31), on obtient alors $(m_z, \ell_z) = (7, -6)$. On formate alors les deux variables sur le format optimal et on calcule la somme des deux intervalles par (2.29), ce qui donne :*

$$z \in [-72; 71.859375]_{(7,-6)}$$

- *largeur fixée : on suppose que la largeur est fixée et qu'on a $w = 8$, et on applique la méthode appropriée à une cible logicielle. On a $\widetilde{m}_z = 6$ et donc $\widetilde{\ell}_z = -1$. On arrondit les variables sur ce format, on obtient :*

$$\begin{aligned} x &\in [-64; 63.5]_{(6,-1)} \\ y &\in [-8; 7.5]_{(6,-1)} \end{aligned}$$

  On obtient comme somme intermédiaire des intervalles, l'intervalle suivant :

$$z \in [-72; 71].$$

Les bornes ne sont pas représentables sur le format $(6, -1)$*, en effet en calculant maintenant* m_z *avec l'algorithme 2 on a* $m_z = 7$ *et donc* $\ell_z = 0$*. Le format commun est donc* $(7, 0)$*, et une fois l'intervalle résultat formaté sur ce format, on a :*

$$z \in [-72; 71]_{(7,0)}$$

(ici les deux bornes sont représentables sur $(7, 0)$ *donc elles ne changent pas, mais le pas de quantification de l'intervalle résultat passe de* 0.5 *à* 1*).*

- *format fixé : on suppose que le FPF du résultat est fixé et qu'on a* $(m_z, \ell_z) = (5, -2)$*. On formate alors les deux variables sur ce format. La variable* x *n'est pas représentable sur ce format (car* $m_x > m_z$*), en utilisant l'arithmétique modulaire on a alors*

$$\begin{aligned} x &\in [-32; 31.75]_{(5,-2)} \\ y &\in [-8; 7.75]_{(5,-2)} \end{aligned}$$

La somme des deux intervalles donne :

$$z \in [-40; 39.5],$$

intervalle qui n'est pas représentable sur $(5, -2)$*, donc le résultat est formaté, toujours en utilisant l'arithmétique modulaire, et on obtient :*

$$z \in [-32; 31.75]_{(5,-2)}.$$

On suppose maintenant, à titre d'exemple, qu'au moment de l'exécution, les variables x *et* y *prennent pour valeur* $x = 36.125$ *et* $y = -5.828125$*, représentables sur leur format respectif.*

La figure 2.16 illustre ces trois différentes sommes. Les résultats obtenus pour $x + y$ *dans les trois cas sont (dans chaque cas les variables* x *et* y *sont formatées sur les formats déterminés ou imposés) :*

- *cas optimal :* $x + y = 36.125 - 5.828125 = 30.296875$*.*
- *cas largeur fixée :* $x + y = 36 - 6 = 30$*.*
- *cas format imposé : ici il est plus délicat de passer par la représentation décimale de* x *et* y*. En effet,* $x = 36.125$ *formaté sur le format* $(5, -2)$ *donne* -28 *(le bit de signe est supprimé et le bit le plus significatif suivant est alors interprété comme nouveau bit de signe, ce qui rend* x *négatif). Ensuite,* $-28 - 6 = -34$ *qui, de la même façon, formaté sur le format* $(5, -2)$*, est réinterprété en* 30*. Donc* $x + y = 30$*.*

2.3.3 Multiplication virgule fixe

Intéressons-nous maintenant à une autre opération arithmétique usuelle, la multiplication, et à comment celle-ci se déroule dans différents cas.

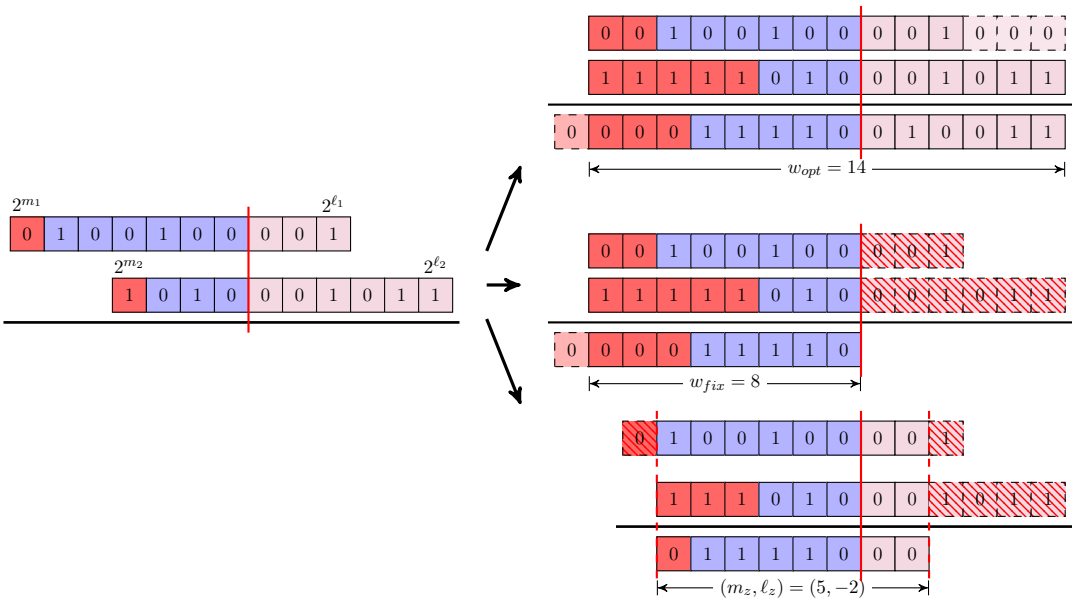


Figure 2.16 – Exemple d'une somme en considérant trois cas différents.

La multiplication considérée ici est la multiplication entre une constante FxP c et une variable FxP v. On appelle x la variable produit de c et de v. La constante c a pour format (m_c, ℓ_c) et la variable v est définie sur l'intervalle virgule fixe $[\underline{v}; \overline{v}]_{(m_v,\ell_v)}$. La variable produit x quant à elle est définie sur l'intervalle FxP $[\underline{x}; \overline{x}]_{(m_x,\ell_x)}$ que l'on souhaite déterminer.

Contrairement à l'addition, les opérandes d'une multiplication ne doivent pas nécessairement partager un format commun, on peut alors considérer deux cas dans notre description de la multiplication :

- le cas matériel : la largeur du résultat n'est pas fixée et donc on considère la largeur optimale du résultat,
- le cas logiciel : la largeur du résultat est fixée et est soit égale à la largeur maximale, soit inférieure à celle-ci et dans ce cas seuls les bits de poids fort sont renvoyés.

2.3.3.1 Cas matériel

Si la largeur du résultat n'est pas fixée par le multiplieur, alors on effectue le calcul sur la largeur dans le pire des cas, notée w_{pc} (**p**ire **c**as), obtenue par la somme de la largeur de la constante, w_c, et celle de la variable, w_v, c'est-à-dire

$$w_{pc} = w_c + w_v. \tag{2.37}$$

De plus, le format optimal pour cette largeur s'obtient par :

$$m_x = m_c + m_v + 1, \tag{2.38}$$

$$\ell_x = \ell_c + \ell_v. \tag{2.39}$$

En fait ce pire cas est atteint uniquement dans un cas, quand $c = -2^{m_c}$ et $v = -2^{m_v}$, c'est-à-dire les plus petites valeurs possibles de c et v (et les plus grandes en valeur absolue) représentables sur leurs formats respectifs. En effet, on a alors :

$$x = c \cdot v = -2^{m_c} \times (-2^{m_v}) = 2^{m_c+m_v}. \tag{2.40}$$

La valeur de x correspond au plus petit nombre FxP représentable sur exactement w_{pc} bits au format (m_x, ℓ_x) (voir figure 2.17). Pour toute autre valeur de c et v le résultat x est alors représentable au plus sur $w_{pc} - 1$ bits, il est donc important d'avoir une meilleure évaluation de cette largeur dans le pire des cas (ainsi que du format dans le pire des cas).

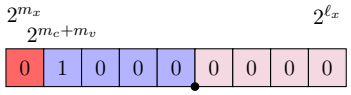


FIGURE 2.17 – Pire cas atteint pour $x = c \cdot v$ avec $c = -2^{m_c}$ et $v = -2^{m_v}$.

Pour connaître la largeur optimale dans le cas matériel, on peut utiliser exactement les mêmes formules décrivant le cas optimal de l'addition (en modifiant l'addition en multiplication et en considérant une constante au lieu d'un intervalle pour l'un des opérandes). En effet, toujours en utilisant l'arithmétique d'intervalle, on calcule le produit de c et v :

$$[\underline{x}; \overline{x}] \triangleq c \times [\underline{v}; \overline{v}] = [\min(c \times \underline{v}, c \times \overline{v}); \max(c \times \underline{v}, c \times \overline{v})] \tag{2.41}$$

Ensuite, de la même façon, m_x est déterminé par le plus grand msb entre celui de $\underline{x}$ et celui de $\overline{x}$, c.-à-d. :

$$m_x \triangleq \max(m_{\underline{x}}, m_{\overline{x}}). \tag{2.42}$$

Pour finir, l'évaluation de ℓ_x ne change pas par rapport à la formule énoncée précédemment, on a :

$$\ell_x \triangleq \ell_c + \ell_v. \tag{2.43}$$

On a donc $x \in [\underline{x}; \overline{x}]_{(m_x, \ell_x)}$.

L'algorithme de multiplication binaire en complément à deux fonctionne de la même façon que la multiplication usuelle en base 10 que nous connaissons tous : chaque chiffre du multiplieur calcule un produit partiel avec le multiplicande, et tous les produits intermédiaires sont décalés successivement puis sommés pour obtenir le résultat final. La figure 2.18 illustre l'algorithme de multiplication entre une constante c et une variable v. Cet algorithme s'applique à des entiers, donc les calculs intermédiaires seront exprimés à partir de C et V, les mantisses respectives de c et v (sur la figure 2.18, les produits intermédiaires sont représentés par c et non C dans un souci de lisibilité uniquement). Les produits intermédiaires sont alors calculés par :

$$C \times V_i = \begin{cases} C \ll i \text{ si } V_i = 1 \\ 0 \text{ sinon} \end{cases} \quad \text{pour } i = 0, \ldots, w_v - 2 \tag{2.44}$$

(on rappelle que $V = -2^{w_v-1} + \sum_{i=0}^{w_v-2} V_i$). Le dernier produit intermédiaire est calculé par $-C \ll w_v-1$ si $v < 0$ (si $v > 0$ alors le dernier produit intermédiaire est nul), où $-C$ représente la mantisse de $-c$, le complément à deux de c. On peut noter que si $c = -2^{m_c}$ alors il faut un bit de plus pour représenter $-c$, *a contrario*, si $c = 2^{m_c} - 2^{\ell_c}$, il faut un bit de moins pour représenter $-c$.

La somme des produits intermédiaires $C \times V_i$ donne la mantisse X du résultat x, ce dernier étant obtenu par $x = X \cdot 2^{\ell_x}$.

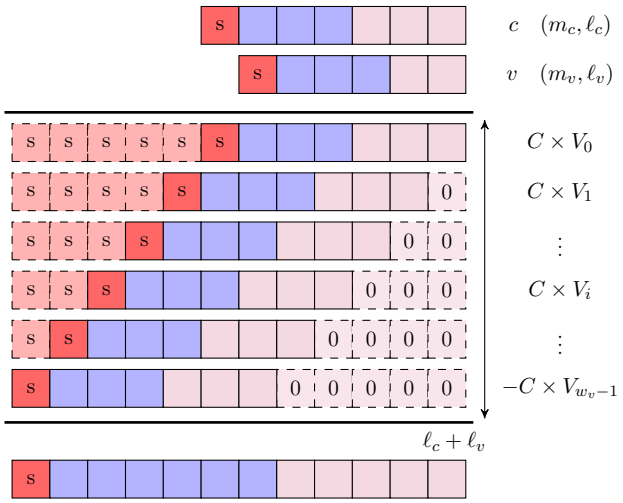


Figure 2.18 – Illustration de la multiplication en virgule fixe.

Exemple 2.7. *Soient une constante virgule fixe* $c = -3.75$ *au format* $(m_c, \ell_c) = (2, -2)$ *et une variable virgule fixe* v *de format* $(m_v, \ell_v) = (2, -1)$ *avec* $[\underline{v}; \overline{v}] = [-4; 3.5]$. *En utilisant l'arithmétique d'intervalle on a :*

$$[\underline{x}; \overline{x}] = [-10.125; 15]. \tag{2.45}$$

Le format du résultat x *est alors, en considérant le produit sur la largeur optimale,* $(m_x, \ell_x) = (4, -3)$. *Supposons maintenant qu'au moment de la multiplication* v *prend la valeur* -2.5. *Le produit* x *de* c *par* v *est illustré par la figure 2.19 en appliquant l'algorithme décrit précédemment, et on obtient* $x = 9.375$.

Remarque 2.7. *On a décrit ici uniquement l'algorithme classique de multiplication binaire, en pratique, ce n'est pas forcément cet algorithme qui est utilisé. En effet, de nombreuses méthodes de multiplication existe, on peut citer l'algorithme de Booth [10], les arbres de Wallace [106], la méthode plus récente dite des tas de bits [11], etc. De plus, dans [11], de Dinechin* et al. *montrent que l'extension des bits de signes (figure 2.19) ne rajoute pas de bits dans leur méthode de calcul.*

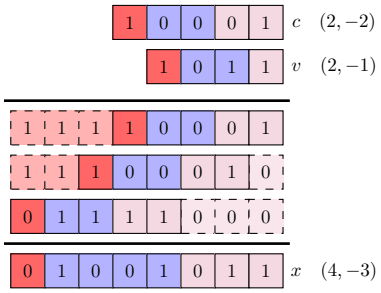


FIGURE 2.19 – Exemple de multiplication virgule fixe.

2.3.3.2 Cas logiciel

En logiciel, on peut distinguer deux types de multiplieurs :

- les multiplieurs dont la largeur du résultat est égale à la somme des largeurs des opérandes, dont les schémas de multiplication (par analogie avec les schémas utilisés pour l'addition) peuvent s'écrire $w \times w \to 2w$ (typiquement $8 \times 8 \to 16$, $16 \times 16 \to 32$, etc), mais aussi $w \times w' \to w + w'$ (par exemple $8 \times 16 \to 24$) ;
- les multiplieurs dont la largeur du résultat est inférieure à la somme des largeurs des opérandes, et généralement égale à la largeur des opérandes. Le schéma de multiplication classique de ce type d'opérateur est $w \times w \to w$ ($16 \times 16 \to 16$, etc).

Dans le premier cas, la largeur du résultat, w_z, correspond à la largeur dans le pire des cas décrite par l'équation (2.37), c'est à dire le cas optimal avec possiblement une répétition du bit de signe (on a en fait $w_z \geqslant w_{opt}$). Sur un tel opérateur, le format sera déterminé de la même façon que dans le pire cas vu précédemment, c'est-à-dire d'après les équations (2.38) et (2.39), et la multiplication se déroulera aussi de la même façon que celle décrite précédemment.

Dans le second cas, la largeur du résultat est inférieure au cas optimal, le calcul se déroule de la même façon que dans le pire cas mais seuls les w_z bits de poids fort du résultats sont renvoyés (avec les potentielles répétitions de signe).

2.4 Quantification

Dans chaque processus de conversion ou dans chaque calcul en précision finie, il y a une étape de quantification, ou d'arrondi. Jusqu'ici nous avons utilisé des fonctions pour arrondir nos résultats (voir les opérateurs $\lfloor \cdot \rceil$, $\lceil \cdot \rceil$ et $\lfloor \cdot \rfloor$ dans l'algorithme 1 par exemple) sans analyser ce que cela impliquait sur notre constante convertie ou sur le résultat d'une opération.

2.4.1 Lois de quantification

Il existe plusieurs façons d'arrondir une valeur, on parle alors de *lois de quantification* (ou parlera aussi de *modes d'arrondi*). Dans ces travaux nous considérerons principalement deux lois de quantifications, la troncature et l'arrondi au plus proche, et nous définirons également l'arrondi vers $+\infty$ et l'arrondi fidèle.

Un arrondi se définit par rapport à un pas de quantification.

2.4.1.1 Troncature (ou arrondi vers $-\infty$)

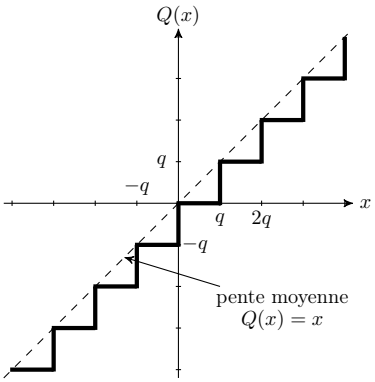


Figure 2.20 – Arrondi par troncature. $Q(x)$ est l'opération de quantification de $x \in \mathbb{R}$ et q le pas de quantification.

Remarque 2.8. *Le terme troncature ici est un abus de langage. En effet le terme troncature signifie, dans la représentation décimale usuelle des nombres, l'action de tronquer les décimales d'un nombre, ainsi* -0.6 *est arrondi vers* 0 *et non vers* -1 *(on parle d'arrondi vers 0). En représentation binaire en complément à deux (et donc en virgule fixe), tronquer des bits revient à prendre l'arrondi vers* $-\infty$ *(comme nous allons le voir), c'est pourquoi on utilisera ce terme dans cette partie pour parler d'arrondi vers* $-\infty$*.*

L'arrondi par troncature d'une constante c suivant un pas de quantification q, consiste à se ramener au plus grand multiple de q inférieur à c (voir figure 2.20). En effet, toute constante $c \in \mathbb{R}$ peut être encadrée par deux multiples consécutifs de q, c'est-à-dire :

$$\forall c \in \mathbb{R}, \exists! k \in \mathbb{Z} \text{ tel que } \quad kq \leqslant c < (k+1)q,$$

et dans ce cas l'arrondi par troncature de c est kq.

En arithmétique binaire, le pas de quantification q est une puissance de 2. L'arrondi par troncature suivant un pas de quantification $q = 2^k$ consiste

alors, dans la décomposition binaire de la constante c, à supprimer (tronquer) tous les bits dont la position est strictement plus petite que k.

Pour une constante c, l'arrondi par troncature suivant q consiste à calculer la partie entière inférieure de $c \cdot q^{-1}$ et de re-multiplier le résultat par q, c'est-à-dire :

$$Q(c) = \left\lfloor \frac{c}{q} \right\rfloor \cdot q \tag{2.46}$$

Exemple 2.8. *Soit une constante réelle $c = 42.5$ et un pas de quantification $q = \frac{1}{3}$. L'arrondi par troncature de c suivant q est obtenu en appliquant (2.46) :*

$$Q(42.5) = \lfloor 42.5 \cdot 3 \rfloor \cdot \frac{1}{3} = \lfloor 127.5 \rfloor \cdot \frac{1}{3} = \frac{127}{3} = 42.333\ldots$$

Pour la suite du rapport, on définit l'*arrondi virgule fixe par troncature* par rapport au d-ième bit.

Définition 2.4 (Arrondi virgule fixe par troncature)**.** *Soient une constante réelle $c \in \mathbb{R}$, et d un entier. L'arrondi virgule fixe par troncature de c, noté $\nabla_d(c)$, est le plus grand nombre virgule fixe dont le lsb est d et qui est plus petit que c. Il s'obtient en calculant :*

$$\nabla_d(c) \triangleq \left\lfloor c \cdot 2^{-d} \right\rfloor \cdot 2^d \tag{2.47}$$

De plus, d'après la décomposition binaire en complément à deux d'une constante c (équation (2.9)), on a :

$$\nabla_d(c) = -c_m \cdot 2^m + \sum_{i=d}^{m-1} c_i 2^i. \tag{2.48}$$

Exemple 2.9. *Considérons la conversion en virgule fixe de $c = \sqrt{2}$ sur $w = 8$ bits avec l'arrondi virgule fixe par troncature. La décomposition en complément à 2 de x est donnée par :*

$$c =_2 01.011010100000100111100110011010\ldots \tag{2.49}$$

La position du bit le plus significatif, m, est égale à 1, donc on a d'après (2.8), $\ell = m - w + 1 = -6$. On prend alors $2^\ell = 2^{-6}$ comme pas de quantification. On obtient alors l'arrondi par troncature de c par rapport au ℓ-ième bit, à partir de (2.47) :

$$\nabla_\ell(c) = \left\lfloor \sqrt{2} \cdot 2^6 \right\rfloor \cdot 2^{-6} = 90 \cdot 2^{-6} = 1.40625 < \sqrt{2}$$

Remarque 2.9. *Il est important de noter ici que l'arrondi par troncature n'est pas symétrique. En effet, on l'a dit, l'arrondi par troncature en complément à deux correspond à un arrondi vers $-\infty$, que ce soit pour les nombres positifs ou négatifs.*

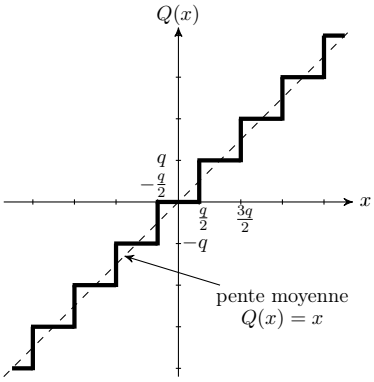


Figure 2.21 – Arrondi au plus proche. $Q(x)$ est l'opération de quantification de $x \in \mathbb{R}$ et q le pas de quantification.

2.4.1.2 Arrondi au plus proche

L'arrondi au plus proche d'une constante c suivant un pas de quantification q, consiste à se ramener au multiple de q le plus proche c (voir figure 2.21).

Soit $k \in \mathbb{Z}$ tel que $kq \leqslant c < (k+1)q$. Si c est plus proche de kq que de $(k+1)q$, c'est-à-dire si $c - kq < (k+1)q - c$, alors $Q(c) = kq$, sinon $Q(c) = (k+1)q$. Par convention le cas $c - kq = (k+1)q - c$ fera partie de ce dernier cas, on parle alors d'arrondi au plus proche vers $+\infty$.

Par analogie avec l'arrondi par troncature, l'arrondi au plus proche suivant q consiste, pour une constante c, à calculer l'entier le plus proche de $c \cdot q^{-1}$ et de re-multiplier le résultat par q, c'est-à-dire :

$$Q(c) = \left\lfloor \frac{c}{q} \right\rceil \cdot q \tag{2.50}$$

Exemple 2.10. *Reprenons la constante réelle $c = 42.5$ et le pas de quantification $q = \frac{1}{3}$. L'arrondi au plus proche de c suivant q est obtenu en appliquant (2.50) :*

$$Q(42.5) = \lfloor 42.5 \cdot 3 \rceil \cdot \frac{1}{3} = \lfloor 127.5 \rceil \cdot \frac{1}{3} = \frac{128}{3} = 42.666\ldots$$

On définit également l'*arrondi virgule fixe au plus proche* par rapport au d-ième bit.

Définition 2.5 (Arrondi virgule fixe au plus proche)**.** *Soient une constante réelle $c \in \mathbb{R}$, et d un entier. L'arrondi virgule fixe au plus proche de c, noté $\circ_d(c)$, est le nombre virgule fixe dont le lsb est d et qui est le plus proche de c. Il s'obtient en calculant :*

$$\circ_d(c) \triangleq \left\lfloor c \cdot 2^{-d} \right\rceil \cdot 2^d. \tag{2.51}$$

De plus, d'après la décomposition binaire en complément à deux d'une constante c (équation (2.9)), on a :

$$\circ_d(c) = -c_m \cdot 2^m + \sum_{i=d}^{m-1} c_i 2^i + c_{d-1} 2^d \tag{2.52}$$

Remarque 2.10. *Dans l'équation (2.52), le terme $c_{d-1}2^d$ s'explique par le fait que c'est précisément ce bit (le bit le plus significatif parmi ceux arrondis) qui va donner l'arrondi soit vers le haut soit vers le bas (si $c_{d-1} = 1$ on arrondit vers le haut, si $c_{d-1} = 0$ on arrondi vers le bas). On ajoute donc ce bit au d-ième bit c_d et on tronque les bits dont la position est inférieure à d. On a alors :*

$$\circ_d(c) = \nabla_d(c + c_{d-1}2^d) = \left(\left\lfloor \frac{c}{2^d} \right\rfloor + c_{d-1}\right) \cdot 2^d.$$

De plus, en matériel, ce bit peut souvent être ajouté gratuitement (voir par exemple les multiplications par une constante de [11]).

Exemple 2.11. *Considérons la conversion en virgule fixe de $c = \sqrt{2}$ sur $w = 8$ bits avec l'arrondi virgule fixe au plus proche. Cela correspond en fait à appliquer l'algorithme 1. On obtient ainsi :*

$$\circ_\ell(c) = \left\lfloor \sqrt{2} \cdot 2^6 \right\rceil \cdot 2^{-6} = 91 \cdot 2^{-6} = 1.421875 > \sqrt{2}$$

On remarque qu'on est ici dans le cas $\lfloor x \rceil > \lfloor x \rfloor$ car $x = \sqrt{2} \cdot 2^6 = 90.509668\ldots$, donc $\circ_\ell(c) > \nabla_\ell(c)$.

2.4.1.3 Arrondi vers $+\infty$ et arrondi fidèle

Arrondi vers $+\infty$: L'arrondi vers $+\infty$ d'un nombre $c \in \mathbb{R}$ suivant un pas de quantification q est le contraire de l'arrondi par troncature tel que défini précédemment, c'est-à-dire le plus petit multiple de q plus grand que c.

Il s'obtient en calculant :

$$Q(c) = \left\lceil \frac{c}{q} \right\rceil \cdot q \tag{2.53}$$

On peut définir l'*arrondi virgule fixe vers* $+\infty$ par rapport au d-ième bit.

Définition 2.6 (Arrondi virgule fixe vers $+\infty$)**.** *Soit une constante réelle $c \in \mathbb{R}$, et d un entier. L'arrondi virgule fixe vers $+\infty$ de c, noté $\triangle_d(c)$, est le plus petit nombre virgule fixe dont le lsb est d et qui est plus grand de c. Il s'obtient en calculant :*

$$\triangle_d(c) \triangleq \left\lceil c \cdot 2^{-d} \right\rceil \cdot 2^d. \tag{2.54}$$

Remarque 2.11. *Attention, on ne peut pas écrire* $\triangle_d(c) = \nabla_d(c) + 2^d$. *En effet si* c *est un multiple de* 2^d, *c'est-à-dire si* $c = k \cdot 2^d$, *avec* $k \in \mathbb{Z}$, *alors* $\triangle_d(c) = \nabla_d(c) = c$. *Si* c *n'est pas un multiple de* 2^d *alors on a bien* $\triangle_d(c) = \nabla_d(c) + 2^d$.

De plus, l'arrondi FxP au plus proche est soit l'arrondi FxP par troncature soit l'arrondi FxP vers $+\infty$, *c'est-à-dire si* c *est plus proche de* $\nabla_d(c)$ *que de* $\triangle_d(c)$ *alors* $\circ_d(c) = \nabla_d(c)$, *et sinon* $\circ_d(c) = \triangle_d(c)$.

Arrondi fidèle : L'arrondi fidèle d'une constante réelle c ne peut pas être défini comme une loi de quantification. On dira que f est un arrondi fidèle de c par rapport au pas de quantification q si f est un des deux multiples de q encadrant c, c'est-à-dire, soit $k \in \mathbb{Z}$ tel que $kq \leqslant c < (k+1)q$, alors kq et $(k+1)q$ sont les deux arrondis fidèles de c par rapport à q.

Si les autres lois de quantification sont déterministes, l'arrondi fidèle lui ne l'est pas. Il a été décrit pour les nombres flottants dans [80] et [92].

On définit tout de même la notion d'arrondi virgule fixe fidèle par rapport au d-ième bit.

Définition 2.7 (Arrondi virgule fixe fidèle)**.** *Soient une constante réelle* $c \in \mathbb{R}$, d *un entier et* $\star_d()$ *un opérateur de quantification dont le pas est* 2^d. $\star_d(c)$ *est dit arrondi virgule fixe fidèle de* c, *si sa valeur est l'un des deux nombres virgule fixe consécutifs dont le lsb est égal à* d, *c'est-à-dire* $\star_d(c)$ *est un arrondi virgule fixe fidèle de* c *si :*

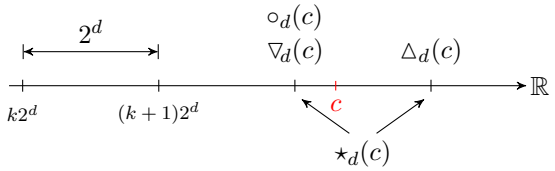
$$\star_d(c) = \triangle_d(c) \quad ou \quad \star_d(c) = \nabla_d(c). \tag{2.55}$$

La figure 2.22 résume l'ensemble des modes d'arrondi décrits pour une constante $c \in \mathbb{R}$.

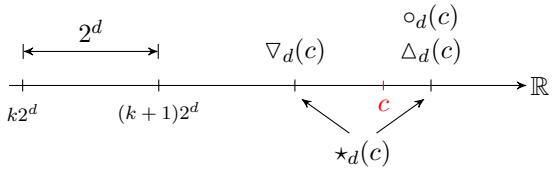
2.4.2 Calcul de l'erreur

Maintenant que nous avons décrit les différentes lois de quantification, nous pouvons nous intéresser à comment estimer une valeur de l'erreur commise lors du processus de quantification. Il faut cependant distinguer deux différents types d'erreur, les *erreurs de paramétrisation* (ou de représentation) et les *erreurs de calcul*. Les erreurs de paramétrisation correspondent aux erreurs d'arrondi commises lors d'une conversion d'une constante réelle en virgule fixe. Les erreurs de calcul quant à elles sont les erreurs d'arrondi apparaissant dans les calculs en virgule fixe, et correspondent aux décalages effectués lors de ces calculs pour aligner les virgules.

Une première façon de donner une valeur à une erreur d'arrondi est de calculer l'erreur absolue et l'erreur relative. L'inconvénient de ces deux erreurs est qu'elles nécessitent de connaître la valeur réelle, or si cela est facile lors de la conversion d'une constante, cela l'est beaucoup moins lors d'une suite d'opérations arithmétiques (il faudrait alors simuler le calcul réel pour



(a) Cas où $\circ_d(c) = \nabla_d(c)$.



(b) Cas où $\circ_d(c) = \triangle_d(c)$.

FIGURE 2.22 – Illustration des différents modes d'arrondi d'une constante réelle c.

comparer les résultats réel et fixe). Toutefois, cette opération est possible en utilisant des certificats `Gappa`[4]`http://gappa.gforge.inria.fr/` (Génération Automatique de Preuves de Propriétés Arithmétiques) [78].

Nous verrons ensuite deux autres façons de manipuler l'erreur, basées sur un principe similaire, celui de modéliser une erreur de quantification comme l'ajout d'une dégradation, sous forme d'un intervalle ou d'un bruit, sur le signal (voir figure 2.23).

En effet, en traitement du signal une erreur est modélisée par une perturbation, ou plus précisément à l'ajout d'un bruit blanc sur le signal [96, 107], tandis qu'en arithmétique une erreur est un intervalle contenant toutes les valeurs possibles de cette erreur qui s'ajoute à la valeur de départ.

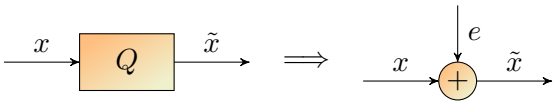


FIGURE 2.23 – Modélisation du processus de quantification. La dégradation e peut être modélisée par un intervalle ou par un bruit.

Puisque les erreurs de paramétrisation proviennent d'une conversion d'une constante réelle, la valeur x à quantifier a une précision infinie, on considère qu'on arrondit une infinité de bits. En traitement du signal on considérera que x est un signal à amplitude continue. À l'inverse, les erreurs de calculs proviennent d'une suppression de bits, d'une conversion d'un nombre x en précision finie en un nombre $\tilde{x}$ également en précision finie. On dira que le signal d'entrée x est à amplitude discrète.

4.

2.4.2.1 Erreurs absolue et relative

Erreur absolue : L'erreur absolue de la quantification d'un nombre x en son quantifié $\tilde{x}$, notée Δx, correspond à la différence des deux valeurs en valeur absolue, c'est-à-dire :

$$\Delta x = |x - \tilde{x}| \tag{2.56}$$

Si $\tilde{x}$ est le résultat de la conversion virgule fixe de x par l'algorithme 1 au format (m, ℓ), alors puisque l'arrondi effectué dans l'algorithme est l'arrondi virgule fixe au plus proche par rapport au ℓ-ième bit, on a :

$$\Delta x \leqslant 2^{\ell-1} \tag{2.57}$$

Dans le cas d'un arrondi par troncature ou d'un arrondi fidèle on aurait eu $\Delta x < 2^{\ell}$.

Erreur relative : Cette erreur correspond à l'erreur effectuée par rapport à la valeur de départ x (pour $x \neq 0$). On la note δx et elle s'obtient en divisant l'erreur absolue par la valeur absolue de x :

$$\delta x = \frac{|x - \tilde{x}|}{|x|} \tag{2.58}$$

On peut montrer qu'on a $\delta x < 2^{w-1}$.

Exemple 2.12. *Reprenons une situation vue précédemment. Soit $c = \sqrt{2}$ une constante réelle que l'on convertit en virgule fixe sur $w = 8$ bits par l'algorithme 1, on obtient alors $\tilde{c} = 1.40625$. Les erreurs absolues et relatives sont données par :*

$$\Delta c = \left|\sqrt{2} - 1.421875\right| = 0.0076614$$
$$\delta c = \frac{\left|\sqrt{2} - 1.421875\right|}{\left|\sqrt{2}\right|} = 0.0054175$$

L'erreur commise lors de la quantification vaut 0.0076614, ce qui correspond à 0.0054175 fois la valeur de x (ou 0.54175% de x), et on vérifie bien que Δc est plus petit que $2^{\ell-1} = 2^{-7} = 0.0078125$.

2.4.2.2 Bruit de quantification

La modélisation du bruit de quantification consiste à voir la quantification d'un signal x comme l'ajout d'un bruit blanc uniformément distribué e sur ce signal. Il est alors possible [96, 107] de déterminer les moments d'ordre un et deux, respectivement la moyenne μ_e et la variance σ_e^2 de ce bruit.

Dans le cas d'un signal à amplitude discrète (erreurs de calcul), la quantification génère un bruit également à amplitude discrète, qui n'a donc pas les

mêmes caractéristiques que le bruit à amplitude continue. Le modèle décrit pour le bruit à amplitude discrète a été proposé par Constantinides *et al.* dans [18].

Pour décrire les moments d'ordre un et deux, dans le cas d'un bruit continu et dans le cas d'un bruit discret, on prend un pas de quantification égal à 2^ℓ et que le nombre de bits arrondis (pour le bruit discret) est égal à d. On obtient ainsi le tableau 2.3.

Moments	Arrondi			
	par troncature		au plus proche	
	continu	discret	continu	discret
μ_e	$2^{\ell-1}$	$2^{\ell-1}(1-2^{-d})$	0	$2^{\ell-d-1}$
σ_e^2	$2^{2\ell}/12$	$2^{2\ell}/12(1-2^{-2d})$	$2^{2\ell}/12$	$2^{2\ell}/12(1-2^{-2d})$

TABLE 2.3 – Moments d'ordre un et deux pour un bruit e pour un pas de quantification 2^ℓ et d bis pour le nombre de bits arrondis pour le bruit discret.

Cette modélisation, statistique, est traditionnellement celle utilisée en traitement numérique du signal. Elle ne donne pas les valeurs précises que peut prendre l'erreur e, elle donne juste la moyenne et la variance qui sert à caractériser la dispersion statistique de l'erreur autour de cette moyenne. Une autre approche, celle utilisée en arithmétique des ordinateurs, est à l'inverse une approche déterministe et donne les intervalles de valeurs précis de l'erreur.

2.4.2.3 Intervalles d'erreur

Cette approche donne les intervalles précis de valeurs que peut prendre l'erreur additive.

On rappelle que e est une erreur additive sous forme d'un intervalle, c'est-à-dire $\tilde{x} = x+e$ avec $e = [\underline{e}, \overline{e}]$. Pour déterminer $\underline{e}$ et $\overline{e}$, on retourne le problème, on définit $\tilde{x}$ un nombre virgule fixe obtenu en quantifiant un x au format (m, ℓ), et on veut savoir quelles erreurs minimale et maximale ont pu être commises pour arriver à ce $\tilde{x}$. On appelle alors $\overline{x}$ et $\underline{x}$ respectivement les plus grande et plus petite valeurs que peut prendre x pour obtenir l'arrondi $\tilde{x}$, on a alors :

$$\begin{cases} \underline{e} = \tilde{x} - \overline{x} \\ \overline{e} = \tilde{x} - \underline{x} \end{cases} \tag{2.59}$$

Si l'arrondi considéré est un arrondi par troncature, la figure 2.24 illustre les pires cas pour x, dans le cas d'un x réel (précision infinie) et dans le cas d'un x virgule fixe (précision finie, décalage de d bits). La valeur b est la valeur du bit $\tilde{x}_\ell$, c'est-à-dire $b \in \mathbb{B}$. En précision finie, dans le meilleur des cas tous les bits tronqués valaient 0, *i.e.* $\tilde{x} = x$, tandis que dans le pires des cas les d bits tronqués valaient 1. En précision infinie, le meilleur cas est le même, l'erreur est nulle car on ne tronque que des bits à 0, et dans le pire des cas, tous les bits

tronqués auraient eu pour valeur 1, mais ce cas n'est pas atteignable (car une suite infinie de 1 en position $\ell - 1$ à $-\infty$ est égale au nombre 2^ℓ). On obtient alors $e =]-2^\ell; 0]$ en précision infinie et $e = [-2^\ell + 2^d; 0]$ en précision finie. Ces résultats, ainsi que ceux obtenus avec l'arrondi au plus proche (voir figure 2.25 où $\overline{b} = 1 - b \in \mathbb{B}$) sont résumés dans le tableau 2.4.

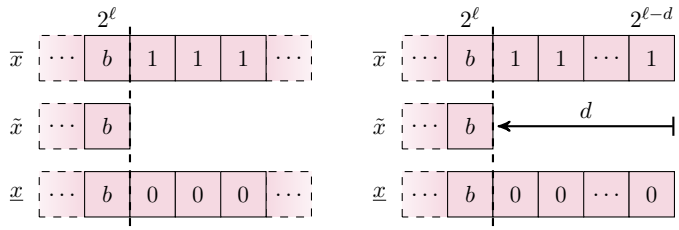


Figure 2.24 – Illustration des pires cas pour l'erreur d'arrondi par troncature.

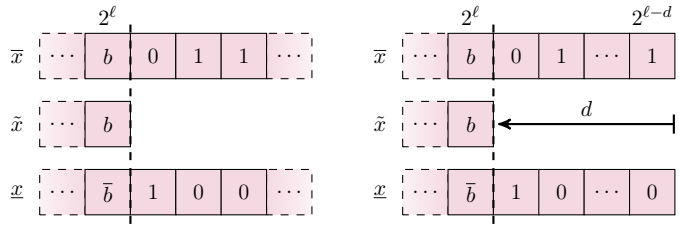


Figure 2.25 – Illustration des pires cas pour l'erreur d'arrondi au plus proche.

Bornes	Arrondi			
	par troncature		au plus proche	
	précision infinie	précision finie	précision infinie	précision finie
$\underline{e}$	$> -2^\ell$	$-2^\ell + 2^{\ell-d}$	$> -2^{\ell-1}$	$-2^{\ell-1} + 2^{\ell-d}$
$\overline{e}$	0	0	$2^{\ell-1}$	$2^{\ell-1}$

Table 2.4 – Bornes de l'intervalle d'erreur e pour un pas de quantification 2^ℓ et d bits pour le nombre de bits arrondis en précision finie.

2.4.2.4 Produit et quantification

Considérons le produit d'une constante par une variable dans le contexte de nos filtres virgule fixe. En effet, dans ce contexte, la variable est un signal virgule fixe qu'on suppose exact (on verra dans le prochain chapitre que les erreurs de calcul amenant à ce signal sont comptabilisées à part), et la constante est l'arrondi virgule fixe d'une constante réelle, ce qui signifie que le dernier bit

de cette constante est potentiellement faux. Compte tenu de cette information, il est intéressant de savoir comment ce bit potentiellement faux va se propager dans le produit.

Soit c la constante virgule fixe au format (m_c, ℓ_c) et v la variable virgule fixe au format (m_v, ℓ_v). En appliquant l'algorithme de multiplication (voir figure 2.26), le bit potentiellement faux, c_{ℓ_c}, est décalé de $w_v - 1$ bits vers la gauche, où w_v est la largeur de v, en partant du *lsb* du produit, égal à $\ell_c + \ell_v$. La position du bit potentiellement faux le plus significatif du produit est alors donnée par :

$$(\ell_c + \ell_v) + (w_v - 1) = \ell_c + m_v. \tag{2.60}$$

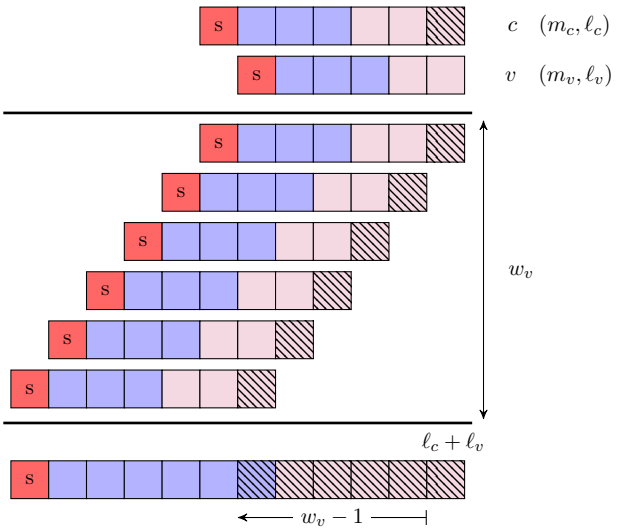


FIGURE 2.26 – Propagation d'une erreur dans une multiplication.

Cette information nous sera principalement utile dans le chapitre 5 lorsque nous calculerons les largeurs des constantes et variables pour ne conserver que les bits exacts de leurs produits.

2.5 Conclusion

Nous avons introduit dans ce chapitre l'arithmétique virgule fixe à travers un formalisme considérant les opérations usuelles comme la conversion d'un nombre réel en un nombre FxP, l'addition, la multiplication par une constante, ou encore la quantification. Ces opérations sont les seules dont nous avons besoin pour décrire l'implémentation d'un filtre linéaire en virgule fixe.

Comme nous l'avons vu en introduction de ce chapitre, cette formalisation de l'arithmétique virgule fixe constitue une première contribution de cette thèse. En effet, si les bases de cette arithmétique sont connues de tous, il

n'existe, à notre connaissance, aucun vrai formalisme pleinement détaillé. Nous avons donc repris ces bases de formalisation afin de proposer un formalisme traitant tous les cas possibles, selon les différents types de cibles (matériel ou logiciel), pour les opérations usuelles propres à notre approche.

On rappelle que dans la suite de ce manuscrit, l'arithmétique utilisée est l'arithmétique virgule fixe signée en complément à deux.

Le prochain chapitre fait le lien entre ces deux premiers chapitres, en proposant les premiers outils d'implémentation virgule fixe pour les filtres linéaires.

Chapitre 3

Vers l'implémentation de filtres linéaires en virgule fixe

Les chapitres précédents ont rappelé d'une part quelques pré-requis concernant les signaux et les filtres linéaires, et proposé d'autre part un formalisme de l'arithmétique virgule fixe. Le but est maintenant, à partir de ces informations, de décrire ce qui se passe quand on veut implémenter un filtre en virgule fixe, à travers l'analyse d'un filtre linéaire. En effet, nous allons proposer dans ce chapitre une analyse de l'évolution d'un signal à travers un filtre, et en déduire des informations utiles pour implémenter ce filtre.

Le chapitre se conclut par la description d'un exemple fil rouge qui sera utilisé dans les chapitres suivants pour illustrer l'approche.

3.1 Évolution d'un signal à travers un filtre

Le premier phénomène à analyser est l'évolution subie par un signal lorsqu'il passe à travers un filtre. En effet, les normes de filtres définies dans la section 1.1.2.5 peuvent permettre d'évaluer le comportement d'une variable aléatoire (on a vu qu'un signal quelconque peut être considéré comme une variable aléatoire) ou d'un intervalle au travers d'un filtre. Ces propositions sont très importantes dans la suite de notre approche. La proposition 3.1 est classique en traitement du signal, tandis que la proposition 3.2 l'est moins, et surtout nouvelle pour l'analyse des erreurs de calcul [6, 45].

Pour une meilleure lisibilité, les intervalles considérés sont exprimés avec la notation centre/rayon ($\langle m, r\rangle$ où m est le centre et r le rayon de l'intervalle).

Proposition 3.1 (Variable aléatoire à travers un filtre)**.** *Soient $\mathcal{H}$ un filtre linéaire MIMO et $\boldsymbol{u}$ et $\boldsymbol{y}$ deux vecteurs de variables aléatoires tels que $\boldsymbol{y} = \boldsymbol{h} * \boldsymbol{u}$ avec $\boldsymbol{h}$ la réponse impulsionnelle de $\mathcal{H}$.*

On suppose de plus que $\boldsymbol{u}(n)$ *et* $\boldsymbol{u}(m)$ *sont décorrélés pour tout* $n \neq m$*, c'est-à-dire*

$$E\left\{\{(\boldsymbol{u}(k-n)-\boldsymbol{\mu_u})(\boldsymbol{u}(k-m)-\boldsymbol{\mu_u})^\top\}_{k\geqslant 0}\right\} = \delta_{n,m}\boldsymbol{\psi_u}, \quad \forall n,m, \tag{3.1}$$

où $\delta_{n,m}$ *désigne le symbole de Kronecker* ($\delta_{n,m} = 1$ *si* $n = m$ *et 0 sinon).*

Si on connait les moments d'ordre un et deux du signal d'entrée $\boldsymbol{u}$*, respectivement* $\boldsymbol{\mu_u}$ *et* $\boldsymbol{\psi_u}$*, alors les moments d'ordre un et deux de* $\boldsymbol{y}$*, notés respectivement* $\boldsymbol{\mu_y}$ *et* $\boldsymbol{\sigma_y^2}$*, sont donnés par :*

$$\boldsymbol{\mu_y} = \langle\!\langle\mathcal{H}\rangle\!\rangle_{\mathrm{DC}} \cdot \boldsymbol{\mu_u}, \tag{3.2}$$

$$\boldsymbol{\sigma_y^2} = \|\mathcal{H}\cdot\boldsymbol{\varphi_u}\|_2^2, \tag{3.3}$$

où $\boldsymbol{\varphi_u}$ *est la matrice triangulaire telle que* $\boldsymbol{\psi_u} = \boldsymbol{\varphi_u}\boldsymbol{\varphi_u}^\top$ *(puisque* $\boldsymbol{\psi_u}$ *est une matrice définie positive, il existe une matrice* $\boldsymbol{\varphi_u}$ *telle que* $\boldsymbol{\varphi_u}\boldsymbol{\varphi_u}^\top = \boldsymbol{\psi_u}$ *à partir d'une factorisation de Choleski de* $\boldsymbol{\psi_u}$*).*

De plus, si le filtre $\mathcal{H}$ *est exprimé par le state-space* $(\boldsymbol{A},\boldsymbol{B},\boldsymbol{C},\boldsymbol{D})$*, alors la norme* $\mathcal{L}_2$ $\|\mathcal{H}\cdot\boldsymbol{\varphi_u}\|_2^2$ *peut se calculer d'après le grammien d'observabilité,* $\boldsymbol{W}_o$*, du filtre* $\mathcal{H}$*, par :*

$$\boldsymbol{\sigma_y^2} = \mathrm{tr}\left(\boldsymbol{\psi_u}(\boldsymbol{D}^\top\boldsymbol{D}+\boldsymbol{B}^\top\boldsymbol{W}_o\boldsymbol{B})\right). \tag{3.4}$$

Démonstration. La démonstration est donnée dans l'annexe C.1. □

On souhaite dans notre approche déterminer le format virgule fixe du signal $\boldsymbol{y}$. Ainsi, la proposition précédente est limitée du fait qu'elle est constitue une approche statistique : on connait la moyenne statistique de la sortie $\boldsymbol{y}$ sur l'ensemble des instants k et la dispersion des différentes sorties $\boldsymbol{y}(k)$ autour de cette moyenne. On ne peut donc pas en déduire de façon sûr un format virgule fixe pouvant représenter toutes les valeurs de la sortie.

L'idée est alors d'établir une proposition équivalente à la proposition 3.1 avec des intervalles, c'est-à-dire de déduire l'intervalle de sortie $\langle\boldsymbol{y}_m,\boldsymbol{y}_r\rangle$ à partir de l'intervalle d'entrée $\langle\boldsymbol{u}_m,\boldsymbol{u}_r\rangle$.

Cependant, il est nécessaire de rappeler que les signaux considérés sont définis pour $k \geqslant 0$, en particulier, le signal de sortie $\boldsymbol{y}$ est tel que $\boldsymbol{y}(k) = 0$ pour $k < 0$. Le système a donc besoin d'un certains temps pour atteindre un équilibre dans un intervalle $\langle\boldsymbol{y}_m,\boldsymbol{y}_r\rangle$. Cette phase est appelée *phase transitoire*, par opposition avec le *régime permanent* quand le système a atteint un équilibre.

Proposition 3.2 (Intervalle à travers un filtre)**.** *Soient* $\mathcal{H}$ *un filtre linéaire MIMO et* $\boldsymbol{u}$ *et* $\boldsymbol{y}$ *deux vecteurs de signaux tels que* $\boldsymbol{y}$ *est la sortie du filtre pour l'entrée* $\boldsymbol{u}$ *(*$\boldsymbol{y} = \boldsymbol{h} * \boldsymbol{u}$ *avec* $\boldsymbol{h}$ *la réponse impulsionnelle de* $\mathcal{H}$*).*
Si l'entrée $\boldsymbol{u}$ *est dans l'intervalle (connu)* $\langle\boldsymbol{u}_m,\boldsymbol{u}_r\rangle$ *alors on peut calculer d'une part le seuil à partir duquel le système est en régime permanent à un* $\boldsymbol{\varepsilon}$ *près, et d'autre part l'intervalle* $\langle\boldsymbol{y}_m,\boldsymbol{y}_r\rangle$ *de ce régime. Plus formellement, si* $\boldsymbol{u} \in$

$\langle \boldsymbol{u}_m, \boldsymbol{u}_r \rangle$, alors pour tout $\boldsymbol{\varepsilon} > 0$, il existe un seuil $K \in \mathbb{N}$ tel que pour tout $k \geqslant K$, on a $\boldsymbol{y}(k) \in \langle \boldsymbol{y}_m, \boldsymbol{y}_r \rangle$ avec :

$$\boldsymbol{y}_m = \langle\!\langle \mathcal{H} \rangle\!\rangle_{\mathrm{DC}} \cdot \boldsymbol{u}_m, \tag{3.5}$$

$$\boldsymbol{y}_r = \langle\!\langle \mathcal{H} \rangle\!\rangle_{\mathrm{wcpg}} \cdot \boldsymbol{u}_r + \boldsymbol{\varepsilon}. \tag{3.6}$$

De plus, il est possible d'exhiber un signal $\boldsymbol{u}$ tel qu'une composante $\boldsymbol{y}_i$ de $\boldsymbol{y}$, pour un certain i, atteint une borne $(\boldsymbol{y}_r \pm \boldsymbol{y}_m)_i$ à une certaine précision près.

Démonstration. La démonstration est donnée dans l'annexe C.2. □

Remarque 3.1. *La proposition 3.2 nous donne l'intervalle de la sortie $\boldsymbol{y}$ pour un régime permanent à un certain ε près donné. Puisque le seuil K de passage entre la phase transitoire et le régime permanent est un entier, on peut déterminer l'intervalle de la sortie dans la phase transitoire par une recherche exhaustive des extrema parmi les valeurs $\boldsymbol{y}(k)$ pour $0 \leqslant k < K$.*

Remarque 3.2. *Dans l'équation (3.6), $\boldsymbol{\varepsilon}$ décroit rapidement en fonction du temps. En effet, cette décroissance suit celle de $\rho(\boldsymbol{A})^k$ en fonction de k, où $\rho(\boldsymbol{A})$ est le rayon spectral[1] de la matrice $\boldsymbol{A}$ et $\boldsymbol{A}$ la matrice des coefficients du filtre $\mathcal{H}$ écrit sous forme d'un state-space $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$. Pour les filtres stables que nous considérons, le rayon spectral de la matrice $\boldsymbol{A}$ est strictement inférieur à 1, donc plus $\rho(\boldsymbol{A})$ est proche de 1, plus $\boldsymbol{\varepsilon}$ décroit lentement, inversement.*

Le régime permanent est atteint quand ε tend vers 0, c'est-à-dire quand k tend vers l'infini (après une infinité de cycles du filtre), on peut donc déduire de la proposition 3.2 une proposition équivalente en régime permanent.

Proposition 3.3 (Intervalle à travers un filtre en régime permanent)**.** *Soient $\mathcal{H}$ un filtre linéaire MIMO et $\boldsymbol{u}$ et $\boldsymbol{y}$ deux vecteurs de signaux tels que $\boldsymbol{y}$ est la sortie du filtre pour l'entrée $\boldsymbol{u}$ ($\boldsymbol{y} = \boldsymbol{h} * \boldsymbol{u}$ avec $\boldsymbol{h}$ la réponse impulsionnelle de $\mathcal{H}$).*
Si l'entrée $\boldsymbol{u}$ est dans l'intervalle (connu) $\langle \boldsymbol{u}_m, \boldsymbol{u}_r \rangle$ alors $\boldsymbol{y}$ est dans l'intervalle $\langle \boldsymbol{y}_m, \boldsymbol{y}_r \rangle$ avec :

$$\boldsymbol{y}_m = \langle\!\langle \mathcal{H} \rangle\!\rangle_{\mathrm{DC}} \cdot \boldsymbol{u}_m, \tag{3.7}$$

$$\boldsymbol{y}_r = \langle\!\langle \mathcal{H} \rangle\!\rangle_{\mathrm{wcpg}} \cdot \boldsymbol{u}_r. \tag{3.8}$$

Remarque 3.3. *Il est important de noter que les applications de ces propositions dans notre approche, pour déterminer le format virgule fixe de la sortie $\boldsymbol{y}$ ou pour le calcul de l'erreur, n'utilisent finalement que le logarithme en base 2 des bornes de l'intervalle de sortie. On peut par conséquent, dans la majorité des cas et sans perte de généralité, utiliser la proposition 3.3 dans notre approche.*

1. Le rayon spectral d'une matrice $\boldsymbol{M}$ correspond au plus grand module des valeurs propres de $\boldsymbol{M}$.

Remarque 3.4. *En notation bornes inférieure/supérieure, avec* $\boldsymbol{u} \in [\underline{\boldsymbol{u}}; \overline{\boldsymbol{u}}]$ *et* $\boldsymbol{y} \in [\underline{\boldsymbol{y}}; \overline{\boldsymbol{y}}]$*, on a :*

$$\boldsymbol{u}_m = \frac{\overline{\boldsymbol{u}} + \underline{\boldsymbol{u}}}{2}, \quad \boldsymbol{u}_r = \frac{\overline{\boldsymbol{u}} - \underline{\boldsymbol{u}}}{2},$$
$$\underline{\boldsymbol{y}} = \boldsymbol{y}_r - \boldsymbol{y}_m, \quad \overline{\boldsymbol{y}} = \boldsymbol{y}_r + \boldsymbol{y}_m,$$

et on obtient l'équivalence suivante pour les équations (3.6) *et* (3.5) *de la proposition 3.2 :*

$$\underline{\boldsymbol{y}} = \langle\!\langle \mathcal{H} \rangle\!\rangle_{\mathrm{DC}} \frac{\overline{\boldsymbol{u}} + \underline{\boldsymbol{u}}}{2} - \langle\!\langle \mathcal{H} \rangle\!\rangle_{\mathrm{WCPG}} \frac{\overline{\boldsymbol{u}} - \underline{\boldsymbol{u}}}{2} \tag{3.9}$$
$$\overline{\boldsymbol{y}} = \langle\!\langle \mathcal{H} \rangle\!\rangle_{\mathrm{DC}} \frac{\overline{\boldsymbol{u}} + \underline{\boldsymbol{u}}}{2} + \langle\!\langle \mathcal{H} \rangle\!\rangle_{\mathrm{WCPG}} \frac{\overline{\boldsymbol{u}} - \underline{\boldsymbol{u}}}{2}, \tag{3.10}$$

L'intervalle du signal d'entrée $\boldsymbol{u}$ est généralement connu, par conséquent la proposition 3.2 nous permet d'exprimer l'intervalle dans lequel se trouvera le signal de sortie du filtre. Cette description de l'évolution d'un signal à travers un filtre sous forme d'un intervalle est nouvelle.

Ces deux propositions sont très importantes dans notre approche, elles vont d'une part pouvoir être appliquées sur l'entrée pour connaître le format virgule de la sortie (ou du moins son *msb* si la largeur n'est pas connue), et d'autre part nous être utile pour l'analyse d'erreur en sortie d'un filtre, comme le décrit la prochaine section.

3.2 Analyse de l'erreur en sortie d'un filtre

3.2.1 Propagation des erreurs de calcul

Nous avons vu qu'il y avait, pour un filtre linéaire donné, plusieurs réalisations possibles, dont par exemple les décompositions en sous-filtres en parallèle ou en cascade, qui sont dues à la linéarité de ces filtres. Une autre application de cette linéarité est à la base de l'analyse d'erreur en sortie d'un filtre.

Considérons alors le filtre $\mathcal{H}$ exprimé sous la forme implicite, en reprenant l'algorithme de la SIF des équations (1.48). En précision infinie, cet algorithme sera calculé exactement, mais en précision finie, c'est en fait l'algorithme d'un filtre, noté $\mathcal{H}^*$ (le filtre $\mathcal{H}$ dégradé par les erreurs de calcul), qui sera calculé :

$$\boldsymbol{t}^*(k+1) = -\boldsymbol{J}'\boldsymbol{t}^*(k+1) + \boldsymbol{M}\boldsymbol{x}^*(k) + \boldsymbol{N}\boldsymbol{u}(k) + \boldsymbol{\varepsilon}_{\boldsymbol{t}}(k) \tag{3.11a}$$
$$\boldsymbol{x}^*(k+1) = \boldsymbol{K}\boldsymbol{t}^*(k+1) + \boldsymbol{P}\boldsymbol{x}^*(k) + \boldsymbol{Q}\boldsymbol{u}(k) + \boldsymbol{\varepsilon}_{\boldsymbol{x}}(k) \tag{3.11b}$$
$$\boldsymbol{y}^*(k) = \boldsymbol{L}\boldsymbol{t}^*(k+1) + \boldsymbol{R}\boldsymbol{x}^*(k) + \boldsymbol{S}\boldsymbol{u}(k) + \boldsymbol{\varepsilon}_{\boldsymbol{y}}(k) \tag{3.11c}$$

où $\boldsymbol{\varepsilon}_{\boldsymbol{z}}(k)$ est le vecteurs des erreurs de calculs à l'instant k sur le vecteur de variables $\boldsymbol{z}$ ($\boldsymbol{\varepsilon}_{\boldsymbol{z}_i}(k)$ est l'erreur de calcul à l'instant k sur la variable $\boldsymbol{z}_i$, pour $1 \leqslant i \leqslant n_z$), et $\boldsymbol{z}^*(k)$ est le vecteur $\boldsymbol{z}(k)$ dégradé par les erreurs de calcul, pour

$\boldsymbol{z} \in \{\boldsymbol{t}, \boldsymbol{x}, \boldsymbol{y}\}$. On note alors $\boldsymbol{\varepsilon}(k)$ le vecteur des erreurs de calculs à l'instant k, c.-à-d. :

$$\boldsymbol{\varepsilon}(k) \triangleq \begin{pmatrix} \boldsymbol{\varepsilon_t}(k) \\ \boldsymbol{\varepsilon_x}(k) \\ \boldsymbol{\varepsilon_y}(k) \end{pmatrix}. \tag{3.12}$$

Remarque 3.5. *Puisqu'on ne considère ici que les erreurs de calcul (ce sera le cas dans la section 3.2.2), on suppose que les coefficients sont les mêmes pour le filtre exact et pour le filtre implémenté. On reviendra sur ce point dans la section 3.2.2.*

On note $\boldsymbol{\delta z}_i(k)$ la différence $\boldsymbol{z}_i^*(k) - \boldsymbol{z}_i(k)$, c'est-à-dire l'erreur à l'instant k en considérant les erreurs de calcul et le rebouclage, pour $\boldsymbol{z} \in \{\boldsymbol{t}, \boldsymbol{x}, \boldsymbol{y}\}$ et $1 \leqslant i \leqslant n_z$. On a alors, en soustrayant (3.11) à l'algorithme de la SIF des équations (1.48) :

$$\boldsymbol{\delta t}(k+1) = -\boldsymbol{J}'\boldsymbol{\delta t}(k+1) + \boldsymbol{M}\boldsymbol{\delta x}(k) + \boldsymbol{\varepsilon_t}(k) \tag{3.13a}$$
$$\boldsymbol{\delta x}(k+1) = \boldsymbol{K}\boldsymbol{\delta t}(k+1) + \boldsymbol{P}\boldsymbol{\delta x}(k) + \boldsymbol{\varepsilon_x}(k) \tag{3.13b}$$
$$\boldsymbol{\delta y}(k) = \boldsymbol{L}\boldsymbol{\delta t}(k+1) + \boldsymbol{R}\boldsymbol{\delta x}(k) + \boldsymbol{\varepsilon_y}(k) \tag{3.13c}$$

Cet algorithme exprime la différence entre l'algorithme de la SIF en précision infinie et celui en précision finie. Il correspond en fait à l'algorithme sous forme implicite d'un filtre $\mathcal{H}_{\boldsymbol{\varepsilon}}$, exprimant le comportement des erreurs de calcul à l'instant k sur la sortie, et dont les entrées sont les $\boldsymbol{\varepsilon}(k)$. En effet, en considérant $\boldsymbol{\delta t}$ comme le vecteur des variables intermédiaires et $\boldsymbol{\delta x}$ le vecteur des variables d'états, cet algorithme calcule bien $\boldsymbol{\delta y}(k)$ à partir de $\boldsymbol{\varepsilon}(k)$. La linéarité des filtres permet de décomposer $\mathcal{H}^*$ en deux filtres distincts, d'un côté le filtre exact $\mathcal{H}$, et de l'autre le filtre de l'erreur $\mathcal{H}_{\boldsymbol{\varepsilon}}$ (voir figure 3.1).

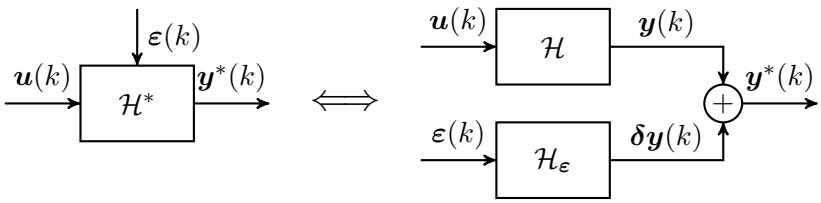


FIGURE 3.1 – Équivalence entre le système dégradé $\mathcal{H}^*$ et la décomposition en deux systèmes distincts $\mathcal{H}$ et $\mathcal{H}_{\boldsymbol{\varepsilon}}$.

Si la SIF de $\mathcal{H}$ est caractérisée par la matrice $\boldsymbol{Z} = \begin{pmatrix} -\boldsymbol{J} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix}$, alors

la SIF de $\mathcal{H}_{\boldsymbol{\varepsilon}}$ est caractérisée par $\boldsymbol{Z}_\varepsilon \triangleq \begin{pmatrix} -\boldsymbol{J} & \boldsymbol{M} & \boldsymbol{M}_t \\ \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{M}_x \\ \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{M}_y \end{pmatrix}$ avec :

$$\boldsymbol{M}_t \triangleq \begin{pmatrix} \boldsymbol{I}_{n_t} & \boldsymbol{0}_{n_t \times n_x} & \boldsymbol{0}_{n_t \times n_y} \end{pmatrix}, \tag{3.14a}$$

$$\boldsymbol{M}_x \triangleq \begin{pmatrix} \boldsymbol{0}_{n_x \times n_t} & \boldsymbol{I}_{n_x} & \boldsymbol{0}_{n_x \times n_y} \end{pmatrix}, \tag{3.14b}$$

$$\boldsymbol{M}_y \triangleq \begin{pmatrix} \boldsymbol{0}_{n_y \times n_t} & \boldsymbol{0}_{n_y \times n_x} & \boldsymbol{I}_{n_y} \end{pmatrix}, \tag{3.14c}$$

où $\boldsymbol{I}_n$ est la matrice identité de taille $n \times n$ et $\boldsymbol{0}_{n \times m}$ est la matrice de zéros de taille $n \times m$.

$\mathcal{H}_{\boldsymbol{\varepsilon}}$ est un filtre à $(n_t + n_x + n_u)$ entrées et n_y sorties.

Proposition 3.4. *La fonction de transfert du filtre $\mathcal{H}_{\boldsymbol{\varepsilon}}$, $\boldsymbol{H}_\varepsilon$, est donnée par :*

$$\boldsymbol{H}_\varepsilon : z \to \boldsymbol{C}_{\boldsymbol{Z}}(z\boldsymbol{I}_n - \boldsymbol{A}_{\boldsymbol{Z}})^{-1}\boldsymbol{M}_1 + \boldsymbol{M}_2, \quad \forall z \in \mathbb{C} \tag{3.15}$$

avec $\boldsymbol{A}_{\boldsymbol{Z}}$ et $\boldsymbol{C}_{\boldsymbol{Z}}$ les matrices définies par (1.45) et

$$\boldsymbol{M}_1 \triangleq \begin{pmatrix} \boldsymbol{K}\boldsymbol{J}^{-1} & \boldsymbol{I}_{n_x} & \boldsymbol{0} \end{pmatrix}, \quad \boldsymbol{M}_2 \triangleq \begin{pmatrix} \boldsymbol{L}\boldsymbol{J}^{-1} & \boldsymbol{0} & \boldsymbol{I}_{n_y} \end{pmatrix}. \tag{3.16}$$

Démonstration. La fonction de transfert s'obtient, à partir de la matrice $\boldsymbol{Z}$ d'une SIF, par l'équation (1.44). Dans le cas du filtre $\mathcal{H}_{\boldsymbol{\varepsilon}}$, la matrice des coefficients est $\boldsymbol{Z}_\varepsilon$, où les matrices $\boldsymbol{N}$, $\boldsymbol{Q}$ et $\boldsymbol{S}$ sont respectivement remplacées par $\boldsymbol{M}_t$, $\boldsymbol{M}_x$ et $\boldsymbol{M}_Y$. Par conséquent, $\boldsymbol{B}_{\boldsymbol{Z}} = \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{Q}$ est remplacée par $\boldsymbol{M}_1 = \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{M}_t + \boldsymbol{M}_x$ et $\boldsymbol{D}_{\boldsymbol{Z}} = \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{S}$ est remplacée par $\boldsymbol{M}_2 = \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{M}_t + \boldsymbol{M}_y$. □

On peut donc, à partir de la représentation sous forme implicite d'un filtre $\mathcal{H}$, déterminer la fonction de transfert du filtre $\mathcal{H}_{\boldsymbol{\varepsilon}}$, exprimant le traitement de l'erreur commise par l'implémentation en précision finie. Le corollaire suivant est une application des propositions 3.1 et 3.2 au filtre $\mathcal{H}_{\boldsymbol{\varepsilon}}$.

Corollaire 3.1. *Soient $\mathcal{H}$ un filtre, $\boldsymbol{\varepsilon}(k)$ le vecteur des erreurs de calcul commises à l'instant k lors de l'implémentation de $\mathcal{H}$ en précision finie, et $\mathcal{H}_{\boldsymbol{\varepsilon}}$ le filtre déduit de $\mathcal{H}$ exprimant le comportement de l'erreur au travers du filtre implémenté. Il est alors possible d'exprimer ce comportement à partir de $\boldsymbol{\varepsilon}(k)$ et de $\mathcal{H}_{\boldsymbol{\varepsilon}}$:*

- *si l'erreur de calcul est modélisée par un vecteur de bruit blanc uniformément distribué avec des vecteur de moments d'ordre un et deux respectivement notés $\boldsymbol{\mu}_{\boldsymbol{\varepsilon}}$ et $\boldsymbol{\sigma}^2_{\boldsymbol{\varepsilon}}$, alors les moments de l'erreur globale $\boldsymbol{\delta y}$ commise sur la sortie du filtre implémenté, notés $\boldsymbol{\mu}_{\boldsymbol{\delta y}}$ et $\boldsymbol{\sigma}^2_{\boldsymbol{\delta y}}$, sont donnés par :*

$$\boldsymbol{\mu}_{\boldsymbol{\delta y}} = \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}} \cdot \boldsymbol{\mu}_{\boldsymbol{\varepsilon}}, \tag{3.17}$$

$$\boldsymbol{\sigma}^2_{\boldsymbol{\delta y}} = \|\mathcal{H}_{\boldsymbol{\varepsilon}} \cdot \varphi_\varepsilon\|_2^2. \tag{3.18}$$

– *si l'erreur de calcul est modélisée par un vecteur d'intervalle, noté* $\langle \boldsymbol{\varepsilon}_m, \boldsymbol{\varepsilon}_r \rangle$*, alors le vecteur d'intervalle de l'erreur globale* $\boldsymbol{\delta y}$ *commise sur la sortie du filtre implémenté, noté* $\langle \boldsymbol{\delta y}_m, \boldsymbol{\delta y}_r \rangle$*, est donné par :*

$$\boldsymbol{\delta y}_m = \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}} \cdot \boldsymbol{\varepsilon}_m, \tag{3.19}$$

$$\boldsymbol{\delta y}_r = \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{wcpg}} \cdot \boldsymbol{\varepsilon}_r. \tag{3.20}$$

Le filtre $\mathcal{H}_{\boldsymbol{\varepsilon}}$ ne considère que le comportement des erreurs de calcul au travers du filtre implémenté, et ne considère donc pas les erreurs paramétriques faites lors de la quantification des coefficients. Lors de nos différents tests, que ce soit dans nos articles ou dans ce rapport de thèse, c'est cette méthode que nous utilisons pour évaluer l'impact de nos erreurs, et donc les comparaisons effectuées entre filtre exact $\mathcal{H}$ et filtre implémenté $\mathcal{H}^*$ considère finalement pour $\mathcal{H}$ un filtre ayant les mêmes coefficients (quantifié) que le filtre $\mathcal{H}^*$ (usuellement, la quantification des coefficients est en effet étudiée à part avec les mesures de sensibilité discutées au chapitre 1). Il est néanmoins possible d'aller plus loin et de déterminer le filtre $\widetilde{\mathcal{H}}_{\boldsymbol{\varepsilon}}$ exprimant le comportement des erreurs de calculs **et** des erreurs paramétriques au travers du filtre implémenté, comme le montre la partie suivante.

3.2.2 Propagation des erreurs de calcul et des erreurs paramétriques

La remarque 3.5 met en avant un point important : on considère en effet dans les équations (3.11) que les matrices de coefficients sont les mêmes entre le filtre exact $\mathcal{H}$ et le filtre implémenté $\mathcal{H}^*$, ce qui est vrai si on veut seulement mettre en avant les erreurs de calculs (on prend les mêmes coefficients, on déroule la méthode et les seules erreurs apparaissant sont les erreurs de calculs). Cependant, si on veut en plus prendre en compte les erreurs paramétriques, alors le système implémenté, noté $\widetilde{\mathcal{H}}$ (différent de $\mathcal{H}^*$ puisqu'il considère l'ensemble des erreurs d'implémentation), peut s'écrire :

$$\widetilde{\boldsymbol{t}}(k+1) = -\widetilde{\boldsymbol{J}'}\widetilde{\boldsymbol{t}}(k+1) + \widetilde{\boldsymbol{M}}\widetilde{\boldsymbol{x}}(k) + \widetilde{\boldsymbol{N}}\boldsymbol{u}(k) + \boldsymbol{\varepsilon}_{\boldsymbol{t}}(k) \tag{3.21a}$$

$$\widetilde{\boldsymbol{x}}(k+1) = \widetilde{\boldsymbol{K}}\widetilde{\boldsymbol{t}}(k+1) + \widetilde{\boldsymbol{P}}\widetilde{\boldsymbol{x}}(k) + \widetilde{\boldsymbol{Q}}\boldsymbol{u}(k) + \boldsymbol{\varepsilon}_{\boldsymbol{x}}(k) \tag{3.21b}$$

$$\widetilde{\boldsymbol{y}}(k) = \widetilde{\boldsymbol{L}}\widetilde{\boldsymbol{t}}(k+1) + \widetilde{\boldsymbol{R}}\widetilde{\boldsymbol{x}}(k) + \widetilde{\boldsymbol{S}}\boldsymbol{u}(k) + \boldsymbol{\varepsilon}_{\boldsymbol{y}}(k) \tag{3.21c}$$

où $\widetilde{\boldsymbol{C}}$ est la matrice des coefficients quantifiés de $\boldsymbol{C}$ pour toute matrice $\boldsymbol{C} \in \{\boldsymbol{J}', \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{K}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{L}, \boldsymbol{R}, \boldsymbol{S}\}$, et $\widetilde{\boldsymbol{z}}(k)$ est le vecteur $\boldsymbol{z}(k)$ bruité par les erreurs de calcul et les erreurs paramétriques, pour $\boldsymbol{z} \in \{\boldsymbol{t}, \boldsymbol{x}, \boldsymbol{y}\}$. Les vecteurs $\boldsymbol{\varepsilon}_{\boldsymbol{t}}(k)$, $\boldsymbol{\varepsilon}_{\boldsymbol{x}}(k)$ et $\boldsymbol{\varepsilon}_{\boldsymbol{y}}(k)$ sont définis comme précédemment comme les vecteurs des erreurs de calculs à l'instant k sur les vecteurs $\boldsymbol{t}$, $\boldsymbol{x}$ et $\boldsymbol{y}$ respectivement.

On note $\boldsymbol{\Delta z}_i(k)$ la différence $\widetilde{\boldsymbol{z}}_i(k) - \boldsymbol{z}_i(k)$, c'est-à-dire l'erreur à l'instant k en considérant les erreurs de calcul, les erreurs paramétriques, et le rebouclage

de ces erreurs, pour $1 \leqslant i \leqslant n_z$ et $\boldsymbol{z} \in \{\boldsymbol{t}, \boldsymbol{x}, \boldsymbol{y}\}$. On note également $\boldsymbol{\Delta C}$ la matrice donnée par $\boldsymbol{C} - \widetilde{\boldsymbol{C}}$. On peut remarquer que l'on a choisi un ordre différent de celui des vecteurs d'erreurs, on a en effet $\boldsymbol{\Delta C} = \boldsymbol{C} - \widetilde{\boldsymbol{C}}$ pour les matrices et $\boldsymbol{\Delta z}_i(k) = \widetilde{\boldsymbol{z}}_i(k) - \boldsymbol{z}_i(k)$ pour les variables, car cela nous permet une écriture simplifiée de la SIF finale. On a alors, à partir de (1.48) et (3.21) :

$$\begin{aligned}
\boldsymbol{\Delta t}(k+1) = &- \widetilde{\boldsymbol{J}'}\boldsymbol{\Delta t}(k+1) + \boldsymbol{\Delta J'}\boldsymbol{t}(k+1) + \widetilde{\boldsymbol{M}}\boldsymbol{\Delta x}(k) && \text{(3.22a)}\\
&- \boldsymbol{\Delta M}\boldsymbol{x}(k) - \boldsymbol{\Delta N}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_t}(k) && \text{(3.22b)}\\
\boldsymbol{\Delta x}(k+1) = &\widetilde{\boldsymbol{K}}\boldsymbol{\Delta t}(k+1) + \boldsymbol{\Delta K}\boldsymbol{t}(k+1) + \widetilde{\boldsymbol{P}}\boldsymbol{\Delta x}(k) && \text{(3.22c)}\\
&- \boldsymbol{\Delta P}\boldsymbol{x}(k) - \boldsymbol{\Delta Q}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_x}(k) && \text{(3.22d)}\\
\boldsymbol{\Delta y}(k) = &\widetilde{\boldsymbol{L}}\boldsymbol{\Delta t}(k+1) + \boldsymbol{\Delta L}\boldsymbol{t}(k+1) + \widetilde{\boldsymbol{R}}\boldsymbol{\Delta x}(k) && \text{(3.22e)}\\
&- \boldsymbol{\Delta R}\boldsymbol{x}(k) - \boldsymbol{\Delta S}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_y}(k) && \text{(3.22f)}
\end{aligned}$$

Les équations ainsi obtenues décrivent l'algorithme d'une SIF, donnée par :

$$\begin{pmatrix}
\boldsymbol{J} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0}\\
-\boldsymbol{\Delta J} & \widetilde{\boldsymbol{J}} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0}\\
-\boldsymbol{K} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0}\\
\boldsymbol{\Delta K} & -\widetilde{\boldsymbol{K}} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0}\\
\boldsymbol{\Delta L} & -\widetilde{\boldsymbol{L}} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I}
\end{pmatrix}
\begin{pmatrix}
\boldsymbol{t}(k+1)\\
\boldsymbol{\Delta t}(k+1)\\
\boldsymbol{x}(k+1)\\
\boldsymbol{\Delta x}(k+1)\\
\boldsymbol{\Delta y}(k)
\end{pmatrix}
= \begin{pmatrix}
\boldsymbol{0} & \boldsymbol{0} & \boldsymbol{M} & \boldsymbol{0} & \boldsymbol{N} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0}\\
\boldsymbol{0} & \boldsymbol{0} & -\boldsymbol{\Delta M} & \widetilde{\boldsymbol{M}} & -\boldsymbol{\Delta N} & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0}\\
\boldsymbol{0} & \boldsymbol{0} & \boldsymbol{P} & \boldsymbol{0} & \boldsymbol{Q} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0}\\
\boldsymbol{0} & \boldsymbol{0} & -\boldsymbol{\Delta P} & \widetilde{\boldsymbol{P}} & -\boldsymbol{\Delta Q} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0}\\
\boldsymbol{0} & \boldsymbol{0} & -\boldsymbol{\Delta R} & \widetilde{\boldsymbol{R}} & -\boldsymbol{\Delta S} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I}
\end{pmatrix}
\begin{pmatrix}
\boldsymbol{t}(k)\\
\boldsymbol{\Delta t}(k)\\
\boldsymbol{x}(k)\\
\boldsymbol{\Delta x}(k)\\
\boldsymbol{u}(k)\\
\boldsymbol{\varepsilon_t}(k)\\
\boldsymbol{\varepsilon_x}(k)\\
\boldsymbol{\varepsilon_y}(k)
\end{pmatrix} \tag{3.23}$$

où le vecteur $\begin{pmatrix}\boldsymbol{t}(k)\\ \boldsymbol{\Delta t}(k)\end{pmatrix}$ représente le vecteur des variables intermédiaires, le vecteur $\begin{pmatrix}\boldsymbol{x}(k)\\ \boldsymbol{\Delta x}(k)\end{pmatrix}$ celui des variables d'états, et $\boldsymbol{\Delta y}(k)$ est le vecteur des sorties du système pour le vecteur d'entrées $\begin{pmatrix}\boldsymbol{u}(k)\\ \boldsymbol{\varepsilon_t}(k)\\ \boldsymbol{\varepsilon_x}(k)\\ \boldsymbol{\varepsilon_y}(k)\end{pmatrix}$, le tout à l'instant k.

On utilise les notations suivantes :

$$\mathcal{J} \triangleq \begin{pmatrix} \boldsymbol{J} & \boldsymbol{0} \\ -\boldsymbol{\Delta J} & \widetilde{\boldsymbol{J}} \end{pmatrix}, \quad \mathcal{K} \triangleq \begin{pmatrix} \boldsymbol{K} & \boldsymbol{0} \\ -\boldsymbol{\Delta K} & \widetilde{\boldsymbol{K}} \end{pmatrix}, \tag{3.24a}$$

$$\mathcal{M} \triangleq \begin{pmatrix} \boldsymbol{M} & \boldsymbol{0} \\ -\boldsymbol{\Delta M} & \widetilde{\boldsymbol{M}} \end{pmatrix}, \quad \mathcal{P} \triangleq \begin{pmatrix} \boldsymbol{P} & \boldsymbol{0} \\ -\boldsymbol{\Delta P} & \widetilde{\boldsymbol{P}} \end{pmatrix} \tag{3.24b}$$

$$\mathcal{L} \triangleq \begin{pmatrix} -\boldsymbol{\Delta L} & \widetilde{\boldsymbol{L}} \end{pmatrix}, \quad \mathcal{R} \triangleq \begin{pmatrix} -\boldsymbol{\Delta R} & \widetilde{\boldsymbol{R}} \end{pmatrix}, \tag{3.24c}$$

$$\mathcal{M}_t \triangleq \begin{pmatrix} \boldsymbol{N} & \boldsymbol{0}_{n_t \times n_t} & \boldsymbol{0}_{n_t \times n_x} & \boldsymbol{0}_{n_t \times n_y} \\ -\boldsymbol{\Delta N} & \boldsymbol{I}_{n_t} & \boldsymbol{0}_{n_t \times n_x} & \boldsymbol{0}_{n_t \times n_y} \end{pmatrix}, \tag{3.24d}$$

$$\mathcal{M}_x \triangleq \begin{pmatrix} \boldsymbol{Q} & \boldsymbol{0}_{n_x \times n_t} & \boldsymbol{0}_{n_x \times n_x} & \boldsymbol{0}_{n_x \times n_y} \\ -\boldsymbol{\Delta Q} & \boldsymbol{0}_{n_t \times n_x} & \boldsymbol{I}_{n_x} & \boldsymbol{0}_{n_x \times n_y} \end{pmatrix}, \tag{3.24e}$$

$$\mathcal{M}_y \triangleq \begin{pmatrix} -\boldsymbol{\Delta S} & \boldsymbol{0}_{n_y \times n_t} & \boldsymbol{0}_{n_y \times n_x} & \boldsymbol{I}_{n_y} \end{pmatrix}. \tag{3.24f}$$

On a alors la correspondance suivante pour la matrice des coefficients, notée $\mathcal{Z}$:

$$\mathcal{Z} \triangleq \left(\begin{array}{cc:cc:cccc} -\boldsymbol{J} & \boldsymbol{0} & \boldsymbol{M} & \boldsymbol{0} & \boldsymbol{N} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{\Delta J} & -\widetilde{\boldsymbol{J}} & -\boldsymbol{\Delta M} & \widetilde{\boldsymbol{M}} & -\boldsymbol{\Delta N} & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \\ \hdashline \boldsymbol{K} & \boldsymbol{0} & \boldsymbol{P} & \boldsymbol{0} & \boldsymbol{Q} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ -\boldsymbol{\Delta K} & \widetilde{\boldsymbol{K}} & -\boldsymbol{\Delta P} & \widetilde{\boldsymbol{P}} & -\boldsymbol{\Delta Q} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} \\ \hdashline -\boldsymbol{\Delta L} & \widetilde{\boldsymbol{L}} & -\boldsymbol{\Delta R} & \widetilde{\boldsymbol{R}} & -\boldsymbol{\Delta S} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{array}\right) = \begin{pmatrix} -\mathcal{J} & \mathcal{M} & \mathcal{M}_t \\ \mathcal{K} & \mathcal{P} & \mathcal{M}_x \\ \mathcal{L} & \mathcal{R} & \mathcal{M}_y \end{pmatrix}. \tag{3.25}$$

Cette SIF est la forme implicite d'un filtre $\widetilde{\mathcal{H}}_{\boldsymbol{\varepsilon}}$ traduisant le comportement des erreurs de calculs **et** des erreurs paramétriques lors de l'implémentation de $\widetilde{\mathcal{H}}$. Comme dans le cas précédent, on peut alors séparer le système entre le filtre exact $\mathcal{H}$ et le filtre des erreurs $\widetilde{\mathcal{H}}$, comme le montre la figure 3.2.

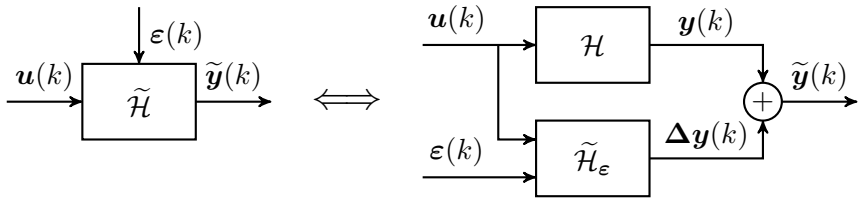


FIGURE 3.2 – Équivalence entre le système dégradé $\widetilde{\mathcal{H}}$ et la décomposition en deux systèmes distincts $\mathcal{H}$ et $\widetilde{\mathcal{H}}_{\boldsymbol{\varepsilon}}$.

Remarque 3.6. *Il y a une différence notable entre les expressions de $\mathcal{H}_{\boldsymbol{\varepsilon}}$ et de $\widetilde{\mathcal{H}}_{\boldsymbol{\varepsilon}}$. En effet, pour le premier filtre, on peut l'exprimer en amont des choix d'implémentation, les erreurs de calculs étant des entrées du système.*

Dans le cas du filtre $\widetilde{\mathcal{H}}_{\boldsymbol{\varepsilon}}$, on peut voir dans l'expression de la SIF de l'équation (3.23) que les matrices quantifiées font partie intégrante du système, il est donc nécessaire de connaître les formats virgule fixe des coefficients et d'effectuer les quantifications avant de pouvoir déterminer la SIF de $\widetilde{\mathcal{H}}_{\boldsymbol{\varepsilon}}$. De plus, les erreurs $\boldsymbol{\Delta y}$ dépendent de l'entrée $\boldsymbol{u}$ (voir figure 3.2), ce qui n'était pas le cas avec les erreurs $\boldsymbol{\delta y}$.

On peut, comme pour $\mathcal{H}_{\boldsymbol{\varepsilon}}$, donner l'expression de la fonction de transfert de $\widetilde{\mathcal{H}}_{\boldsymbol{\varepsilon}}$.

Proposition 3.5. *La fonction de transfert du filtre $\widetilde{\mathcal{H}}_{\boldsymbol{\varepsilon}}$, $\widetilde{\boldsymbol{H}}_{\varepsilon}$, est donnée par :*

$$\widetilde{\boldsymbol{H}}_{\varepsilon} : z \to \boldsymbol{\mathcal{C}}_{\boldsymbol{\mathcal{Z}}}(z\boldsymbol{I}_n - \boldsymbol{\mathcal{A}}_{\boldsymbol{\mathcal{Z}}})^{-1}\boldsymbol{\mathcal{B}}_{\boldsymbol{\mathcal{Z}}} + \boldsymbol{\mathcal{D}}_{\boldsymbol{\mathcal{Z}}}, \quad \forall z \in \mathbb{C} \tag{3.26}$$

avec

$$\begin{aligned} \boldsymbol{\mathcal{A}}_{\boldsymbol{\mathcal{Z}}} &= \boldsymbol{\mathcal{K}}\boldsymbol{\mathcal{J}}^{-1}\boldsymbol{\mathcal{M}} + \boldsymbol{\mathcal{P}}, & \boldsymbol{\mathcal{B}}_{\boldsymbol{\mathcal{Z}}} &= \boldsymbol{\mathcal{K}}\boldsymbol{\mathcal{J}}^{-1}\boldsymbol{\mathcal{M}}_t + \boldsymbol{\mathcal{M}}_x, \\ \boldsymbol{\mathcal{C}}_{\boldsymbol{\mathcal{Z}}} &= \boldsymbol{\mathcal{L}}\boldsymbol{\mathcal{J}}^{-1}\boldsymbol{\mathcal{M}} + \boldsymbol{\mathcal{R}}, & \boldsymbol{\mathcal{D}}_{\boldsymbol{\mathcal{Z}}} &= \boldsymbol{\mathcal{L}}\boldsymbol{\mathcal{J}}^{-1}\boldsymbol{\mathcal{M}}_t + \boldsymbol{\mathcal{M}}_y. \end{aligned} \tag{3.27}$$

Démonstration. La fonction de transfert s'obtient, à partir de la matrice $\boldsymbol{Z}$ d'une SIF, par l'équation (1.44), et par correspondance avec la matrice $\mathbf{Z}$ des coefficients de $\widetilde{\mathcal{H}}_{\boldsymbol{\varepsilon}}$ donnée par l'équation 3.25. □

Finalement, le corollaire 3.1 donne le vecteur d'intervalles de l'erreur globale (ou les moments d'ordre 1 et deux si les erreurs sont modélisées par des bruits) en l'appliquant au filtre $\widetilde{\mathcal{H}}_{\boldsymbol{\varepsilon}}$ et à l'entrée $\begin{pmatrix} \boldsymbol{u}(k) \\ \boldsymbol{\varepsilon_t}(k) \\ \boldsymbol{\varepsilon_x}(k) \\ \boldsymbol{\varepsilon_y}(k) \end{pmatrix}$ (et non seulement $\boldsymbol{\varepsilon}(k)$ comme c'est le cas pour $\mathcal{H}_{\boldsymbol{\varepsilon}}$).

Remarque 3.7. *Étant donné que cette approche de propagation de l'ensemble des erreurs est très récente dans nos travaux (la SIF de $\widetilde{\mathcal{H}}_{\boldsymbol{\varepsilon}}$ et ce qui en découle ont été déterminés à un moment avancé de la rédaction de cette thèse), on préférera utiliser l'analyse de l'erreur de calcul par le filtre $\mathcal{H}_{\boldsymbol{\varepsilon}}$. Cependant, on montrera dans l'exemple 4.5.1 une application de cette analyse de l'erreur par le filtre $\widetilde{\mathcal{H}}_{\boldsymbol{\varepsilon}}$ dans le cas d'un state-space.*

Pour appliquer ces nouvelles propriétés, il faut comprendre comment sont calculés les algorithmes de filtre linéaires. Cette information nous sera utile notamment pour déterminer comment évaluer les erreurs de calculs.

3.3 Des filtres aux produits scalaires

Cette partie présente la relation étroite un filtre linéaire et un produit scalaire constantes par variables. En effet ce genre de produit scalaire est à la base de tout algorithme de filtre LTI. Ensuite nous verrons que ces produits scalaires de filtres ont certaines propriétés intéressantes et nécessaires pour la méthode utilisée pour implémenter un filtre.

3.3.1 Relation entre filtres et produits scalaires

On a vu dans le chapitre 1 qu'un filtre peut être exprimé par différentes réalisations (une infinité en tenant compte des changements de bases possibles), et tous ces algorithmes sont en fait soit un, soit une succession de sommes de produits (*Sum-of-Products*, SoP), du type :

$$s = \sum_{i=1}^{N} c_i \times v_i = \sum_{i=1}^{N} p_i \tag{3.28}$$

où c_i est une constante, v_i une variable, et $p_i = c_i \times v_i$ pour $i = 1, \ldots, N$, et s est le résultat du produit scalaire.

Il est en effet clair, si on prend l'algorithme de calcul totalement développé correspondant à la SIF, donné par les équations (1.47), qu'il s'agit d'une succession de SoP.

Cependant, on peut remarquer dans les algorithmes de filtres, les variables ont une informations temporelles (la valeur d'un signal à un instant k). En considérant ces calculs comme des SoP comme dans l'équation (3.28), on constate qu'on perd l'information temporelle en notant simplement v_i les variables au lieu de $v_i(k)$, mais cette information est inutile pour faire le produit à calculer. De plus, on perd également l'information de rebouclage des calculs, mais cette information est également inutile car gérée à part (par le filtre $\mathcal{H}_\varepsilon$ décrit en 3.2).

Il faut noter enfin que ces SoP ne sont pas de simples produits scalaires (c.-à-d. une expression mathématiques calculant un résultat avec pour seules informations des constantes et des variables), nous avons des informations supplémentaires dues à la particularité des filtres linéaires, comme le décrit la partie suivante.

3.3.2 Spécificités des produits scalaires de filtres

L'arithmétique d'intervalle n'est pas applicable dans le calcul des SoP (si la somme des valeurs absolues des coefficients est plus grande que 1), car cet arithmétique fait grandir les intervalles lors des calculs et ne prend donc pas en compte la corrélation entre les variables. De même, l'arithmétique affine pourrait marcher [72], mais est inutile par l'utilisation du WCPG qui nous permet, par la proposition 3.2, de déterminer au plus juste l'intervalle d'existence du résultat d'un SoP. À partir de ce résultat, on peut appliquer une propriété de l'arithmétique en complément à deux (parfois appelée règle de Jackson) [50] sur l'ensemble des formats intermédiaires de nos oSoP.

Cette règle énonce que, dans une succession de sommes en complément à deux, certains résultats intermédiaires peuvent déborder du format donné, si le résultat est représentable sur ce format alors il est valide. Illustrons cette propriété due à l'arithmétique modulaire par un exemple.

Exemple 3.1. *Soit S la somme de trois entiers de largeurs 8 bits en complément à deux, par exemple* $104+82-94$*. Le résultat est alors* 92*, qui est représentable sur 8 bits dans l'intervalle* $[-128;127]$*. Si on calcule d'abord* $104+82$*, le résultat,* 186*, n'est pas représentable sur cet intervalle, il y a donc débordement et l'arithmétique modulaire donne (en réinterprétant les bits restants comme un nouveau nombre en complément à deux)* -70 *sur 8 bits. Le dernier calcul,* $-70-94$ *produit également un débordement (*$-164 \notin [-128;127]$*), et par arithmétique modulaire on obtient, de la même manière,* 92 *qui est bien le résultat attendu.*

Cette règle est également applicable à l'arithmétique virgule fixe [70]. Pour un SoP défini comme dans l'équation (3.28), avec (m_s, ℓ_s) le format du résultat connu par la proposition 3.2, cela signifie qu'on peut supprimer de chaque p_i les bits dont la position est plus grands que m_s, puis sommer les bits restants sans considérer les bits de retenues plus grand que m_s. Puisque s est connu pour être au format (m_s, ℓ_s), le résultat est valide, les bits supprimés et les retenues non considérées ne peuvent effectivement pas influencer les bits de poids plus faibles.

Remarque 3.8. *On peut démontrer plus formellement cette affirmation. Pour cela, il est nécessaire de réinterpréter les produits en des nombres non signés, de calculer la somme modulo* 2^{m_s} *(pour supprimer les bits de poids forts des* p_i *et les éventuelles retenues indésirées) des nombres positifs obtenus, et de réinterpréter la somme dans la représentation signée.*

L'opération de passage de la représentation signée à la représentation non signée est décrite par l'équation suivante pour un entier X *:*

$$\widetilde{X} = (X + 2^{m+1}) \mod 2^{m+1}, \tag{3.29}$$

où $\widetilde{X}$ *est la réinterprétation de* X *en non signé. L'opération inverse s'obtient par :*

$$X = ((\widetilde{X} + 2^{m_s}) \mod 2^{m+1}) - 2^{m_s}. \tag{3.30}$$

Enfin, la somme de deux termes non signés $\widetilde{X}$ *et* $\widetilde{Y}$ *s'obtient, comme en signé, par complément à deux, en calculant* $(\widetilde{X} + \widetilde{Y}) \mod 2^{m_s}$*.*

Pour appliquer cette méthode à des nombres en virgules fixe, il suffit de l'appliquer à leurs mantisses (toutes alignées sur le même format), et de multiplier le résultat par le facteur d'échelle 2^{ℓ_s}*.*

Remarque 3.9. *Il est important de noter que les équations de la remarque précédentes expriment mathématiquement la réinterprétation des bits restants après suppressions des bits de poids forts superflus. Dans les faits, cette règle est appliqué automatiquement si les opérateurs et registres utilisent l'arithmétique modulaire en cas de débordement, ce qui est généralement le cas. En effet, en langage* `C` *par exemple, si on veut stocker un nombre de plus de 16 bits de large dans une variable définie sur 16 bits par un entier de type* `int16_t`*, la*

variable supprimera automatiquement les bits de positions plus grandes que 16 et réinterprétera les bits restants comme un entier 16 bits.

Les propriétés vues dans ce chapitre vont servir de base de notre méthode d'implémentation. Nous allons maintenant pouvoir décrire la méthodologie d'implémentation, dans le cas *logiciel* (chapitre 4) et dans le cas *matériel* (chapitre 5).

Avant cela, on décrit un exemple fil rouge qui servira à illustrer nos propos tout au long des deux prochains chapitres.

3.4 Exemple de filtre

Soit $\mathcal{H}$ un filtre SISO d'ordre $n = 3$ et H sa fonction de transfert donnée par :

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3}}{1 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3}}, \quad \forall z \in \mathbb{C} \tag{3.31}$$

avec

$$\begin{array}{rclrcl} b_0 & = & -0.602538, & a_1 & = & -0.317866 \\ b_1 & = & 0.0217266, & a_2 & = & 0.251000 \\ b_2 & = & -0.0965601, & a_3 & = & 0.0306691 \\ b_3 & = & -0.0514917, & & & \end{array} \tag{3.32}$$

Ce filtre a été obtenu en utilisant la commande `drss` de `Matlab` (*discrete randomized state-space*), qui renvoie un filtre stable (pôles de module strictement inférieur à 1) aléatoire.

L'ensemble des valeurs données dans cette section sont des valeurs approchées, exceptés les 0 et les 1 qui sont des valeurs exactes (sauf précision contraire).

Ce filtre sera exprimé, selon les propriétés à illustrer, par deux différentes réalisations, une forme directe I et une représentation d'état.

Dans les deux cas, le signal d'entrée u est une variable tirée aléatoirement à chaque instant k, tel que $u(k) \in [-25; 30]$. En appliquant la proposition 3.2, on obtient l'intervalle pour la sortie $y(k)$, c'est-à-dire $y(k) \in [-23.7637; 19.9824]$ (la borne inférieure est une valeur approchée avec un arrondi vers $-\infty$ tandis que la borne supérieure est une valeur approchée avec un arrondi vers $+\infty$).

3.4.1 Forme Directe I

La réalisation en DFI est obtenue à partir de (1.14) avec les coefficients ci-dessus :

$$y(k) = \sum_{i=0}^{3} b_i u(k-i) - \sum_{i=1}^{3} a_i y(k-i). \tag{3.33}$$

3.4.1.1 Analyse d'erreur

Étant donné que la DFI est une réalisation très simple (elle se compose en effet d'une seule équation), on choisit ici de refaire l'analyse de l'erreur pour déterminer la fonction de transfert de l'erreur $\mathcal{H}_\varepsilon$, plutôt que de passer par l'analyse générale faite sur la SIF, ce qui alourdirait inutilement les notations. La SIF correspondant à la DFI pour cet exemple est néanmoins donnée par la suite.

On souhaite donc déterminer $\mathcal{H}_\varepsilon$. Pour cela, on exprime la DFI qui considère les erreurs de calculs effectuées lors de l'implémentation :

$$y^*(k) = \sum_{i=0}^{3} b_i u(k-i) - \sum_{i=1}^{3} a_i y^*(k-i) + \varepsilon_y(k) \tag{3.34}$$

On note alors $\delta y(k) \triangleq y^*(k) - y(k)$, et d'après (3.33) et (3.34), il vient rapidement :

$$\delta y(k) = \varepsilon_y(k) - \sum_{i=1}^{3} a_i \delta y(k-i) \tag{3.35}$$

que l'on peut réécrire

$$\delta y(k) = \sum_{i=0}^{3} b'_i \varepsilon_y(k-i) - \sum_{i=1}^{3} a_i \delta y(k-i) \tag{3.36}$$

avec

$$b'_0 \triangleq 1, \quad b'_i \triangleq 0 \quad \forall 1 \leqslant i \leqslant 3. \tag{3.37}$$

On a donc une écriture d'un filtre, sous forme d'une réalisation en DFI, entre l'entrée $\varepsilon_y(k)$ (l'erreur de calcul sur $\mathcal{H}$ à l'instant k) et la sortie $\delta y(k)$ (l'impact de l'erreur de calcul à travers le filtre à l'instant k). Ce filtre correspond à $\mathcal{H}_\varepsilon$ et sa fonction de transfert est donnée par :

$$H_\varepsilon(z) = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3}} \tag{3.38}$$

On peut alors calculer le DC-gain et le wors-case-peak-gain de cette fonction de transfert, on a en effet

$$\langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle_{\mathrm{DC}} = 1.037556, \quad \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle_{\mathrm{wcpg}} = 1.722146, \tag{3.39}$$

ce qui nous permettra d'appliquer le corollaire 3.1, en fonction des erreurs de calcul effectuées.

3.4.1.2 SIF

Comme vu précédemment, dans la SIF le vecteur $\boldsymbol{x}$ sert à stocker l'état courant pour l'instant suivant tandis que le vecteur $\boldsymbol{t}$ sert à stocker les calculs intermédiaires qui seront utilisés au même instant. Puisque dans la DFI on calcule $y(k)$ en fonction des $y(k-i)$ et $u(k-i)$ pour $1 \leqslant i \leqslant 3$, on a besoin de stocker les $y(k-i)$ et $u(k-i)$ pour l'instant suivant, ces variables vont donc composer l'état $\boldsymbol{x}(k)$:

$$\boldsymbol{x}(k) \triangleq \begin{pmatrix} y(k-3) \\ y(k-2) \\ y(k-1) \\ u(k-3) \\ u(k-2) \\ u(k-1) \end{pmatrix}. \tag{3.40}$$

Au moment du calcul de l'état pour l'instant $k+1$, on aura $\boldsymbol{x}_3(k+1) = y(k)$, et on devra donc stocker dans l'état le calcul de la sortie. Or, plutôt que de calculer $y(k)$ deux fois (une fois pour la stocker dans $\boldsymbol{x}(k+1)$ et une fois pour la sortie à retourner à l'instant k), on effectue le calcul une fois et on le stocke dans la variable intermédiaire $t(k+1)$ (dans le cas présent il n'y a qu'un seul calcul intermédiaire à stocker donc t est un scalaire). On calcule donc

$$t(k+1) = -\sum_{i=1}^{3} a_i \boldsymbol{x}_{4-i}(k) + \sum_{i=1}^{3} b_i \boldsymbol{x}_{7-i}(k) \tag{3.41}$$

On peut en déduire l'algorithme suivant :

$$t(k+1) \leftarrow -\sum_{i=1}^{3} a_i \boldsymbol{x}_{4-i}(k) + \sum_{i=1}^{3} b_i \boldsymbol{x}_{7-i}(k) \tag{3.42a}$$

$$\boldsymbol{x}_1(k+1) \leftarrow x_2(k) \tag{3.42b}$$

$$\boldsymbol{x}_2(k+1) \leftarrow x_3(k) \tag{3.42c}$$

$$\boldsymbol{x}_3(k+1) \leftarrow t(k+1) \tag{3.42d}$$

$$\boldsymbol{x}_4(k+1) \leftarrow x_5(k) \tag{3.42e}$$

$$\boldsymbol{x}_5(k+1) \leftarrow x_6(k) \tag{3.42f}$$

$$\boldsymbol{x}_6(k+1) \leftarrow u(k) \tag{3.42g}$$

$$y(k+1) \leftarrow t(k+1) \tag{3.42h}$$

Finalement, on a la matrice des coefficients $\boldsymbol{Z}_{DFI}$:

$$\boldsymbol{Z}_{DFI} \triangleq \begin{pmatrix} -\boldsymbol{J} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} = \left(\begin{array}{c:ccccccc} -1 & -a_3 & -a_2 & -a_1 & b_3 & b_2 & b_1 & b_0 \\ \hdashline 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hdashline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right). \tag{3.43}$$

3.4.2 State-Space

On considérera également une réalisation sous forme de state-space. Le state-space suivant a été obtenu en appliquant une transformation par une matrice $\boldsymbol{T}$ aléatoire sur une forme équilibrée :

$$\begin{pmatrix} \boldsymbol{x}(k+1) \\ y(k) \end{pmatrix} = \begin{pmatrix} \boldsymbol{A} & \boldsymbol{b} \\ \boldsymbol{c} & d \end{pmatrix} \begin{pmatrix} \boldsymbol{x}(k) \\ u(k) \end{pmatrix}, \tag{3.44}$$

avec

$$\begin{aligned} \boldsymbol{A} &\triangleq \begin{pmatrix} 17.04616 & 39.74407 & 28.37811 \\ -298.2602 & -695.7694 & -496.7940 \\ 407.5404 & 951.0029 & 679.0411 \end{pmatrix}, \\ \boldsymbol{c} &\triangleq \begin{pmatrix} 58.71038 & 67.44930 & 46.92929 \end{pmatrix}, \\ \boldsymbol{b} &\triangleq \begin{pmatrix} -0.00379708 \\ -0.00549377 \\ 0.00902796 \end{pmatrix}, \quad d \triangleq -0.6025375. \end{aligned}$$

Dans le cas du state-space le passage sous forme SIF est évident, il suffit de prendre $n_t = 0$ et on a :

$$\boldsymbol{Z}_{SS} \triangleq \begin{pmatrix} \boldsymbol{J} & \boldsymbol{M} & \boldsymbol{N} \\ -\boldsymbol{K} & \boldsymbol{P} & \boldsymbol{Q} \\ -\boldsymbol{L} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \boldsymbol{A} & \boldsymbol{b} \\ \cdot & \boldsymbol{c} & d \end{pmatrix} \tag{3.45}$$

3.4.2.1 Analyse d'erreur

On peut ici reprendre l'analyse d'erreur faîte pour la SIF en prenant $n_t = 0$ (et $n_x = 3$ dans cet exemple). D'après les notations des équations (3.14), on a :

$$\boldsymbol{Z}_{SS\varepsilon} \triangleq \begin{pmatrix} \boldsymbol{A} & \boldsymbol{M}_x \\ \boldsymbol{c} & \boldsymbol{M}_y \end{pmatrix} \tag{3.46}$$

avec

$$\boldsymbol{M}_x \triangleq \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad \boldsymbol{M}_y \triangleq \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.47}$$

On déduit de la matrice des coefficients $\boldsymbol{Z}_{SS\varepsilon}$ la fonction de transfert de l'erreur, $\boldsymbol{H}_{\boldsymbol{\varepsilon}}$, en utilisant l'équation (1.24) :

$$\boldsymbol{H}_{\boldsymbol{\varepsilon}}(z) \triangleq \boldsymbol{c}(z\boldsymbol{I}_3 - \boldsymbol{A})^{-1}\boldsymbol{M}_x + \boldsymbol{M}_y, \quad \forall z \in \mathbb{C}. \tag{3.48}$$

On peut alors calculer le DC-gain et le WCPG, de $\mathcal{H}_{\boldsymbol{\varepsilon}}$, en utilisant les équations (1.31) et (1.33). D'une part, le DC-gain de $\mathcal{H}_{\boldsymbol{\varepsilon}}$ est donnée par :

$$\langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}} = \begin{pmatrix} 62.0626 & 94.5898 & 66.6383 & 1 \end{pmatrix} \tag{3.49}$$

et d'autre part on a le WCPG de $\mathcal{H}_\varepsilon$ donné par :

$$\langle\langle \mathcal{H}_\varepsilon \rangle\rangle = \begin{pmatrix} 76.2873 & 124.264 & 87.8531 & 1 \end{pmatrix} \tag{3.50}$$

Ces valeurs nous serons utiles pour appliquer le corollaire 3.1, une fois qu'auront été déterminées les intervalles des erreurs de calculs.

3.5 Conclusion

Ce chapitre nous a permis de poser les briques de bases de notre approches, à savoir l'analyse d'erreur et la spécification des algorithmes de filtres linéaires en somme de produits particuliers.

Si l'évolution d'un signal à travers un filtre était connue et déjà appliquée aux bruits pour analyser la propagation des erreurs de calculs, l'utilisation du WCPG sur une entrée du filtre donnée sous forme d'intervalle est une approche nouvelle. De plus, la formalisation de la propagation des erreurs de calcul ainsi que des erreurs paramétriques est une contribution réalisée durant cette thèse.

L'analyse de l'évolution d'un intervalle au travers d'un filtre permet de déterminer précisément et analytiquement le comportement des erreurs à travers l'implémentation virgule fixe de ce filtre. Cette analyse permet également de déduire, à partir de l'intervalle de départ, l'intervalle de sortie atteignable dans le pire des cas, et donc de déterminer le format virgule fixe de sortie du filtre, et ainsi d'appliquer des règles bien connues d'arithmétique en complément à deux. La règle autorisant les débordements intermédiaires dans une somme de plusieurs termes tant que le résultat est valide sur le format final donné est connue mais n'a jamais été appliquée pour l'implémentation de filtres linéaires.

Le chapitre s'achève par la description d'un exemple fil rouge qui sera repris dans les chapitres suivants concernant les implémentations logicielle d'une part et matérielle d'autre part.

Chapitre 4

Implémentation logicielle de filtres et produits scalaires

"*People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on eachother, like a wall of mini stones.*"
- Donald Knuth

L'objectif de ce chapitre est de présenter la méthode mise en œuvre pour implémenter un filtre linéaire dans le cas *logiciel*, en partant d'une réalisation décrivant le filtre et en allant jusqu'à la génération de code. Cette méthode s'applique en fait à un produit scalaire spécifique aux filtres et est répétée autant de fois qu'il y a de produit scalaire dans la réalisation choisie pour représenter le filtre.

Nous verrons dans ce chapitre qu'une réalisation de filtre peut en effet s'écrire comme une succession de produits scalaires constantes par variables, mais également que ces produits scalaires ont des spécificités (comme le fait de connaître analytiquement le format virgule fixe du résultat). Chaque produit scalaire en précision finie (et dans une implémentation logicielle) peut générer des dégradations différentes suivant l'ordre dans lequel les sommes sont calculées, ce point sera également discuté ici, ainsi que l'évaluation de ces dégradations numériques. Enfin, la génération de code virgule fixe correspondant à un bon schéma d'évaluation (choisi selon des critères définis) sera abordée.

À la suite de la description de la méthode, cette méthode sera, en exemple, appliquée à un filtre décrit dans une réalisation composée de plusieurs produits scalaires.

4.1 Produits scalaires ordonnés

En précision infinie, l'ordre des opérations dans le calcul d'un produit scalaire tel que celui défini en (3.28) n'est pas important, car l'addition est associative et commutative. En précision finie, les choses peuvent êtres différentes. En effet, en *logiciel*, où les opérandes et les résultats ont une largeur fixée par les opérateurs (généralement des puissances de 2), l'addition peut ne pas être associative, comme le met en avant l'exemple suivant.

Exemple 4.1. *On considère trois nombres à additionner avec un additionneur* 4 *bits :* 4, 1.5 *et* 0.5. *On peut sommer ces nombres selon trois différents ordres :*

- $(4 + 1.5) + 0.5$ *: on ajoute d'abord* 4 *et* 1.5 *sur* 4 *bits, on obtient* 5, *puis on ajoute* 0.5 *à* 5 *et le résultat reste inchangé car sur* 4 *bits les deux* 0.5 *(les deux bits en positions* -1*) sont absorbés.*
- $(4 + 0.5) + 1.5$ *: on ajoute d'abord* 4 *et* 0.5 *sur* 4 *bits, on obtient* 4, *puis on ajoute* 1.5 *à* 4 *et on obtient* 5 *une nouvelle fois.*
- $(1.5 + 0.5) + 4$ *: on ajoute d'abord* 1.5 *et* 0.5 *sur* 4 *bits, on obtient* 2, *puis on ajoute* 4 *à* 2 *et on obtient* 6.

Les figures 4.1 et 4.2 illustrent les deux résultats obtenus, 5 *et* 6.

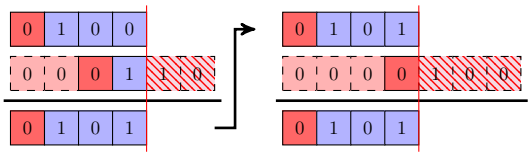


Figure 4.1 – $(4 + 1.5) + 0.5 = 5$.

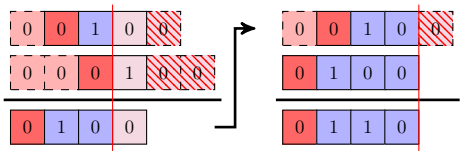


Figure 4.2 – $(1.5 + 0.5) + 4 = 6$.

À noter qu'en logiciel, *sur un additionneur 8 bits par exemple, le résultat aurait été le même pour les trois ordres. En* matériel, *on aurait choisit de représenter nos nombres sur* 5 *bits et on n'aurait eu aucune absorption de bits.*

En précision finie il est par conséquent important de considérer l'ordre dans lequel vont être effectuées les additions d'un produit scalaire si on n'a pas assez de bits pour éviter les absorptions. On définit alors la notion de *produit scalaire ordonné*.

Définition 4.1 (Produit scalaire ordonné)**.** *Un produit scalaire ordonné (ou en anglais* ordered-Sum-of-Products*, oSoP) est un produit scalaire dont les additions s'effectuent dans un ordre (ou parenthésage) donné.*

Soit $s = \sum_{i=1}^{4} \boldsymbol{p}_i$ avec $\boldsymbol{p}_i = \boldsymbol{c}_i \times \boldsymbol{v}_i$ un produit scalaire tel que défini en (3.28) avec $N = 4$. Un oSoP issu de ce produit scalaire peut être celui-ci :

$$((\boldsymbol{p}_1 + \boldsymbol{p}_4) + \boldsymbol{p}_2) + \boldsymbol{p}_3, \tag{4.1}$$

ou encore celui-ci

$$(\boldsymbol{p}_1 + \boldsymbol{p}_3) + (\boldsymbol{p}_2 + \boldsymbol{p}_4). \tag{4.2}$$

Au lieu d'une représentation par parenthésages, peu commode au niveau de la lisibilité pour un nombre de produits N élevé, on peut considérer une représentation plus visuelle des différents ordres en utilisant des arbres syntaxiques. Ainsi, un oSoP peut être vu comme un *arbre binaire entier*[1] dans lequel les feuilles sont des paires constantes/variables $\boldsymbol{c}_i/\boldsymbol{v}_i$, le nœud père d'un couple de feuilles est un multiplieur (ce multiplieur calcule alors $\boldsymbol{p}_i = \boldsymbol{c}_i \times \boldsymbol{v}_i$), et les autres nœuds internes sont des additionneurs. Le nœud racine de l'arbre indique simplement de façon visuelle le résultat de la somme calculée et n'est donc pas considéré comme un nœud de l'oSoP à part entière (ce qui remettrait en cause la correspondance oSoP/arbre binaire entier).

La figure 4.3 présente la légende graphique associée aux oSoP.

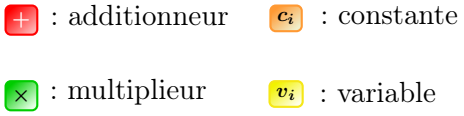


FIGURE 4.3 – Légende des différents noeuds des arbres syntaxiques.

En reprenant les exemples ci-dessus d'oSoP de taille 4 donnés par les équations (4.1) et (4.2), on obtient respectivement les figures 4.4 et 4.5.

Il est alors nécessaire de connaître deux choses, le nombre d'oSoP différents pour un SoP donné, et comment les générer. Si on sait comment les générer, alors en analysant l'algorithme de génération, nous serons capable de dire combien il y en a, la preuve pouvant en effet se faire par construction.

Deux approches de générations des oSoP (ou des arbres binaires entiers) ont été effectuées au début de cette thèse, une approche de construction par récurrence des oSoP et une approche utilisant une pile. La première permet d'expliciter de manière évidente le nombre d'oSoP différents pour un SoP donné.

1. la distinction est faite entre un arbre syntaxique où un nœud possède *au plus* deux fils, et un arbre binaire entier, arbre syntaxique où un nœud possède soit deux fils (noeud interne), soit aucun fils (feuille).

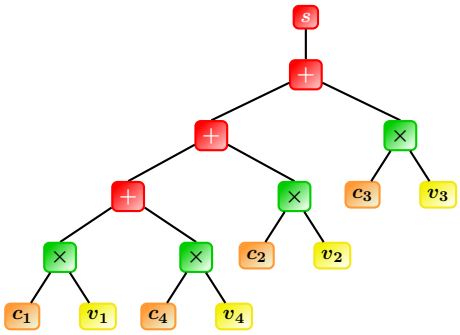


Figure 4.4 – oSoP correspondant au calcul $((\boldsymbol{p}_1 + \boldsymbol{p}_4) + \boldsymbol{p}_2) + \boldsymbol{p}_3$.

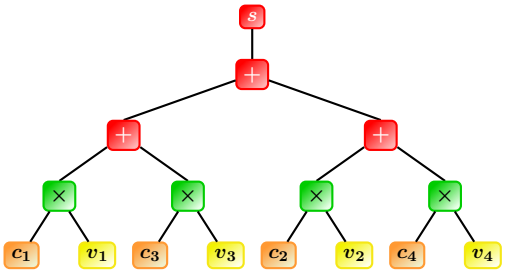


Figure 4.5 – oSoP correspondant au calcul $(\boldsymbol{p}_1 + \boldsymbol{p}_3) + (\boldsymbol{p}_2 + \boldsymbol{p}_4)$.

À noter : pour la description des deux approches, étant donné qu'une constante $\boldsymbol{c}_i$ ne peut être que multipliée avec sa variable associée $\boldsymbol{v}_i$, on remplacera le multiplieur et les deux feuilles $\boldsymbol{c}_i$ et $\boldsymbol{v}_i$ par une *feuille produit* $\boldsymbol{p}_i$ (voir figure 4.6).

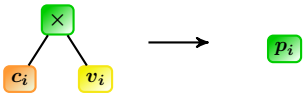


Figure 4.6 – Passage d'un multiplieur constante/variable à une feuille produit.

Génération par récurrence : Cette approche se fait par récurrence, en partant d'un arbre à deux multiplieurs $\boldsymbol{p}_1$ et $\boldsymbol{p}_2$ et un additionneur puis en ajoutant (en *branchant*) successivement les feuilles $\boldsymbol{p}_3$ à $\boldsymbol{p}_N$ aux différents endroits possibles. Brancher une feuille consiste à ajouter un additionneur dont les deux nœuds fils sont la feuille à ajouter et un nœud déjà existant. La feuille $\boldsymbol{p}_3$ peut alors se brancher à trois endroits :

- entre $\boldsymbol{p}_1$ et l'additionneur existant : on obtient la somme $(\boldsymbol{p}_1 + \boldsymbol{p}_3) + \boldsymbol{p}_2$
- entre $\boldsymbol{p}_2$ et l'additionneur existant : on obtient la somme $\boldsymbol{p}_1 + (\boldsymbol{p}_2 + \boldsymbol{p}_3)$
- au-dessus de l'additionneur : on obtient la somme $(\boldsymbol{p}_1 + \boldsymbol{p}_2) + \boldsymbol{p}_3$

La figure 4.7 illustre ce passage d'un arbre à deux multiplieurs à trois arbres à trois multiplieurs. On remarque que les seuls endroits où on peut ajouter notre nouvelle branche sont les arêtes (voir les points rouges sur l'arbre à deux multiplieurs sur la figure). En effet, le nouvel additionneur ne peut que se situer au-dessus d'un multiplieur ou d'un autre additionneur.

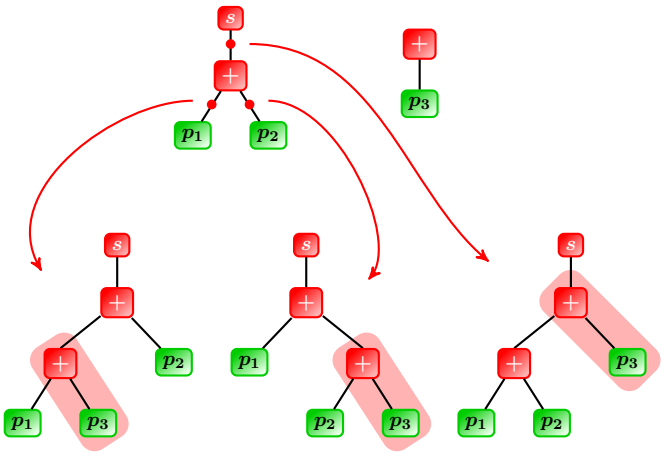


FIGURE 4.7 – Passage d'un oSoP de taille $N = 2$ à trois oSoP de taille $N = 3$.

Par conséquent, pour un arbre de taille k, il y a autant d'oSoP possibles à l'étape $k + 1$ que d'arêtes dans cet arbre, c'est-à-dire $2k - 1$ puisque nous considérons des arbres binaires entiers (on ne compte pas ici les arêtes des constantes et des variables puisque l'additionneur ne peut pas se connecter sous un multiplieur).

Proposition 4.1 (Nombre d'oSoP de taille N [60])**.** *Pour un produit scalaire de taille N il y a*

$$\prod_{i=1}^{N-1} 2i - 1 \tag{4.3}$$

différents oSoP possibles.

Ce nombre est également appelé double factorielle des nombres impairs [2].

Démonstration. La preuve se fait par récurrence. Il y a un seul oSoP de taille 2 et à partir d'un oSoP de taille k on peut construire $2k - 1$ oSoP de taille $k + 1$ en connectant la nouvelle branche sur toutes les arêtes de l'arbre. □

2. La suite des nombres décrite par l'équation (4.3) pour les différentes valeurs successive de N est répertoriée dans *the On-Line Encyclopedia of Integer Sequences* : `http://oeis.org/A001147`

Remarque 4.1 (Nombre de Catalan)**.** *Le n-ième nombre de Catalan (du mathématicien belge Eugène Charles Catalan) est défini par :*

$$C_n \triangleq \frac{1}{n+1}\binom{2n}{n} \tag{4.4}$$

En combinatoire, ce nombre a de nombreuses propriétés, il correspond notamment au nombre d'arbres binaires entiers à $n+1$ feuilles. Pourtant, appliqué à un SoP de taille N, C_{N-1} est plus petit que $\prod_{i=1}^{N-1} 2i-1$, le terme donné dans la proposition 4.1. En fait, C_{N-1} ne donne pas l'ensemble de tous les arbres, toutes permutations comprises, ce nombre donne juste, pour N termes triés, les différentes façons de connecter les branches sans permuter les termes à sommer. Par exemple, pour $N = 4$ on a $C_3 = 5$, et la figure 4.8 montre les cinq arbres binaires possibles sans permuter les produits $\boldsymbol{p}_i$. On peut également utiliser la notation par parenthésages :

$$\begin{gathered}((\boldsymbol{p}_1+\boldsymbol{p}_2)+\boldsymbol{p}_3)+\boldsymbol{p}_4\\ (\boldsymbol{p}_1+(\boldsymbol{p}_2+\boldsymbol{p}_3))+\boldsymbol{p}_4\\ \boldsymbol{p}_1+((\boldsymbol{p}_2+\boldsymbol{p}_3)+\boldsymbol{p}_4)\\ \boldsymbol{p}_1+(\boldsymbol{p}_2+(\boldsymbol{p}_3+\boldsymbol{p}_4))\\ (\boldsymbol{p}_1+\boldsymbol{p}_2)+(\boldsymbol{p}_3+\boldsymbol{p}_4).\end{gathered}$$

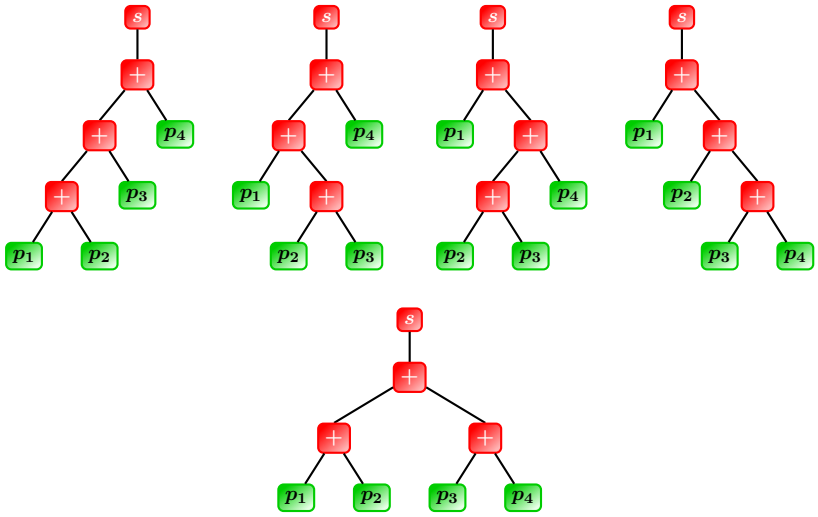


Figure 4.8 – Les cinq oSoP considérés par le nombre de catalan C_3.

À titre d'exemple, l'ordre $(\boldsymbol{p}_1+\boldsymbol{p}_3)+(\boldsymbol{p}_2+\boldsymbol{p}_4)$ n'est pas pris en compte. En

fait, en appliquant la formule (4.3) on obtient 15 oSoP différents, qui sont :

$$
\begin{array}{lll}
((\boldsymbol{p}_1+\boldsymbol{p}_2)+\boldsymbol{p}_3)+\boldsymbol{p}_4 & ((\boldsymbol{p}_2+\boldsymbol{p}_3)+\boldsymbol{p}_1)+\boldsymbol{p}_4 & (\boldsymbol{p}_1+\boldsymbol{p}_2)+(\boldsymbol{p}_3+\boldsymbol{p}_4) \\
((\boldsymbol{p}_1+\boldsymbol{p}_2)+\boldsymbol{p}_4)+\boldsymbol{p}_3 & ((\boldsymbol{p}_2+\boldsymbol{p}_3)+\boldsymbol{p}_4)+\boldsymbol{p}_1 & (\boldsymbol{p}_1+\boldsymbol{p}_3)+(\boldsymbol{p}_2+\boldsymbol{p}_4) \\
((\boldsymbol{p}_1+\boldsymbol{p}_3)+\boldsymbol{p}_2)+\boldsymbol{p}_4 & ((\boldsymbol{p}_2+\boldsymbol{p}_4)+\boldsymbol{p}_1)+\boldsymbol{p}_3 & (\boldsymbol{p}_2+\boldsymbol{p}_3)+(\boldsymbol{p}_1+\boldsymbol{p}_4) \\
((\boldsymbol{p}_1+\boldsymbol{p}_3)+\boldsymbol{p}_4)+\boldsymbol{p}_2 & ((\boldsymbol{p}_2+\boldsymbol{p}_4)+\boldsymbol{p}_3)+\boldsymbol{p}_1 & \\
((\boldsymbol{p}_1+\boldsymbol{p}_4)+\boldsymbol{p}_2)+\boldsymbol{p}_3 & ((\boldsymbol{p}_3+\boldsymbol{p}_4)+\boldsymbol{p}_1)+\boldsymbol{p}_2 & \\
((\boldsymbol{p}_1+\boldsymbol{p}_4)+\boldsymbol{p}_3)+\boldsymbol{p}_2 & ((\boldsymbol{p}_3+\boldsymbol{p}_4)+\boldsymbol{p}_2)+\boldsymbol{p}_1 &
\end{array}
$$

L'inconvénient de cette construction par récurrence est qu'il faut construire tous les oSoP des étapes précédentes ($i = 1, \ldots, N-1$) pour construire les oSoP de taille N, ce qui peut prendre, suivant l'implémentation et N, beaucoup d'espace mémoire. Par contre, il n'y a pas de condition à respecter lors de la génération et donc l'algorithme de construction en lui-même est assez rapide s'il n'est pas limité par la mémoire disponible.

Génération par utilisation d'une pile : Cette approche consiste à considérer une pile contenant au départ une liste des N produits $\boldsymbol{p}_i$, et à effectuer le traitement suivant :

- on prend la liste au sommet de la pile, puis on crée de nouvelles listes à partir de celle-ci en remplaçant deux termes par leur somme. Par exemple, si la liste dépilée est $[\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3]$, alors les listes crées sont $[(\boldsymbol{p}_1+\boldsymbol{p}_2), \boldsymbol{p}_3]$, $[(\boldsymbol{p}_1+\boldsymbol{p}_3), \boldsymbol{p}_2]$ et $[\boldsymbol{p}_1, (\boldsymbol{p}_2+\boldsymbol{p}_3)]$. Les listes ainsi créées sont empilées dans la pile.
- si une liste créée ne possède qu'un seul élément, alors cet élément est un oSoP et il n'est pas replacé dans la pile.

La méthode est résumée dans l'algorithme 3. Dans cet algorithme, certaines fonctions sont utilisées :

- $mg(oSoP)$: retourne l'indice du multiplieur $\boldsymbol{p}_i$ le plus à gauche du sous-arbre oSoP (on parcourt oSoP en choisissant toujours les fils gauches, jusqu'à la feuille la plus à gauche) ;
- $Longueur(L)$: retourne le nombre d'éléments de la liste L ;
- $Copier(L)$: retourne une copie de la liste L ;
- $Retirer(osop, L)$: supprime l'élément $osop$ contenu dans la liste L ;
- $Inserer(osop, L)$: insère l'élément $osop$ au début de la liste L ;
- $Ajouter(osop, L)$: ajoute l'élément $osop$ au bout de la liste L.

De plus, on considère que le premier élément d'une liste est indexé par 1, ainsi $L[1]$ désigne le premier élément (sous-oSoP) de la liste L.

La principale difficulté de cette approche est d'éviter les doublons, c'est-à-dire les oSoP qui sont équivalents (à commutativité de l'addition près) à d'autres oSoP déjà générés. La proposition suivante énonce que l'algorithme 3 génère bien tous les oSoP possibles et aucun doublon.

Algorithme 3: Génération d'arbres par utilisation d'une pile

Données: Une pile P contenant une liste de tous les p_i ordonnés
Résultat: La liste oSoP contenant la liste de tous les arbres générés

```
Tant que P n'est pas vide Faire
    # On dépile le premier élément de P et on le stocke dans une liste L
    L ← P[1] ;
    # On parcourt tous les tuples (i, j) d'indices de L
    Pour j de 2 à Longueur(L) Faire
        Pour i de 1 à j Faire
            # On évite les doublons
            Si mg(L[i]) ⩾ mg(L[1]) Alors
                # On crée la liste [(L[i] + L[j]), ...]
                L' ← Copier(L) ;
                op_i ← L'[i] ;
                op_j ← L'[j] ;
                Retirer(op_i, L') ;
                Retirer(op_j, L') ;
                Inserer((op_i + op_j), L') ;
                # Si L' contient un oSoP fini (liste de taille 1), on l'ajoute
                à la liste résultat, sinon on le met en haut de la pile P
                Si Longueur(L') = 1 Alors
                    Ajouter(L', oSoP) ;
                Sinon
                    Ajouter(L', P) ;
                Fin
            Fin
        Fait
    Fait
Fait
Retourner oSoP
```

Proposition 4.2. *L'algorithme 3 génère tous les oSoP possibles, et la condition $mg(L[i]) \geqslant mg(L[1])$ suffit à garantir l'absence de doublons dans les oSoP générés.*

Démonstration. La démonstration est donnée en annexe C.3. □

Proposition 4.3. *La pile utilisée dans l'algorithme 3 contient, au cours de l'exécution de l'algorithme pour générer les oSoP de taille N, au plus*

$$-(N-3) + \sum_{i=0}^{N-3} \binom{N-i}{2} \tag{4.5}$$

listes de sous-oSoP, où $\binom{N}{k}$ *représente le coefficient binomial* k *parmi* N*.*

Démonstration. À la première étape de l'algorithme, on a une liste de N éléments à partir de laquelle on crée et empile une liste par paire possible, soit $\binom{N}{2}$ listes de tailles $N-1$. Ensuite, à chaque étape, on dépile une liste de taille k et on crée de la même manière *au plus* (il faut respecter la condition de la proposition 4.2 pour générer une liste) $\binom{k}{2}$ listes de tailles $k-1$. Le pire cas se produit si la condition est toujours respectée et donc si on génère tous les oSoP, mêmes les doublons. Pour $k = 3$, on génère alors trois oSoP et la taille maximale de la pile est atteinte, on a alors :

$$\binom{N}{2} - 1 + \binom{N-1}{2} - 1 + \binom{N-2}{2} + \ldots - 1 + \binom{4}{2} - 1 + \binom{3}{2}. \tag{4.6}$$

□

Remarque 4.2. *On peut montrer que* (4.5) *est égal à* $\frac{N(N^2-1)}{6} + 2 - N$*. Une récurrence peut aisément montrer que* $\sum_{i=0}^{N-3} \binom{N-i}{2} = \frac{N(N^2-1)}{6} - 1$ *et il suffit ensuite de soustraire* $N-3$ *au résultat. Ceci nous permet de déduire une majoration assez large de* (4.5) *par* N^3*.*

La figure 4.9 illustre cet algorithme pour des oSoP possédant trois multiplieurs. À la première étape on part de la liste de départ avec nos trois multiplieurs et on crée trois listes possédant chacune un oSoP partiel et un multiplieur. Les trois listes sont empilées dans la pile. À la deuxième étape on prend la pile au sommet de la pile, il y a une seule somme possible, qui crée alors un oSoP qui est retourné. On répète cette étape pour les deux autres listes afin d'obtenir les deux autres oSoP.

Un des avantages de l'utilisation d'une pile est qu'il n'est pas nécessaire de stocker en mémoire tous les arbres de taille $N-1$, la pile stocke des sous-arbres de tailles variées en nombre négligeable (voir proposition 4.3). De plus, dans la suite on va vouloir effectuer un traitement (propagation de formats, d'intervalles, ou encore d'erreurs) sur nos arbres, et ce traitement peut être effectué sur un sous-arbre de la pile et sera donc effectué une seul fois pour tous les arbres possédant ce sous-arbre traité. Par conséquent, même si l'algorithme de génération est légèrement plus long, le temps gagné en connectant des sous-arbres déjà traités ensemble rend cette méthode globalement plus rapide que la première décrite.

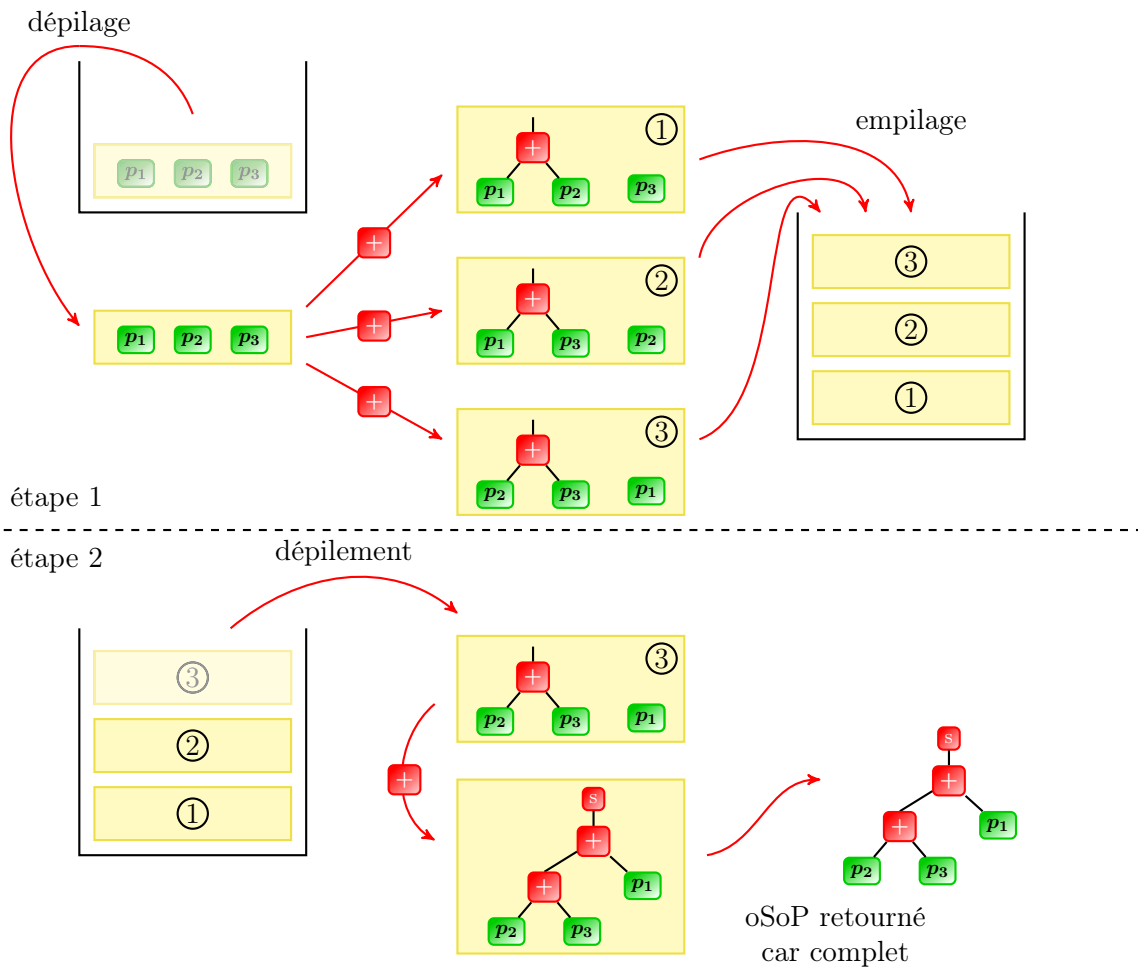


Figure 4.9 – Les deux premières étapes de la génération des oSoP par l'utilisation d'une pile, pour $N = 3$.

4.2 Propagation

Nous avons vu dans la partie précédente qu'il était nécessaire de considérer les différents ordres de sommations pour nos produits scalaires de filtres étant donné que ceux-ci pouvaient avoir un impact sur le résultat final en précision finie. Nous avons également vu comment générer ces produits scalaires ordonnés et comment les représenter à l'aide d'arbres binaires entiers.

L'intérêt de la génération des différents oSoP est de déterminer la dégradation numérique globale qui sera effectuée sur chacun d'eux, afin d'obtenir celui qui produit la plus petite dégradation numérique. La différence d'erreur d'un oSoP à l'autre provient des additions, et donc des formatages (et plus préci-

sément des arrondis des bits de poids faibles, correspondant en logiciel à des décalages à droite) que l'ont effectue sur les opérandes pour les aligner sur un format commun. Or, nous connaissons les constantes (ce sont les coefficients du filtre dans la réalisation choisie), nous connaissons également l'intervalle dans lequel vit chaque variable d'entrée et de sortie (d'après la proposition 3.2), par conséquent nous pouvons propager ces informations à travers nos arbres pour déterminer les différents décalages de chacun d'eux, et ainsi déterminer la dégradation numérique globale de chacun.

La première approche adoptée pour propager les informations des constantes et des variables à travers les différentes oSoP a été de propager leurs formats virgule fixe [71]. En effet, dans cette approche on supposait que seuls les FPF des constantes et des variables étaient connus, et on les propageait à travers nos oSoP en utilisant certaines règles de propagations définies sur les opérateurs (additionneurs et multiplieurs) entre les opérandes (entrées des opérateurs) et le résultat des opérations. Cela consistait en fait à considérer le pire cas pour les constantes et pour les variables et donc le pire cas pour la propagation au travers des opérations.

Afin d'être plus précis sur la détection des formatages et sur la propagation des informations dans les oSoP, la seconde approche adoptée a été de propager les intervalles plutôt qu'uniquement les formats virgule fixe. Les règles de propagations pour la multiplication et l'addition découlent alors directement des formalismes décrits en 2.3 dans les cas à largeur des opérateurs fixée :

- Multiplication : le format du produit est donné par les équations (2.38) et (2.39), et l'intervalle par l'équation (2.41). Si la largeur du résultat de la multiplication est plus petit que la somme des largeurs des opérandes (par exemple pour un schéma $16 \times 16 \rightarrow 16$), alors un re-formatage de la variable est appliqué.
- Addition : le format commun pour les opérandes est calculé d'après l'algorithme 2 et l'intervalle d'après l'équation (2.36), en tenant compte de la cible.

Exemple : propagation des formats dans l'oSoP complet

On considère à nouveau notre exemple fil rouge (voir 3.4). Supposons que la réalisation choisie est la DFI de l'équation (3.33). On a donc un seul produit scalaire, et on génère l'ensemble des oSoP possibles pour ce produit scalaire, et parmi ceux-ci on en sélectionne un pour illustrer la propagation (figure 4.10). Au départ, les intervalles d'existence des variables et les valeurs des constantes sont connus, et on suppose de plus que les variables et les constantes sont codées sur 16 bits. De même, les multiplieurs considérés ici sont des multiplieurs $16 \times 16 \rightarrow 16$. Enfin, les nombres indiqués dans les noeuds constantes (en orange) sont les mantisses sur 16 bits des constantes a_i et b_i.

En appliquant ces règles de propagation sur l'oSoP de la figure 4.10, on obtient l'oSoP pleinement paramètré de la figure 4.11. Sur cet oSoP ne sont

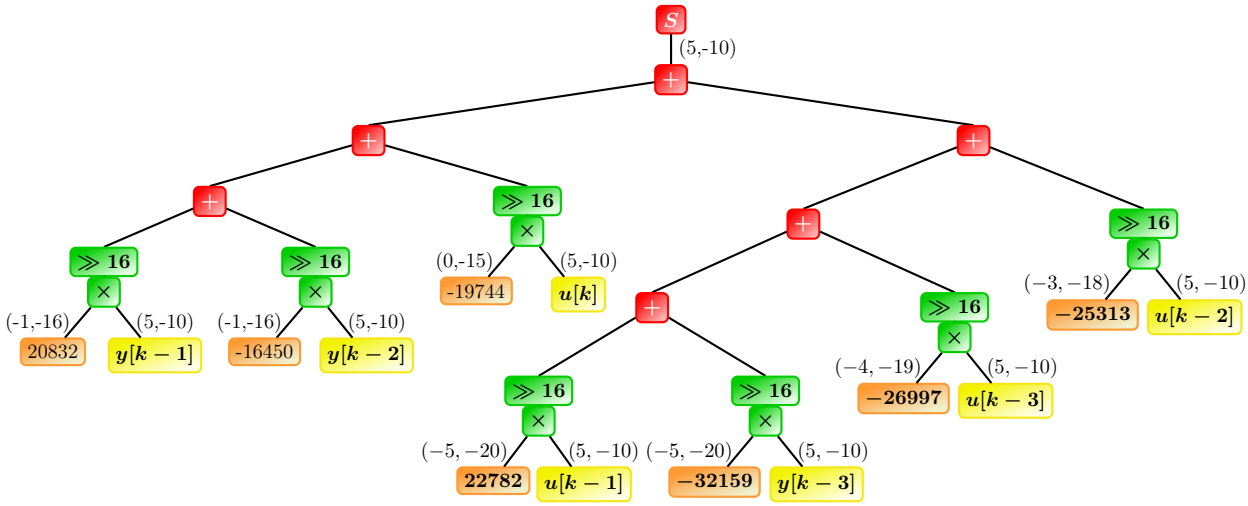


Figure 4.10 – oSoP avant propagation des formats.

représentés que les formats pour des raisons de lisibilité, mais on connait l'ensemble des intervalles intermédiaires.

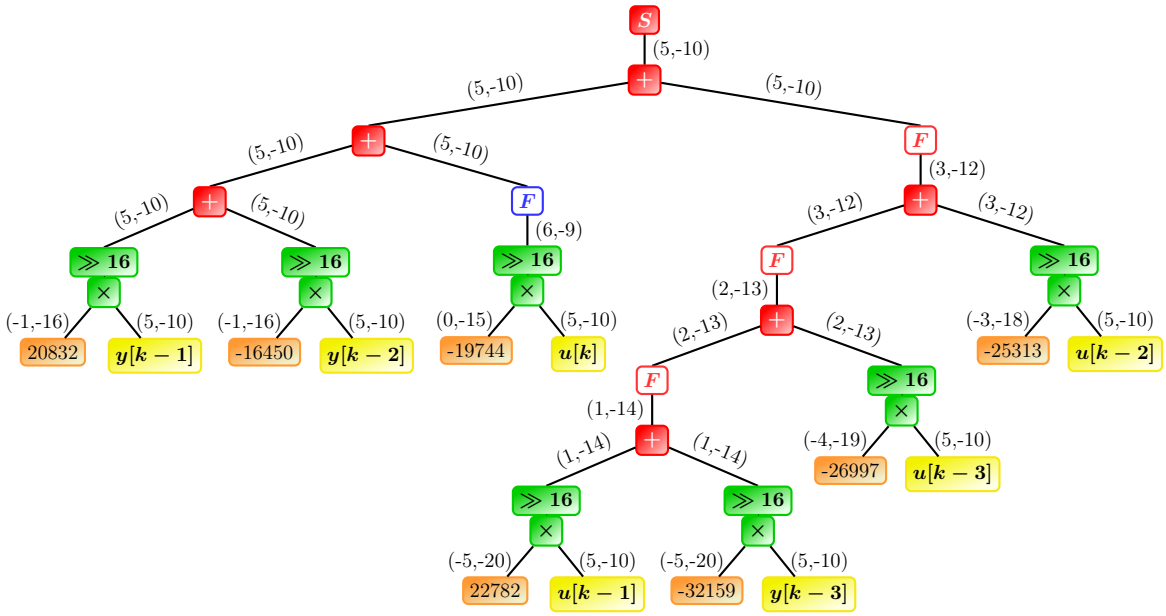


Figure 4.11 – oSoP après propagation des intervalles.

Cette approche de la propagation des informations est logiquement plus fine que la première approche brièvement décrite précédemment, puisque cette

dernière considérait toujours le pire cas sans prendre en compte les valeurs des constantes et les bornes des intervalles.

L'étape suivante consiste à évaluer les valeurs des décalages détectés afin d'avoir une valeur de la dégradation numérique globale de chaque oSoP.

4.3 Évaluation de l'erreur et choix d'un oSoP

On a vu dans la partie précédente comment propager des intervalles à travers les différents oSoP, et que ces propagations nécessitaient des formatages, dont certains produisaient des arrondis, et donc des erreurs. Nous allons voir maintenant comment évaluer ces erreurs d'arrondis, en utilisant les descriptions des erreurs vues en 2.4.2.

Les valeurs des différents formatages apparaissant dans les oSoP suite à la propagation des informations ne sont pas corrélées. En effet, on a vu dans la partie 3.2 qu'on pouvait décomposer le filtre implémenté en précision finie en deux filtres indépendants, le filtre exact $\mathcal{H}$ et le filtre de l'erreur $\mathcal{H}_\varepsilon$. Cette décomposition permet de ne calculer que les erreurs de calculs sur le ou les oSoP, et l'erreur finale, comprenant le rebouclage (la corrélation) de ces erreurs de calcul, est ensuite obtenue par le corollaire 3.1. Par conséquent, si l'erreur de calcul globale $\varepsilon(k)$ d'un oSoP se calcule comme la somme des valeurs des erreurs de chaque formatage, il est important de noter que l'erreur finale sur un filtre se calcule en appliquant le corollaire 3.1 sur $\varepsilon(k)$. On peut également noter que le filtre $\mathcal{H}_\varepsilon$ ne dépend que du choix de la réalisation (on a vu qu'on pouvait l'obtenir directement à partir de la forme implicite du filtre $\mathcal{H}$), mais que $\varepsilon(k)$ dépend du choix de l'implémentation (choix de l'oSoP pour représenter un SoP, propagation des formats ou des intervalles).

Pour calculer la valeur d'un formatage, on considère deux approches, celle du traitement du signal utilisant des bruits et celle de l'arithmétique utilisant des intervalles d'erreurs.

4.3.1 Bruit de quantification

On a vu en 2.4.2.2 que le bruit de quantification est une modélisation statistique d'une erreur d'arrondi, interprétant cette erreur comme un bruit blanc uniformément distribué. Le tableau 2.3 donne les formules pour les moments d'ordre un et deux de ce bruit. Dans le cas présent où l'on considère les erreurs d'arrondi dues aux formatages, seul le cas des bruits à amplitude discrète nous intéresse.

On note F_i le formatage d'une variable sur le format (m_i, ℓ_i) impliquant un arrondi de d_i bits, et μ_i et σ_i^2 les moments d'ordre un et deux de l'erreur commise par le formatage F_i (déterminés d'après le tableau 2.3). Alors, en notant μ_s et σ_s^2 les moments de l'erreur globale sur le résultat s, on a :

– dans le cas d'un arrondi par troncature :

$$\mu_s = \sum_{i=1}^{n_f} 2^{\ell_i - 1}(1 - 2^{-d_i}) \tag{4.7}$$

$$\sigma_s^2 = \sum_{i=1}^{n_f} \frac{2^{2\ell_i}}{12}(1 - 2^{-2d_i}) \tag{4.8}$$

– dans le cas d'un arrondi au plus proche :

$$\mu_s = \sum_{i=1}^{n_f} 2^{\ell_i - d_i - 1} \tag{4.9}$$

$$\sigma_s^2 = \sum_{i=1}^{n_f} \frac{2^{2\ell_i}}{12}(1 - 2^{-2d_i}) \tag{4.10}$$

avec n_f le nombre de formatages impliquant des arrondis présents dans l'oSoP.

Exemple 4.2. *À titre d'exemple, reprenons l'oSoP de la figure 4.11 en considérant que les arrondis se font par troncature. Commençons par un seul formatage : on a dans l'oSoP un formatage pour passer du format* $(1, -14)$ *au format* $(2, -13)$*. En utilisant le tableau 2.3 avec* $\ell = -13$ *et* $d = 1$*, on a les moments d'ordre un et deux, notés respectivement* μ *et* σ^2*, suivants :*

$$\mu = 2^{-14}(1 - 2^{-1}) = 2^{-15} \tag{4.11}$$

$$\sigma^2 = \frac{2^{-26}}{12}(1 - 2^{-2}) = 2^{-30} \tag{4.12}$$

On applique maintenant cette méthode à chaque formatage de l'oSoP de la figure 4.11, on obtient :

$$\mu_s = 4.57764e\text{-}04, \tag{4.13}$$

$$\sigma_s^2 = 7.91624e\text{-}08. \tag{4.14}$$

4.3.2 Intervalle d'erreur

Les intervalles d'erreur (voir 2.4.2.3) donnent une plage précise des valeurs que peut prendre l'erreur commise par un formatage. Le tableau 2.4 donne les formules pour les bornes inférieure et supérieure de l'erreur d'arrondi, selon le mode d'arrondi choisi (on considère ici les arrondis en précision finie, à savoir les arrondis de calcul).

En ré-utilisant les notations précédentes, c'est-à-dire (m_i, ℓ_i) pour le format d'arrivée et d_i pour le nombre de bits arrondis, et en notant $[\underline{e}_i; \overline{e}_i]$ l'intervalle de l'erreur commise par le formatage F_i, et $[\underline{e}_s; \overline{e}_s]$ l'intervalle de l'erreur globale sur s, on a :

– dans le cas d'un arrondi par troncature :

$$\underline{e}_s \triangleq \sum_{i=1}^{n_f} \underline{e}_i = \sum_{i=1}^{n_f} -2^{\ell_i}(1-2^{-d_i}) \tag{4.15}$$

$$\overline{e}_s \triangleq \sum_{i=1}^{n_f} \overline{e}_i = 0 \tag{4.16}$$

– dans le cas d'un arrondi au plus proche :

$$\underline{e}_s \triangleq \sum_{i=1}^{n_f} \underline{e}_i = \sum_{i=1}^{n_f} -2^{\ell_i-1}(1-2^{-d_i}) \tag{4.17}$$

$$\overline{e}_s \triangleq \sum_{i=1}^{n_f} \overline{e}_i = \sum_{i=1}^{n_f} 2^{\ell_i-1} \tag{4.18}$$

avec n_f le nombre de formatages impliquant des arrondis présents dans l'oSoP.

Exemple 4.3. *On applique ces formules pour calculer l'intervalle d'erreur global de l'oSoP de la figure 4.11, en considérant que les arrondis se font par troncature. De la même manière que précédemment, commençons par évaluer l'intervalle d'erreur pour le formatage pour passer du format* $(1, -14)$ *au format* $(2, -13)$*, en utilisant le tableau 2.4 avec* $\ell = -13$ *et* $d = 1$*. L'intervalle d'erreur* $[\underline{e}; \overline{e}]$ *est donné par :*

$$\underline{e} = -2^{-13}(1-2^{-1}) = -2^{-14} \tag{4.19}$$

$$\overline{e} = 0 \tag{4.20}$$

On applique maintenant cette méthode à chaque formatage de l'oSoP de la figure 4.11, on obtient :

$$\underline{e}_s = -9.15527e\text{-}04 \tag{4.21}$$

$$\overline{e}_s = 0. \tag{4.22}$$

4.3.3 Autre critère de choix, le parallélisme

Nous venons de voir comment calculer l'impact numérique des arrondis dans un arbre, celui-ci pouvant être vu soit comme un bruit, soit comme l'ajout d'une erreur sous forme d'un intervalle. L'erreur globale calculée sur un oSoP est un critère important pour choisir un oSoP plutôt qu'un autre parmi l'ensemble des oSoP générés. En effet, l'un des enjeux majeurs de l'implémentation en virgule fixe est la précision des calculs, on aura donc tendance à vouloir implémenter le calcul le plus précis. Mais si plusieurs oSoP ont exactement le même impact numérique, lequel choisir ? Autrement formulé, peut-on déterminer, à partir des oSoP pleinement paramétrés à notre disposition et

de l'architecture de la cible visée, un autre critère qui puisse permettre de sélectionner un *meilleur* oSoP ?

Ce nouveau critère de choix discuté ici est le *parallélisme*. En effet, si l'architecture de la cible le permet (plusieurs multiplieurs, plusieurs additionneurs), il est possible de paralléliser les calculs, c'est-à-dire d'effectuer plusieurs calculs en même temps. Si les multiplications sont indépendantes les unes des autres et donc toutes parallélisables, la parallélisation des additions quant à elle dépend du choix de l'oSoP. En effet, considérons par exemple les deux oSoP de la figure 4.12. Celui de gauche est une accumulation séquentielle, chaque terme s'ajoute au résultat précédent l'un après l'autre, il n'y a pas possibilité de calculer deux additions en même temps. À l'opposé, dans l'oSoP de droite, les additions $\boldsymbol{p}_1+\boldsymbol{p}_2$ et $\boldsymbol{p}_3+\boldsymbol{p}_4$ peuvent être exécutées en même temps, la dernière somme nécessitant le résultat des deux premières additions.

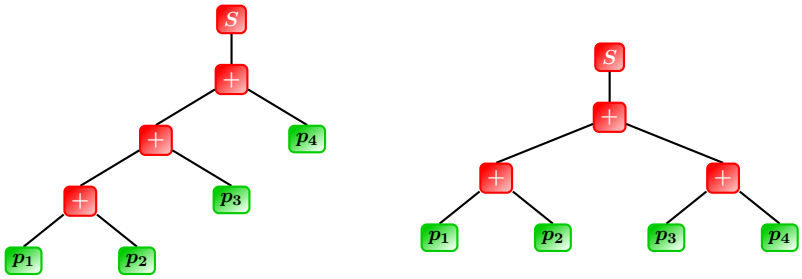


Figure 4.12 – Deux oSoP de même ordre illustrant la notion de parallélisme.

Deux additions d'un même oSoP peuvent s'effectuer en parallèle si elles sont indépendantes l'une de l'autre. En considérant la représentation sous forme d'arbre binaire entier, aucun des deux nœuds additions ne doit être parent ou descendant de l'autre (les deux nœuds doivent se situer sur des branches différentes de l'arbre). On peut en déduire qu'un arbre sera le plus parallélisable possible quand il possédera le plus grand nombre de branches différentes, c'est-à-dire quand sa hauteur sera minimale.

Proposition 4.4. *La hauteur minimale d'un arbre binaire entier à n feuilles, notée h, est donnée par :*

$$h \triangleq \lceil \log_2(n) \rceil \tag{4.23}$$

Pour la hauteur de nos oSoP, on ne considère pas la distance de 1 entre l'addition finale et le résultat S de la somme, cette arête finale ne servant qu'à rendre visuelle la sortie de l'addition. De même, les noeuds de formatages indiqués par un F ne sont pas pris en compte, seules les additions et les multiplications sont prises en compte pour le calcul de la hauteur.

Exemple 4.4. *Reprenons l'exemple fil rouge (voir 4.2), l'un des oSoP qui minimise la hauteur, et donc maximise les possibilités de parallélisation, est illustré par la figure 4.13.*

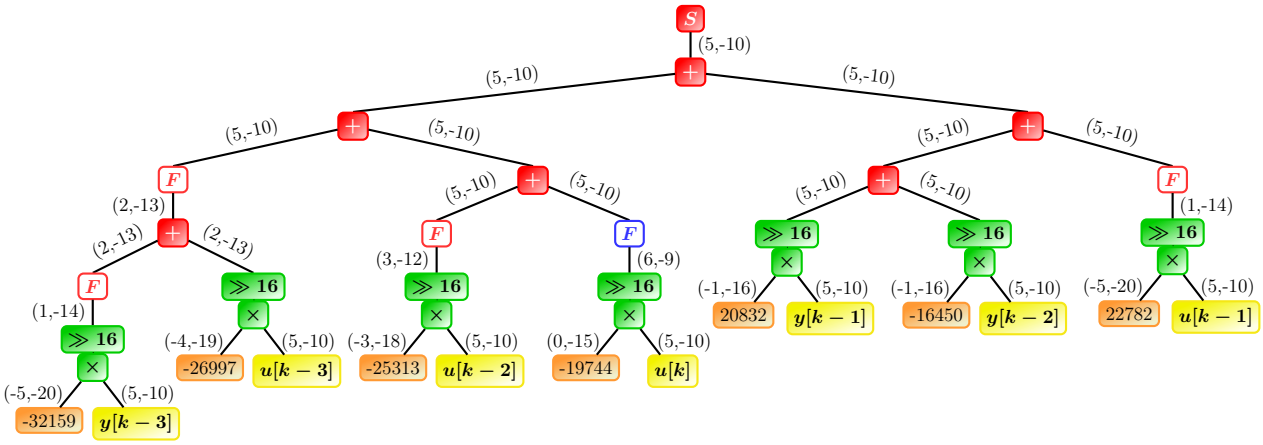


FIGURE 4.13 – oSoP de hauteur minimale.

4.3.4 Choix d'un oSoP

On l'a vu, le principal critère de choix d'un oSoP parmi tous ceux générés à partir d'un SoP est l'erreur numérique globale (modélisée par une accumulation de bruits ou d'intervalles), mais on peut également considérer le parallélisme d'un oSoP. Ces différents critères pourrait faire l'objet d'un problème d'optimisation multi-critère, ce qui n'a pas été fait ici. En lieu et place de cette optimisation multi-critère, le meilleur oSoP est choisi ici soit comme celui qui minimise la hauteur parmi tous ceux qui ont la même erreur globale, soit comme celui qui minimise l'erreur globale parmi ceux qui ont la hauteur minimale donnée par la proposition 4.4. Il n'y a par ailleurs pas unicité de cet oSoP, par exemple, parmi ceux qui minimise l'erreur globale, il peut y en avoir plusieurs qui minimisent la hauteur. Dans ce cas, on peut choisir n'importe lequel de ces oSoP.

L'oSoP de la figure 4.11 est l'un des oSoP qui, parmi ceux qui minimisent l'erreur globale, minimisent aussi la hauteur.

4.4 Génération de code virgule fixe pour un oSoP

Une fois que le meilleur oSoP (selon les critères discutés précédemment) a été déterminé, on veut en déduire l'algorithme virgule fixe qu'il représente, c'est-à-dire la description de chaque opération, chaque formatage, d'après la méthode analytique utilisée pour propager les formats. En effet, l'oSoP est une description visuelle de l'algorithme dont on va pouvoir déduire le-dit algorithme et le code associé.

Les langages qui nous intéressent ici sont le langage `C` [53], généralement utilisé (avec l'*Assembleur*) pour la programmation sur DSP (*Digital Signal*

Processor), le `C++` pour la simulation virgule fixe, et le langage de description matériel `VHDL` [5] à destination des FPGA (*Field-Programmable Gate Array*) et ASIC (*Application-Specific Integrated Circuit*). La conception de code `VHDL` n'a pas été abordée pendant la thèse mais sera néanmoins brièvement discuté.

4.4.1 Algorithme virgule fixe

La première étape dans la génération de code est la description de l'algorithme, permettant ensuite d'effectuer une simple traduction de celui-ci dans le langage souhaité. Cette description de l'algorithme pour un oSoP donné se fait en parcourant récursivement l'arbre binaire de la racine vers les feuilles (en considérant les multiplications et leurs couples constantes/variables comme des feuilles étendues) en passant par les nœuds internes (les additions), chaque opération décrivant le calcul qu'elle effectue.

Algorithme de Sethi-Ullman : Cet algorithme [95] est utilisé, au moins à titre indicatif, pour déterminer le nombre minimal de registres nécessaire pour notre algorithme lors de l'implémentation. Cet algorithme annote les noeuds d'un arbre binaire entier pour déterminer le nombre minimal de registres mais également pour donner l'ordre de parcours qui atteint ce nombre minimal.

L'algorithme original de Sethi-Ullman est donné par l'algorithme 4. Il est appliqué à la racine de l'oSoP et utilise des appels récursifs sur les nœuds fils de l'arbre binaire pour parcourir ce dernier.

Cependant, étant donné que les feuilles de notre arbre binaire ont toutes la même forme (multiplication constante/variable), on peut simplifier le parcours de l'arbre en annotant d'un 1 les nœuds correspondants à une multiplication, sans avoir à annoter les constantes et les variables. Appliqué à l'oSoP de la figure 4.10, l'algorithme de Sethi-Ullman et la simplification donne l'annotation de l'arbre de la figure 4.14, en considérant les multiplications comme des feuilles (voir figure 4.6) avec le changement de variables suivant :

$$\boldsymbol{p}_i = \begin{cases} \boldsymbol{b}_{i-1} u(k-i) & \text{pour} \quad 1 \leqslant i \leqslant 4 \\ \boldsymbol{a}_{i-4} y(k-i+4) & \text{pour} \quad 5 \leqslant i \leqslant 7 \end{cases}. \tag{4.24}$$

Dans cet exemple on voit qu'il faut au minimum trois registres pour effectuer le calcul du produit-scalaire.

Parcours de l'oSoP et génération de l'algorithme : Une fois l'algorithme de Sethi-Ullman appliqué, on parcours l'oSoP de la racine vers les feuilles pour "lire" les opérations, en suivant l'annotation des nœuds selon les règles suivantes, pour un nœud donné :

- si les deux nœuds fils du nœud ont le même label, l'ordre de parcours n'a pas d'importance,

Algorithme 4: Algorithme de Sethi-Ullman

Données: R : nœud racine d'un arbre binaire entier
Résultat: R annoté par le label de Sethi-Ullman

$label_X$ désigne le label de Sethi-Ullman du nœud X

Si *R est un nœud interne* **Alors**
  # Appel récursif sur les nœuds fils
  Pour *chacun des deux nœuds fils de R* **Faire**
    Si *le nœud n'a pas de label de S-U* **Alors**
      Appliquer l'algorithme sur ce nœud ;
    Fin
  Fait
  # On note N_1 et N_2 les deux nœuds fils de R
  Si $label_{N_1} = label_{N_2}$ **Alors**
    $label_R \leftarrow label_{N_1} + 1$;
  Sinon
    $label_R \leftarrow \max(label_{N_1}, label_{N_2})$;
  Fin
Sinon
  Si *R correspond à un terme constant* **Alors**
    $label_R \leftarrow 0$;
  Sinon
    $label_R \leftarrow 1$;
  Fin
Fin

– si les deux nœuds fils du nœud ont un label différent, on parcourt d'abord le sous-arbre ayant pour racine le nœud fils ayant le plus grand label.

L'algorithme déduit de l'oSoP est un algorithme entier, c'est-à-dire que toutes les constantes et les variables, ainsi que les résultats intermédiaires, sont des entiers, et les alignements de formats virgule fixe sont exprimés par des décalages (puisqu'on se trouve dans le cas d'une implémentation logicielle).

Chaque multiplication est stockée dans un registre et chaque addition de deux valeurs stockées dans des registres est stockée dans le registre de la première valeur ($R_x \leftarrow R_x + R_y$, où R_x et R_y sont deux registres).

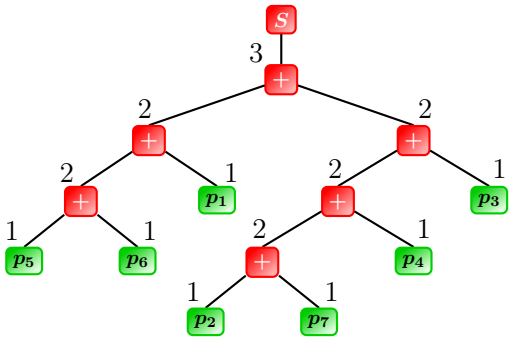


Figure 4.14 – oSoP annoté par l'algorithme de Sethi-Ullman.

À titre d'exemple, l'algorithme 5 est l'algorithme virgule fixe déduit de l'oSoP de la figure 4.14.

Données : les mantisses associées aux entrées $u[k-i]$ pour $i = 0, \ldots, 3$ et les sorties $y[k-i]$ pour $i = 1, \ldots, 3$
Résultat : $y[k]$

D'après l'annotation de Sethi-Ullman, il faut trois registres 16 bits, notés R_0, R_1 et R_2

1 $R_0 \leftarrow (20832 \times y[k-1]) \gg 16$;
2 $R_1 \leftarrow (-16450 \times y[k-2]) \gg 16$;
3 $R_0 \leftarrow R_0 + R_1$;
4 $R_1 \leftarrow (-19744 \times u[k]) \gg 16$;
5 $R_1 \leftarrow R_1 \ll 1$;
6 $R_0 \leftarrow R_0 + R_1$;
7 $R_1 \leftarrow (22782 \times u[k-1]) \gg 16$;
8 $R_2 \leftarrow (-32159 \times y[k-3]) \gg 16$;
9 $R_1 \leftarrow R_1 + R_2$;
10 $R_1 \leftarrow R_1 \gg 1$;
11 $R_2 \leftarrow (-26997 \times u[k-3]) \gg 16$;
12 $R_1 \leftarrow R_1 + R_2$;
13 $R_1 \leftarrow R_1 \gg 1$;
14 $R_2 \leftarrow (-25313 \times u[k-2]) \gg 16$;
15 $R_1 \leftarrow R_1 + R_2$;
16 $R_1 \leftarrow R_1 \gg 2$;
17 $R_0 \leftarrow R_0 + R_1$;
18 $y[k] \leftarrow R_0$;

Algorithme 5: Algorithme virgule fixe obtenu à partir de la figure 4.14.

L'algorithme virgule fixe obtenu à partir d'un oSoP nous donne donc la description des calculs de cet oSoP, et cet algorithme va pouvoir être traduit dans un langage informatique pour générer du code.

4.4.2 Code C entier

Le premier code qu'on génère à partir de l'algorithme est un code en langage C ne faisant intervenir que des entiers et des décalages, ce qui correspond à une traduction directe de l'algorithme généré précédemment.

Le code est généré dans une fonction qui, pour des entrées $\boldsymbol{v}_i$ données pour $1 \leqslant i \leqslant n$, calcule le résultat du produit scalaire $s = \sum_{i=1}^{n} \boldsymbol{c}_i \times \boldsymbol{v}_i$. Appliquée à un produit scalaire de filtre, par exemple avec une forme directe I, en considérant le changement de variable donné par l'équation (4.24), la fonction calcule la sortie y du filtre à un instant k.

Le code 4.1 présente le code généré pour l'algorithme 5, c.-à-d. celui décrivant le calcul de l'oSoP de la figure 4.14 (ou de la figure 4.11).

Code 4.1 – Code C entier correspondant à la figure 4.11.

```
int16_t Code_C_int(int16_t v1,int16_t v2,int16_t v3,
        int16_t v4,int16_t v5,int16_t v6, int16_t v7)
{
    // Registers declaration
        int16_t r0, r1, r2;
        // Computation of c5*v5 in r0
        r0 = 20832 * v5 >> 16;
        // Computation of c6*v6 in r1
        r1 = -16450 * v6 >> 16;
        // Computation of r0+r1 in r0
        r0 = r0 + r1;
        // Computation of c1*v1 in r1
        r1 = -19744 * v1 >> 16;
        r1 = r1 << 1;
        // Computation of r0+r1 in r0
        r0 = r0 + r1;
        // Computation of c2*v2 in r1
        r1 = 22782 * v2 >> 16;
        // Computation of c7*v7 in r2
        r2 = -32159 * v7 >> 16;
        // Computation of r1+r2 in r1
        r1 = r1 + r2;
        r1 = r1 >> 1;
        // Computation of c4*v4 in r2
        r2 = -26997 * v4 >> 16;
        // Computation of r1+r2 in r1
        r1 = r1 + r2;
        r1 = r1 >> 1;
        // Computation of c3*v3 in r2
        r2 = -25313 * v3 >> 16;
        // Computation of r1+r2 in r1
        r1 = r1 + r2;
        r1 = r1 >> 2;
        // Computation of r0+r1 in r0
        r0 = r0 + r1;
        // The result is returned
        return r0;
}
```

4.4.3 Code C++ virgule fixe

Il est également possible de générer du code C++, destiné à la simulation, utilisant des librairies virgule fixe telles que la librairie ac_fixed (ac pour Algorithmic C) de Mentor Graphics [3], désormais distribuée par Calypto [4] (Calypto fourni également Catapult C pour faire de la synthèse matérielle à partir

3. http ://www.mentor.com/
4. http ://calypto.com/en/products/catapult/catapult_overview

de code C++). Cette librairie permet la déclaration de variables virgule fixe de largeur arbitraire en précisant leurs formats, et d'autres options. Le template d'une déclaration `ac_fixed` s'écrit comme suit :

```
ac_fixed<int W, int I, bool S, ac_q_mode Q, ac_o_mode O>
```

avec les paramètres suivants :

- `W` : un entier correspondant à la largeur de la variable définie (correspondant à notre notation w),
- `I` : un entier précisant le nombre de bits pour la partie entière ($m + 1$ dans notre notation, où m est le msb de la variable),
- `S` : un booléen pour le signe de la variable (`true` si la variable est signée, `false` si elle ne l'est pas),
- `Q` (optionnel) : le mode d'arrondi de la variable, par troncature ou par arrondi au plus proche. Pour chaque mode il existe plusieurs versions selon la direction de l'arrondi. Le mode d'arrondi par défaut est l'arrondi par troncature dirigé vers $-\infty$, nommé `AC_TRN` dans la librairie,
- `O` (optionnel) : le mode de débordement, par arithmétique modulaire (suppression des bits qui débordent) ou par saturation (vers la plus grande valeur représentable la plus proche du nombre ou vers 0). Par défaut, le mode de débordement est celui de l'arithmétique modulaire, nommé `AC_WRAP` dans la librairie.

Remarque 4.3. *La librairie* ***`ac_fixed`*** *est fournie avec un ensemble de librairies labellisées Algorithmic C, dont la librairie* ***`ac_int`*** *qui permet d'écrire du code entier destiné à la simulation avec des entiers de tailles arbitraires. On peut alors imaginer générer un code où les entiers sont codés sur 20 bits ou un code où chaque entier à sa propre largeur.*

Ce code n'utilise donc pas les entiers comme le code `C` vu précédemment mais les valeurs décimales (les mantisses multipliées par leurs facteurs d'échelle), comme le montre l'exemple de déclaration suivant.

Exemple 4.5. *Soit le nombre virgule fixe $x = 6.1875$, que l'on considère signé, avec les modes d'arrondis et de débordements par défaut. La ligne de code utilisée pour définir la variable x (avec le nombre minimal de bits) dans le code est :*

```
ac_fixed<8, 4, true> x = 6.1875;
```

En effet, l'écriture binaire, en complément à deux signé, de $x = 6.1875$ est $x =_2 \ldots 000110.001100\ldots$, donc le nombre minimale de bits pour le représenter sans perte d'information est 8 (en comptant le bit de signe), et il y a alors 4 bits de partie entière.

Pour un produit scalaire de filtre donné le résultat a toujours le même format (le résultat est connu pour être dans un intervalle, d'après la proposition

3.2), il est donc possible de définir une fonction renvoyant un type `ac_fixed` de ce format, et calculant le produit scalaire pour des entrées également en `ac_fixed` dans les bons formats.

Ici, il est difficile de respecter les annotations de l'algorithme de Sethi-Ullman puisque chaque registre doit être défini en un type `ac_fixed` dans un format précis et fixé (en `C++` il n'est pas possible de changer le type d'une variable déjà définie). Les registres sont si possibles réutilisés quand ils sont libres (c.-à-d. s'ils ont servi auparavant à stocker un résultat intermédiaire) et au bon format.

À l'opposé du code `C` entier vu précédemment, il n'y pas besoin, avec la librairie `ac_fixed`, de gérer les arrondis et les overflow, ils font partie du typage des variables. Ainsi, on peut ajouter deux termes virgule fixe de formats différents sans alignement manuel au préalable, le format du résultat étant le format de la variable dans laquelle est stocké ce résultat.

Le code 4.2 montre le code généré pour l'oSoP de la figure 4.11.

Code 4.2 – Code `C++` virgule fixe correspondant à la figure 4.11.

```
ac_fixed<16,6,true> Code_C_fixed(ac_fixed<16,6,true> v1,
        ac_fixed<16,6,true> v2, ac_fixed<16,6,true> v3,
        ac_fixed<16,6,true> v4, ac_fixed<16,6,true> v5,
        ac_fixed<16,6,true> v6, ac_fixed<16,6,true> v7)
{
        //Computation of c5*v5 in r0
        ac_fixed<16,0,true> c5 = 0.31787109375;
        ac_fixed<16,6,true> r0 = c5*v5;
        //Computation of c6*v6 in r1
        ac_fixed<16,0,true> c6 = -0.251007080078125;
        ac_fixed<16,6,true> r1 = c6*v6;
        //Computation of r0+r1 in r0
        r0 = r0 + r1;
        //Computation of c1*v1 in r2
        ac_fixed<16,1,true> c1 = -0.6025390625;
        ac_fixed<16,7,true> r2 = c1*v1;
        //Computation of r0+r2 in r0
        r0 = r0 + r2;
        //Computation of c2*v2 in r3
        ac_fixed<16,-4,true> c2 = 0.0217266082763671875;
        ac_fixed<16,2,true> r3 = c2*v2;
        //Computation of c7*v7 in r4
        ac_fixed<16,-4,true> c7 = -0.03066921234130859375;
        ac_fixed<16,2,true> r4 = c7*v7;
        //Computation of r3+r4 in r3
        r3 = r3 + r4;
        //Computation of c4*v4 in r5
        ac_fixed<16,-3,true> c4 = -0.0514926910400390625;
        ac_fixed<16,3,true> r5 = c4*v4;
        //Computation of r4+r5 in r5
        r5 = r4 + r5;
        //Computation of c3*v3 in r6
        ac_fixed<16,-2,true> c3 = -0.096561431884765625;
```

```
        ac_fixed<16,4,true> r6 = c3*v3;
        //Computation of r5+r6 in r6
        r6 = r5 + r6;
        //Computation of r0+r6 in r0
        r0 = r0 + r6;
        // The result is returned
        return r0;
}
```

4.4.4 Code VHDL

Comme il a été dit en introduction de cette partie, la génération de code VHDL pour un produit scalaire n'a pas été abordée durant cette thèse. Cependant, il existe plusieurs outils pouvant générer du code VHDL optimisé à partir d'expressions paramétrés tels que nos oSoP paramétrés, par exemple l'outil FloPoCo[5] [24].

À ce moment de la rédaction, des travaux sont en cours pour pouvoir faire appel à FloPoCo pour générer le code VHDL à partir de nos oSoP, mais ces travaux ne sont pas suffisamment avancés pour donner un exemple de code VHDL correspondant à un de nos oSoP.

4.5 Résumé de la méthode et illustration sur un filtre réalisé par un state-space

On a vu dans ce chapitre comment, à partir d'un produit scalaire de filtre et de ses spécificités, générer le meilleur code virgule fixe (meilleur selon certains critères), soit implicite (des entiers dont on gère l'alignement) soit explicite (avec la librairie ac_fixed), qui implémente ce produit scalaire. Afin de résumer la méthode, nous rappelons chacune des étapes nécessaires, pour passer d'un produit scalaire à du code :

1. Convertir les constantes en virgule fixe sur la largeur donnée (on rappelle qu'on est dans un contexte d'implémentation logicielle, les largeurs sont donc fixées), et déterminer les intervalles virgule fixe (intervalles plus formats) des variables d'entrées (données) et des variables intermédiaires ou de sorties (en utilisant la proposition 3.2) ;
2. générer un à un les différents oSoP possibles (on préférera utiliser la génération par utilisation d'une pile) ;
3. propager les données de départ (constantes et intervalles virgule fixe des variables) à travers chaque oSoP généré en utilisant l'arithmétique d'intervalle ;
4. choisir, en fonction des critères définis (dégradation numérique, parallélisme), le meilleur oSoP parmi ceux générés ;

5. http://flopoco.gforge.inria.fr/

5. pour ce meilleur oSoP, générer l'algorithme et/ou le code dans le langage désiré.

Cette méthode ne s'applique qu'à un produit scalaire de filtre, or, comme on l'a vu au début de ce chapitre, un filtre peut s'écrire comme une succession de produits scalaires décrivant l'algorithme complet de ce filtre. Pour prendre en compte plusieurs produits scalaires, il suffit d'appliquer la méthode précédente à chaque produit scalaire de l'algorithme, comme le montre l'exemple de la partie suivante.

4.5.1 Exemple de traitement d'un filtre réalisé par plusieurs produits scalaires

Cette partie présente un exemple d'application de la méthode vue tout au long de ce chapitre sur une réalisation de filtre composée de plusieurs produits scalaires. Le filtre considéré est le filtre fil rouge défini dans la partie 3.4, page 91, avec la réalisation state-space donnée par l'équation (3.44). La largeur des coefficients, de toutes les variables est fixée est 16 bits et celle des opérateurs (multiplieurs et additionneurs), est fixée à 32 bits. Le mode d'arrondi considéré est l'arrondi par troncature. L'algorithme virgule fixe décrit pas le state-space est, une fois tous les calculs développés, le suivant :

$$\begin{aligned}
\boldsymbol{x}_1(k+1) &\leftarrow \widetilde{\boldsymbol{A}}_{11}\boldsymbol{x}_1(k) + \widetilde{\boldsymbol{A}}_{12}\boldsymbol{x}_2(k) + \widetilde{\boldsymbol{A}}_{13}\boldsymbol{x}_3(k) + \widetilde{\boldsymbol{b}}_1 u(k) \\
\boldsymbol{x}_2(k+1) &\leftarrow \widetilde{\boldsymbol{A}}_{21}\boldsymbol{x}_1(k) + \widetilde{\boldsymbol{A}}_{22}\boldsymbol{x}_2(k) + \widetilde{\boldsymbol{A}}_{23}\boldsymbol{x}_3(k) + \widetilde{\boldsymbol{b}}_2 u(k) \\
\boldsymbol{x}_3(k+1) &\leftarrow \widetilde{\boldsymbol{A}}_{31}\boldsymbol{x}_1(k) + \widetilde{\boldsymbol{A}}_{32}\boldsymbol{x}_2(k) + \widetilde{\boldsymbol{A}}_{33}\boldsymbol{x}_3(k) + \widetilde{\boldsymbol{b}}_3 u(k) \\
y(k) &\leftarrow \widetilde{\boldsymbol{c}}_1\boldsymbol{x}_1(k) + \widetilde{\boldsymbol{c}}_2\boldsymbol{x}_2(k) + \widetilde{\boldsymbol{c}}_3\boldsymbol{x}_3(k) + \widetilde{d}u(k)
\end{aligned}$$

avec

$$\begin{aligned}
\widetilde{\boldsymbol{A}} &\triangleq \begin{pmatrix} 17.0458984375 & 39.744140625 & 28.3779296875 \\ -298.265625 & -695.78125 & -496.796875 \\ 407.546875 & 951. & 679.03125 \end{pmatrix} \\
\widetilde{\boldsymbol{c}} &\triangleq \begin{pmatrix} 58.7109375 & 67.44921875 & 46.9296875 \end{pmatrix} \\
\widetilde{\boldsymbol{b}} &\triangleq \begin{pmatrix} -0.003797054290771484375 \\ -0.005493640899658203125 \\ 0.009027957916259765625 \end{pmatrix}, \quad \widetilde{d} \triangleq -0.6025390625
\end{aligned}$$

Il y a donc quatre produits scalaires à considérer. Le signal d'entrée u prend ses valeurs dans l'intervalle $[-25; 30]$ (c.-à-d. $u(k) \in [-25; 30]$ à tout instant k), donc, par la proposition 3.2 on a :

$$\begin{aligned}
\boldsymbol{x}_1 &\in [-1.47308; 1.39374], \\
\boldsymbol{x}_2 &\in [-22.9346; 23.9521], \\
\boldsymbol{x}_3 &\in [-32.7637; 31.3848], \\
\boldsymbol{y} &\in [-23.7637; 19.9824],
\end{aligned}$$

dont on peut déduire les formats virgule fixe pour les variables :

$$\begin{aligned}(m_{\boldsymbol{x}_1}, \ell_{\boldsymbol{x}_1}) &= (1, -14),\\ (m_{\boldsymbol{x}_2}, \ell_{\boldsymbol{x}_2}) &= (5, -10),\\ (m_{\boldsymbol{x}_3}, \ell_{\boldsymbol{x}_3}) &= (6, -9),\\ (m_{\boldsymbol{y}}, \ell_{\boldsymbol{y}}) &= (6, -9).\end{aligned}$$

On exécute alors notre méthode sur chacun des produits scalaires. Après, génération des différents oSoP possibles (4 fois 15 oSoP), propagation des informations par arithmétique d'intervalle et choix du meilleur oSoP avec comme critère l'oSoP ayant la plus petite hauteur parmi ceux qui ont la plus petite dégradation numérique, les quatre meilleurs oSoP sont présentés par la figure 4.15.

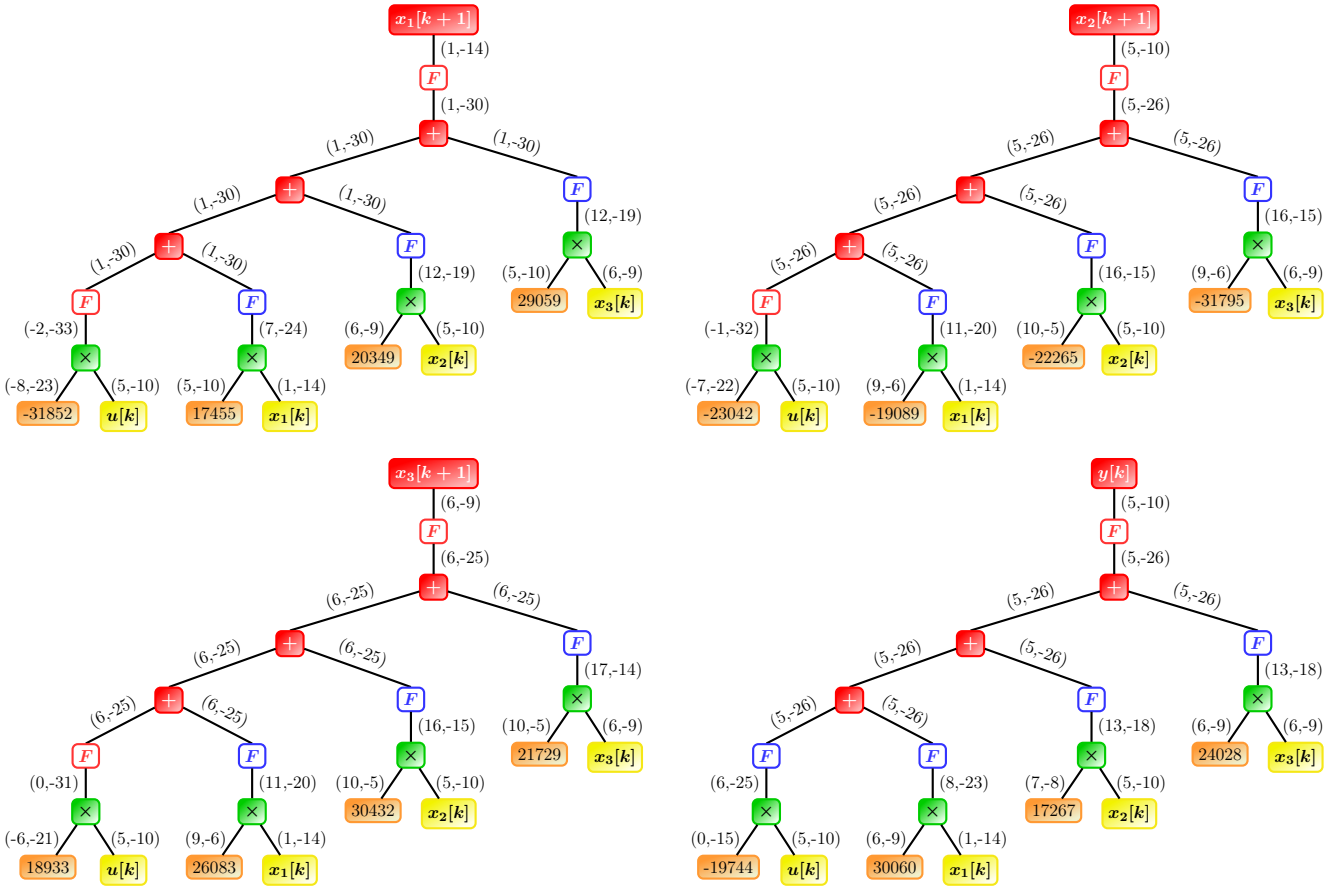


Figure 4.15 – Les 4 meilleurs oSoP pour l'exemple considéré avec calculs sur 32 bits.

On peut déjà constater que sur chaque oSoP, la règle de Jackson pour la

virgule fixe supprime une quantité non négligeable de bits. L'application de la règle permet de conserver plus de bits du produits $\boldsymbol{b}_1 \times u(k)$: sans elle, pour aligner $\boldsymbol{b}_1 \times u(k)$ avec $\boldsymbol{A}_{11} \times \boldsymbol{x}_1(k)$, il aurait fallu supprimer 9 bits du premier produit au lieu de 3 en applicant la règle.

Les intervalles d'erreurs pour les quatre oSoP sont calculés d'après les équations (4.15) et (4.16), on a alors :

$$\begin{aligned}
[\underline{e}_{x_1}; \overline{e}_{x_1}] &= [-6.10351\mathtt{e}\text{-}05; 0] \\
[\underline{e}_{x_2}; \overline{e}_{x_2}] &= [-0.000976563; 0] \\
[\underline{e}_{x_3}; \overline{e}_{x_3}] &= [-0.00195313; 0] \\
[\underline{e}_{y}; \overline{e}_{y}] &= [-0.000976548; 0]
\end{aligned}$$

où $[\underline{e}_X; \overline{e}_X]$ désigne l'intervalle d'erreur sur la variable X avec $X \in \{x_1, x_2, x_3, y\}$.

On peut alors appliquer le corollaire 3.1 (et les équations (3.9) et (3.10) qui donnent les formules pour la sortie sous forme d'intervalle bornes inf/sup plutôt que centre/rayon) sur le filtre $\mathcal{H}_{\boldsymbol{\varepsilon}}$ donné par l'équation (3.48) et le vecteur d'erreur $\boldsymbol{\varepsilon}(k)$ donné par les intervalles précédents, c'est-à-dire :

$$\boldsymbol{\varepsilon}(k) \in \begin{pmatrix} [\underline{e}_{x_1}; \overline{e}_{x_1}] \\ [\underline{e}_{x_2}; \overline{e}_{x_2}] \\ [\underline{e}_{x_3}; \overline{e}_{x_3}] \\ [\underline{e}_{y}; \overline{e}_{y}] \end{pmatrix}. \tag{4.25}$$

Le corollaire 3.1 donne alors les bornes de l'intervalle de l'erreur $\delta y(k) \triangleq y^*(k) - y(k)$, où $y^*(k)$ est le résultat implémenté en virgule fixe, et $y(k)$ est la sortie exacte. L'intervalle de l'erreur théorique (dont les bornes sont représentées en rouge sur la figure) est donné par le corollaire 3.1, il vaut :

$$[\underline{\delta y}; \overline{\delta y}] = [-0.262932; 0.0356409].$$

Pour valider la théorie, on génère le code `C` entier de ces oSoP (voir les codes B.1, B.2, B.3 et B.4 en annexes), et on compare ces codes avec un code flottant double précision (utilisant des nombres flottants 64 bits). Puisque le filtre $\mathcal{H}_{\boldsymbol{\varepsilon}}$ ne considère que le comportement des erreurs de calculs à travers lors de l'implémentation (et pas les erreurs paramétriques, c'est-à-dire les erreurs de conversion des coefficients), l'implémentation double précision de référence utilise elle aussi les coefficients sur 16 bits, les calculs intermédiaires et les variables sont sur 64 bits.

Le signal d'entrée u pour cette comparaison est tiré aléatoirement (pseudo-alétoirement en réalité, en utilisant la fonction `rand()` du langage `C`) dans l'intervalle $[-25; 30]$. Ce tirage est effectué en double précision et converti en entier 16 bits pour le code entier généré. La figure 4.17 montre la courbe de la différence $\delta y(k) = y^*(k) - y(k)$ entre la sortie de notre code $y^*(k)$ et la sortie de référence $y(k)$, calculée en double précision, pour $0 \leqslant k \leqslant 10000$. Les deux droites rouges représentent quant à elle les bornes de l'intervalle théorique de

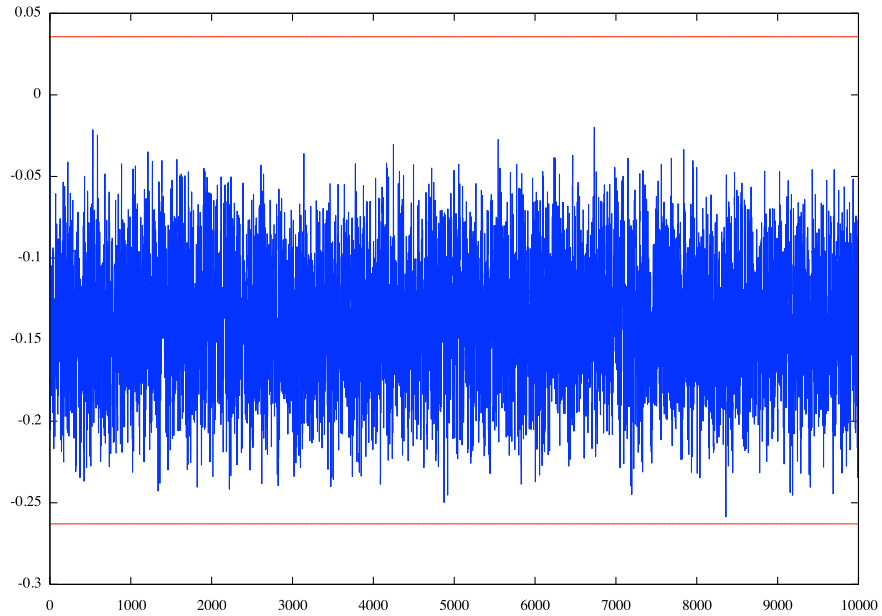


Figure 4.16 – Différence entre la sortie y flottante double précision et la sortie 32 bits du code généré.

l'erreur δy, $[-0.262932; 0.0356409]$. On constate alors que l'erreur observée est bien dans l'intervalle théorique calculé et s'en rapproche.

Remarque 4.4. *Nous avons vu dans la partie 3.2.2 une autre approche pour la majoration théorique de l'erreur, considérant l'ensemble des erreurs (erreurs de calcul et erreurs paramétriques), que l'on va appliquer à cet exemple. Pour cela, on considère comme code de référence un code double avec des paramètres double précision également. L'erreur mesurée par la comparaison du code de référence et du code virgule fixe généré prend donc en compte les erreurs de paramétrisation et de calcul. Les bornes théoriques de cette erreur sont données par le corollaire 3.1, appliqué au filtre de l'erreur globale $\widetilde{\mathcal{H}}_{\boldsymbol{\varepsilon}}$ et à l'entrée* $\begin{pmatrix} [\underline{u}; \overline{u}] \\ [\underline{e}_{x_1}; \overline{e}_{x_1}] \\ [\underline{e}_{x_2}; \overline{e}_{x_2}] \\ [\underline{e}_{x_3}; \overline{e}_{x_3}] \\ [\underline{e}_y; \overline{e}_y] \end{pmatrix}$*, comme expliqué dans la partie 3.2.2.*

La figure 4.17 présente la différence $\Delta y(k) = \widetilde{y}(k) - y(k)$ (on rappelle que $\widetilde{y}(k)$ est la sortie bruitée des erreurs de calcul et de paramétrisation à l'instant k), pour $0 \leqslant k \leqslant 10000$. L'intervalle théorique de l'erreur est $[\underline{\Delta y}; \overline{\Delta y}] = [-0.803801; 0.491936]$.

Le principe majeur derrière cet exemple est que le nombre de bits influe directement sur l'exactitude du résultat du calcul. Ce principe va de soi, mais

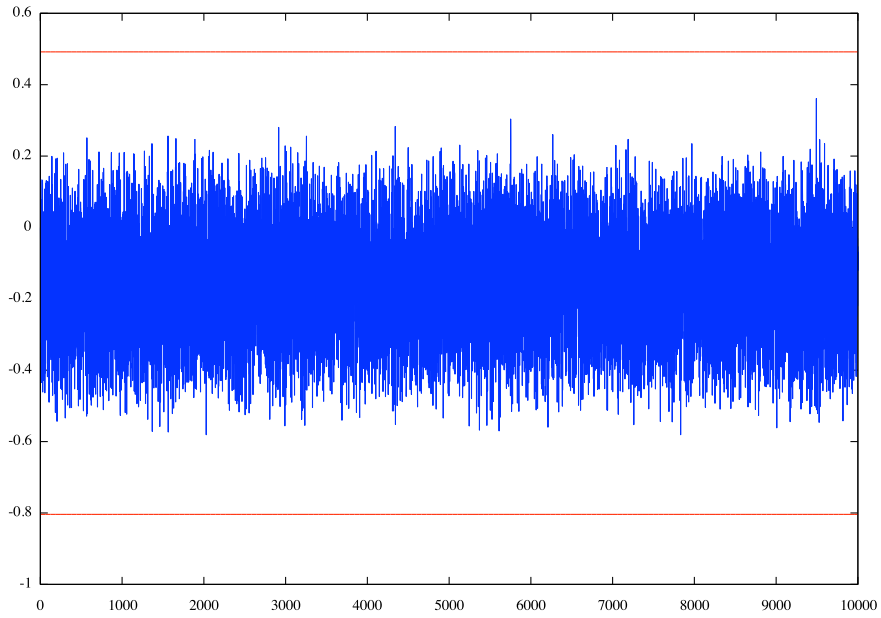


FIGURE 4.17 – Différence entre la sortie y flottante double précision et la sortie 32 bits du code généré en considérant l'ensemble des erreurs.

permet de se poser la question du choix du meilleur nombre de bits pour un intervalle d'erreur donné. Si dans le domaine du logiciel toutes les variables et coefficients ont la même largeur (généralement 8, 16 ou 32), en matériel chaque coefficient, chaque variable, peut avoir sa propre largeur. Par conséquent, le problème de trouver le meilleur ensemble de largeurs pour correspondre à un intervalle d'erreur donné devient un problème d'optimisation combinatoire, ce qui est l'objet du prochain chapitre.

4.6 Conclusion

Nous avons proposé, dans ce chapitre, une méthode pour implémenter des algorithmes de filtres linéaires en virgule fixe à destination de cibles logicielles. Cette méthode se déroule en plusieurs étapes : la génération des différents schémas d'évaluations possibles, la propagation des intervalles et l'évaluation de l'erreur globale sur chaque schéma, et le choix du meilleur d'entre eux.

La méthodologie d'implémentation que nous proposons utilise les briques de bases décrites dans le chapitre précédent, pour déterminer le format virgule fixe final et ainsi appliquer la règle d'arithmétique modulaire autorisant les débordements intermédiaires. Cette méthode analytique permet de limiter l'expansion inutile du format virgule fixe final (due à l'utilisation de l'arithmétique d'intervalle) et donc de limiter l'effet d'absorption des bits de poids

faibles.

Nous avons également décrit dans ce chapitre comment générer du code virgule fixe à partir du schéma virgule fixe choisit d'après nos critères, et présenté une application de cette méthodologie sur l'exemple fil rouge.

Chapitre 5

Optimisation des largeurs

Le précédent chapitre traitait du cas d'une implémentation logicielle où les opérateurs, les constantes et les variables ont une largeur fixée par la cible. Dans le cas d'une implémentation matérielle (sur FPGA ou ASIC), les opérateurs sont décrits par un langage de description matériel (comme le `VHDL`) et c'est l'utilisateur qui décide des largeurs (en entrée et en sortie) des opérateurs, mais aussi des largeurs des constantes et des variables. Toutes les largeurs peuvent être différentes, ce principe est souvent nommé dans la littérature le *paradigme des largeurs multiples* [20]. Il est montré dans cet article que pour chaque choix d'ensemble de largeurs pour les différentes données , il est possible d'évaluer l'impact numérique de ce choix. Or, lorsque l'on est face à un choix, la question naturelle à se poser est comment faire le bon choix, et selon quels critères. De l'intérêt d'une implémentation à largeurs multiples est né le problème d'*optimisation des largeurs*, un problème d'optimisation dans lequel on cherche à minimiser les largeurs tout en respectant des contraintes, ici des contraintes numériques.

L'optimisation consiste, dans un ensemble de solutions réalisables, à trouver la ou les meilleures solutions par rapport à un objectif donné. Une solution réalisable est un ensemble de valeurs qui répond à des contraintes. L'objectif s'exprime généralement par la recherche du minimum ou du maximum d'une fonction appelée *fonction objectif*.

Dans notre contexte, les solutions réalisables sont des ensembles de nombres entiers (des largeurs exprimées en nombre de bits) et l'ensemble des solutions réalisables est fini, on parle alors d'*optimisation combinatoire*.

Ce chapitre présentera un état de l'art de l'optimisation des largeurs avant de présenter une méthode qui réduit, sans problème d'optimisation, les bits à considérer dans un produit scalaire. Ensuite, nous formaliserons le problème d'optimisation adapté à notre approche et utilisant la méthode de réduction du nombre de bits. Enfin, nous montrerons comment ce problème d'optimisation peut être résolu avec les outils existants, avant de présenter un exemple

d'application de ce problème.

5.1 État de l'art autour de l'optimisation des largeurs en virgule fixe

Cette section présente dans un premier temps un état de l'art de l'optimisation combinatoire et des méthodes et algorithmes les plus connus. Dans un second temps, un historique des travaux réalisés dans le contexte de l'optimisation des largeurs est dressé.

5.1.1 Quelques notions d'optimisation combinatoire et principaux algorithmes

Il existe principalement deux types de méthodes de recherche de solutions dans un problème d'optimisation, les méthodes *exactes* et les méthodes *approchées*.

Les méthodes exactes utilisent généralement un arbre de recherche (ou arbre de décision) et le parcourent pour trouver la meilleure solution. Elles sont capables de trouver la solution *optimale*, c'est-à-dire celle qui répond le mieux au problème d'optimisation. L'algorithme par *Séparation et Évaluation* (BaBBaB pour le terme anglais *Branch and Bound*) [59] est le représentant le plus connu des méthodes exactes. L'inconvénient de ces méthodes est que le temps de calcul peut être très grand selon la taille du problème.

Les méthodes approchées, quant à elles, permettent de traiter plus rapidement des problèmes de grande taille mais ne garantissent pas l'optimalité, on parle alors de solutions *sous-optimales*. Les algorithmes gloutons ou encore les métaheuristiques [51] sont des exemples de méthodes approchées. Ces méthodes utilisent généralement des recherche locales ou des approches stochastiques pour trouver une solution.

Les algorithmes d'optimisation combinatoire peuvent être réunis en deux classes, les algorithmes *déterministes* et les algorithmes *randomisés* (ou *probabilistes*). Les premiers sont des algorithmes dans lesquels un état est directement déduit de l'état précédent, de la fonction objectif et des contraintes du problème. Les algorithmes randomisés, quand à eux, sont des algorithmes qui progressent aléatoirement parmi un ensemble d'états possibles à chaque étape.

On dresse maintenant une liste non-exhaustive des principaux algorithmes d'optimisation combinatoire et des méthodes de recherche d'une solution.

Recherche locale : Cette méthode est une métaheuristique qui cherche une nouvelle solution réalisable à partir d'une solution réalisable donnée en utilisant une notion de voisinage entre les solutions réalisables. Cette méthode ne garantie pas d'avoir un optimum globale, elle est donc classée dans les

méthodes approchées. Les algorithmes de recherches locales sont par exemple couramment utilisés pour le problème Minimum Vertex Cover [1] [12].

Recherche gloutonne [23] : Cette métaheuristique cherche à chaque étape l'optimum parmi les solutions réalisables proposées à cette étape. Combiné à la recherche locale, cette méthode choisit alors, à chaque étape la meilleure solution parmi les voisins de la solution courante. Elle permet en générale au mieux de trouver un optimum local.

Recherche tabou (ou recherche avec tabous) [30, 31] : Cette métaheuristique utilise aussi la notion de voisinage. En effet, à partir d'un état donné, le meilleur voisin (selon l'objectif souhaité) est choisi, et les voisins déjà visités et menant à une solution moins bonne sont stockés dans une pile appelée *liste tabous* afin de les retirer des voisins à explorer. L'algorithme se termine si plus aucun déplacement n'est possible ou si le nombre d'itérations maximal est atteint. Il est montré que dans un temps infini (pas de nombre d'itérations maximal), l'optimum global peut être atteint, mais dans la pratique il est nécessaire d'avoir une limite du nombre d'itérations, et donc la méthode est en général sous-optimale.

Séparation et Évaluation [59] : Cet algorithme énumère de façon implicite l'ensemble des solutions réalisables. L'étape de *séparation* divise le problème en plusieurs sous-problèmes (par exemple, dans la recherche d'un vecteur de n entiers, le premier entier est fixé à une valeur réalisable et on cherche le meilleur vecteur de $n-1$ entiers réalisable avec l'entier fixé, et ainsi de suite de façon récursive) et crée ainsi un arbre de recherche. L'étape d'*évaluation*, quant à elle, détermine, pour un nœud donné, l'optimum du sous-problème associé au nœud si cela est possible, ou au contraire l'inexistence d'un optimum dans le sous-problème associé. Cet algorithme est exacte et peut être associé à des métaheuristiques pour accélérer l'algorithme.

Programmation linéaire en nombres entiers [85] : La programmation linéaire en nombres entiers (PLNE) est un problème d'optimisation combinatoire dans lequel la fonction objectif et les fonctions de contraintes sont linéaires. L'idée est alors, si par exemple les contraintes ne sont pas linéaires, de les réécrire de façon linéaire sans modifier leur sens (les solutions qui sont réalisables pour les contraintes non linéaires doivent le rester pour les contraintes linéaires, et les contraintes linéaires ne doivent pas apporter de solutions réalisables supplémentaires). Pour résoudre la PLNE, on peut utiliser différentes méthodes : le BaB ou la Recherche tabou sont couramment utilisées.

1. Appelé en français Problème de Couverture Minimum par Sommets, ce problème de la théorie des graphes consiste à trouver un ensemble de sommets d'un graphe de taille minimale tel que chaque arête du graphe est incidente à un sommet de cet ensemble.

Recuit Simulé (RS) [16, 54] : Cette métaheuristique s'inspire d'un procédé métallurgique consistant à alterner cycles de refroidissement lent et de réchauffage (recuit) afin d'amener le matériau à un état minimisant son énergie interne. Cet algorithme part d'une solution réalisable initiale choisie aléatoirement, puis à chaque itération analyse l'énergie d'un voisin : si celle-ci est inférieure à l'énergie de l'état courant, alors le voisin est choisi comme nouvel état de référence, sinon le voisin est choisi selon une certaine probabilité (dépendant fortement de la température du système). L'algorithme s'arrête si le système n'évolue plus depuis un certain nombre d'itérations fixé ou si la température diminue jusqu'à atteindre un seuil fixé lui aussi. L'inconvénient de cet algorithme est principalement le choix des différents paramètres (choix de la solution initiale, de la température initiale, de la loi de décroissance de la température, des critères d'arrêts, etc). Plusieurs études montrent que le recuit simulé peut converger vers un optimum global sous certaines conditions et sans limite de temps [28], mais dans le cas général, la solution est sous-optimale.

Algorithmes génétiques [32, 47] : Ces algorithmes sont une sous-classes des algorithmes *évolutionnistes* (ou *évolutionnaires*) dont le principe se base sur l'évolution naturelle en biologie. En génétique, lors de la phase de reproduction entre deux organismes, on distingue trois opérations constituant le brassage génétique : la sélection, l'enjambement (ou croisement) et la mutation. La sélection détermine quel candidat, au sein d'une population, est le plus apte à donner de bons résultats (en sélection naturelle on parle du candidat le plus apte à survivre). L'enjambement produit de nouvelles solutions (de nouveaux candidats) à partir des deux organismes se reproduisant, en échangeant des gènes entre eux. Enfin, la mutation modifie, de façon aléatoire, un gène chez un organisme. Un algorithme génétique utilise ses trois opérations pour trouver une bonne solution : à partir d'une population initiale aléatoire, on sélectionne deux bons candidats, on effectue un enjambement de ces deux candidats puis on applique des mutations aléatoires pour créer une nouvelle population. L'algorithme s'arrête alors quand un maximum d'itérations est atteint ou quand la solution n'évolue plus ou plus assez rapidement (selon des critères définis par l'utilisateur).

Procédure de recherche gloutonne aléatoire et adaptative (GRASP) [27] : Cette métaheuristique combine une recherche gloutonne et une recherche locale. Une première phase, dite de *construction*, produit itérativement une liste de morceaux candidats à former une solution réalisable, puis la solution réalisable est construite à partir de cette liste en prenant aléatoirement des morceaux parmi les meilleurs candidats. Ensuite la phase de recherche locale part de cette solution créée et recherche dans le voisinage une meilleure solution réalisable. La solution résultante est alors conservée comme meilleure solution de l'itération de l'algorithme. Ce dernier s'arrête quand la condition d'arrêt

(généralement le nombre d'itérations) est atteinte, et la meilleure solution de toutes les itérations est retournée par l'algorithme.

5.1.2 Optimisations des largeurs

C'est au milieu des années 1990 que l'intérêt autour de l'implémentation à largeurs multiples et de l'optimisation des largeurs est apparu [97, 98, 105]. Il faut cependant attendre le début des années 2000 pour une formalisation du problème [20] et une étude de sa complexité [22]. En effet, dans [22], Constantinides et Woeginger expriment le problème de décision lié au problème d'optimisation des largeurs de la façon suivante :
"*Existe-t-il un ensemble de largeurs minimales qui minimise la fonction objectif tout en respectant les contraintes de précision*",
et démontrent que ce problème est NP-difficile [2]. En fait, l'expression exacte du problème de décision donnée dans leur article dépend de leur fonction objectif et de leurs contraintes, mais celles-ci sont assez génériques pour que le résultat soit applicable à l'ensemble (ou au moins la grande majorité) des problèmes d'optimisations des largeurs (avec d'autres fonctions objectifs et d'autres contraintes).

Passons maintenant en revue les différents travaux effectués sur l'optimisation des largeurs. L'idée n'est pas d'être le plus exhaustif possible mais de montrer la diversité des approches proposées et l'évolution de celles-ci. Les travaux des principaux acteurs du domaine sont décrits, ainsi que des travaux un peu moins connus.

Ki-Il Kum et Wonyong Sung ont initié les travaux sur l'optimisation des largeurs [97, 98]. Leurs premiers travaux consistaient d'une part à minimiser la largeur des parties entières des différents signaux par simulation, puis à effectuer une optimisation des largeurs fractionnaires pour minimiser le coût d'implémentation matériel tout en respectant une contrainte de performances, d'autre part. Afin de minimiser le nombre de largeurs à optimiser, une phase de groupement des opérateurs partageant la même largeur est effectuée avant la phase d'optimisation. Pour chaque vecteur de largeurs, la performance est évaluée elle aussi par simulation, la fonction de coût à optimiser est calculée d'après une table contenant les coûts des opérations en fonction des largeurs. Une approche exhaustive et une approche utilisant une métaheuristique sont comparées pour la phase d'optimisation. La comparaison montre que leur métaheuristique donne en moins de temps un résultat très proche de la solution de la recherche exhaustive (le coût d'implémentation additionnel est en moyenne inférieur à 5%). Dans [56], leur méthode de regroupement des opérateurs est améliorée et le partage des ressources matérielles est pris en compte. De ce

2. Un problème de décision $\mathcal{P}$ est dit NP-difficile s'il existe, pour tout problème $\mathcal{R}$ de la classe NP, une réduction en temps polynomiale de $\mathcal{R}$ vers $\mathcal{P}$.

fait, un algorithme d'ordonnancement est proposé, puis amélioré dans [55], qui accélère la phase de regroupement et donc le temps global de l'optimisation des largeurs.

Cantin *et al.* [15] proposent en 2001 une méthode pour optimiser les largeurs en minimisant le coût d'implémentation matériel. Cette méthode est fortement basée sur la simulation : une première simulation est faite en virgule flottante, des largeurs minimales sont déterminées, une simulation est effectuée en virgule fixe avec ces largeurs minimales et le calcul d'erreur s'effectue en comparant les simulations flottantes et fixe. L'optimisation consiste alors à trouver les largeurs optimales en simulant à chaque fois l'implémentation fixe et en comparant cette simulation à la simulation flottante. Dans cet article, quatre algorithme d'optimisation sont proposés, une recherche exhaustive, deux recherches gloutonnes (l'une partant d'une solution initiale minimale en ajoutant un bit par itération, l'autre partant d'une solution maximale en soustrayant un bit par itération), et une procédure évolutive (une métaheurstique convertissant un à un les opérandes flottants en opérandes fixe en faisant varier la largeurs de ceux-ci à partir de 0 jusqu'à répondre aux exigences souhaitées). Dans [13], cette méthode est appliquée à neuf algorithmes d'optimisation, dont les procédures gloutonnes, exhaustive et évolutive déjà discutées dans [15], mais aussi la métaheuristique de Jum et Sung proposée dans [98], un algorithme de Séparation et Évaluation, un algorithme de Recuit Simulé, un hybride de deux recherches gloutonnes (on part d'une solution minimale, on ajoute b bits distribués entre les différentes largeurs, lui on décrémente d'un bit par itération) et une procédure pré-planifiée. Une nouvelle métrique, prenant en compte à la fois le coût d'implémentation, les erreurs de calculs et les spécifications de l'utilisateur, est considérée comme objectif à optimiser. Cette métrique est améliorée dans [14] pour prendre en compte plusieurs modèles d'erreurs entre les simulations flottante et fixe.

Constantinides *et al.* [20, 21] proposent aussi en 2001 leur première méthodologie d'optimisation des largeurs. Basée sur une représentation sous forme de graphe flot de données, les auteurs adoptent une approche analytique de l'évaluation des erreurs (en propageant les formats virgule fixe à travers leurs graphes) représentées par des bruits. Ils décrivent ensuite leur problème d'optimisation sous forme d'une Programmation Linéaire Mixte (*Mixed Integer Linear Progamming*), qu'ils résolvent en utilisant le solveur `BonsaiG` [35], dont l'algorithme de résolution est basé sur un BaB. Dans [20], une métaheuristique est également présentée, reprise dans [19]. Il s'agit d'un algorithme glouton dont la recherche du voisinage est affinée par une heuristique dépendant de la fonction objectif. L'évaluation des erreurs de précision est améliorée dans [17] par un hybride d'évaluations analytique et par simulation. L'essentiel de ces travaux est repris dans le livre [18]. Dans [63] les auteurs proposent une réso-

lution de leur problème à l'aide d'un algorithme de Recuit Simulé Adaptatif [49] [3].

Ménard *et al.* [38] reprennent le principe de groupement des opérations partageant la même largeur de Kum et Sung dans leur méthodologie. Celle-ci consiste à coupler la synthèse d'architecture et l'optimisation des largeurs dans un processus itératif. À chaque itération, un ensemble de largeurs est déterminé et est testé dans la phase de synthèse de l'architecture. L'algorithme d'optimisation utilisé est un algorithme glouton utilisant comme critère de décision pour l'incrémentation le rapport entre le gain sur la précision et l'augmentation du coût. Cet algorithme est présenté en détail dans la thèse de Nicolas Hervé [37], avec un autre algorithme d'optimisation consistant à résoudre d'une part le problème dans le domaine continue et à discrétiser l'ensemble de largeurs résultant d'autre part (en utilisant un algorithme de type BaB). Dans [76], les auteurs proposent une méthode d'évaluation analytique des erreurs de calculs pour les filtres LTI, afin d'améliorer la rapidité du processus d'optimisation et de synthèse. Un nouvel algorithme d'optimisation est proposé dans [83], basé sur la métaheuristique GRASP, il combine une recherche tabou et une recherche locale. La notion de voisinage est définie en fonction du coût et de l'évaluation des erreurs de calcul. Cet algorithme est pleinement décrit dans la thèse d'Hai-Nam Nguyen [82]. Dans [81], la détermination des largeurs des parties entières, généralement effectuée analytiquement ou par simulation, est modélisée par un problème d'optimisation basé sur la tolérance à un certains degré de débordement. En effet, la simulation prend généralement du temps mais la méthode analytique est souvent pessimiste (pour assurer l'absence de tout débordement), et diminuer ces largeurs peut aussi faire diminuer le coût d'implémentation. Enfin, dans [86], Parashar, Ménard et Sentieys proposent un algorithme en temps polynomial pour résoudre le problème d'optimisation des largeurs en utilisant une relaxation convexe du problème et un solveur dédié à cette classe de problème.

Ahmadi et Zwolinski [1] proposent une optimisation multi-objectifs de la surface et du bruit. Pour l'optimisation, ils utilisent un algorithme génétique classique. Dans [3], cette approche est insérée dans un processus de synthèse de haut niveau, et dans [2] le modèle de bruit utilisé est modifié en un modèle symbolique avec une représentation algébrique du bruit.

Pour finir, on peut trouver quelques travaux où les auteurs se sont intéressés occasionnellement ou très récemment au problème d'optimisation. L'un de ces travaux, parmi les premiers sur ce sujet, par Wakekar et Parker [105], propose comme Kum et Sung quelques années plutôt, une solution algorithmique pour

3. Lester Ingber propose un outil implémentant son algorithme sur sa page personnelle, utilisé par Lee *et al.* dans leur article : `http://www.ingber.com/#ASA`

regrouper les opérations qui devraient avoir des largeurs proches, pour ainsi limiter le nombre de largeurs différentes et minimiser le coût d'implémentation en terme de surface. L'algorithme utilisé pour l'optimisation des largeurs de ces groupes consiste en une recherche aléatoire basée sur un algorithme génétique (détaillé dans la thèse de Wadekar [93]). Un algorithme génétique est également utilisé par Leban et Tasic dans [62] pour une classe particulière de FIR.

Belanovic et Rupp, dans [87], propose une optimisation des largeurs incluse dans leur outil de conversion flottant vers fixe, `Fixify` [88]. Cette optimisation offre trois algorithmes possibles : une recherche exhaustive sur des ensembles restreints de valeurs possibles, une recherche gloutonne et un BaB.

Dans [64], Lee et Gerstlauer utilise le même algorithme de recuit simulé adaptatif que Constinides *et al.* dans [63], et considère une nouvelle approche heuristique pour simplifié en amont la recherche des largeurs en utilisant notamment un modèle de pseudo bruit de quantification à la place du modèle usuel. Enfin, récemment, Vakili *et al.* [101] ont proposé une approche utilisant l'arithmétique affine pour déterminer les intervalles des variables et l'évaluation de l'erreur. Deux algorithmes sont proposés, mixant évaluation par simulation et évaluation analytique de l'erreur, le premier est un algorithme glouton et l'autre un algorithme de BaB avec une métaheuristique qui analyse le problème et donne un arbre de recherche syntaxique.

Le bilan à dresser de cet état de l'art est qu'il existe de nombreuses méthodologies et approches du problème d'optimisation des largeurs. Cette diversité s'explique notamment par les objectifs que chacun cherche à optimiser, ceux-ci pouvant être l'exactitude numérique, la surface ou encore le coût de calcul, induits par un ensemble de largeurs. Le formalisme virgule fixe et l'approche algorithmique, souvent différents d'une équipe de recherche à une autre, interviennent également dans les causes de cette diversité. C'est pour ces mêmes raisons, et surtout parce que nous nous limitons aux filtres LTI, pour lesquels nous avons des outils plus fins tels que l'approche par le WCPG ou l'arithmétique modulaire, que nous proposons une nouvelle approche. En effet, la prochaine section propose une solution algorithmique supprimant de nos calculs les bits les qui sont le moins significatifs pour le résultat final, si l'on tolère un arrondi fidèle sur ce résultat. Cette première approche algorithmique permettra par la suite de formuler une contrainte de notre problème d'optimisation.

5.2 Première approche de réduction des bits des largeurs : le formatage de bits

La première approche pour prendre en compte le paradigme des largeurs multiples a été d'utiliser la propriété des filtres linéaires concernant le comportement d'un intervalle au travers de ces filtres (proposition 3.2). En effet, on

a vu dans le chapitre 3 que ce résultat pouvait nous permettre de connaître le format du résultat d'un produit scalaire, et ainsi, en utilisant une propriété de l'arithmétique en complément à deux, de supprimer des bits de poids forts inutiles. Nous allons alors montrer que la connaissance de la position du *lsb* du résultat peut elle aussi être exploitée pour limiter le nombre de bits des termes à sommer, au niveau de leurs bits les moins significatifs.

5.2.1 Problématique et notations

Comme dans le chapitre précédent, on s'intéresse à un produit scalaire $s = \sum_{i=1}^{N} \boldsymbol{p}_i$, où $\boldsymbol{p}_i = \boldsymbol{c}_i \times \boldsymbol{v}_i$ pour $1 \leqslant i \leqslant N$, dont le résultat est sur un format virgule fixe connu par la proposition 3.2, noté (m_s, ℓ_s). On rappelle que l'on considère une implémentation matérielle, par conséquent les $\boldsymbol{p}_i$ peuvent avoir des largeurs différentes, et la somme des termes peut s'effectuer sans perte sur le format optimal. Puisque le FPF du résultat est connu, on suppose qu'on calcule d'abord une somme intermédiaire exacte (sans perte par rapport aux $\boldsymbol{p}_i$), puis ce résultat est ramené sur le format final pour obtenir le résultat final. On note alors :

- (m_i, ℓ_i) le format de $\boldsymbol{p}_i$ pour $1 \leqslant i \leqslant N$,
- (m_{opt}, ℓ_{opt}) le format intermédiaire optimal, avec :

$$m_{opt} \triangleq \max_i(m_i) + \lceil \log_2(N-1) \rceil, \tag{5.1}$$

$$\ell_{opt} \triangleq \min_i(\ell_i), \tag{5.2}$$

  où $\lceil \log_2(N-1) \rceil$ correspond au nombre de bits de retenus possibles lors d'une somme de N termes.
- s_{opt} la somme exacte intermédiaire des $\boldsymbol{p}_i$ au format (m_{opt}, ℓ_{opt}), et s la somme finale au format (m_s, ℓ_s),
- $\boldsymbol{p}_{i,j}$ le j-ième bit de $\boldsymbol{p}_i$.

Les notations sont illustrés par la figure 5.1a.

On a en fait $s = \diamond_d(s_{opt})$, avec $\diamond_d \in \{\circ_d(s), \nabla_d(s)\}$, c.-à-d. soit l'arrondi au plus proche, soit l'arrondi par troncature, selon le choix du mode d'arrondi pour l'implémentation.

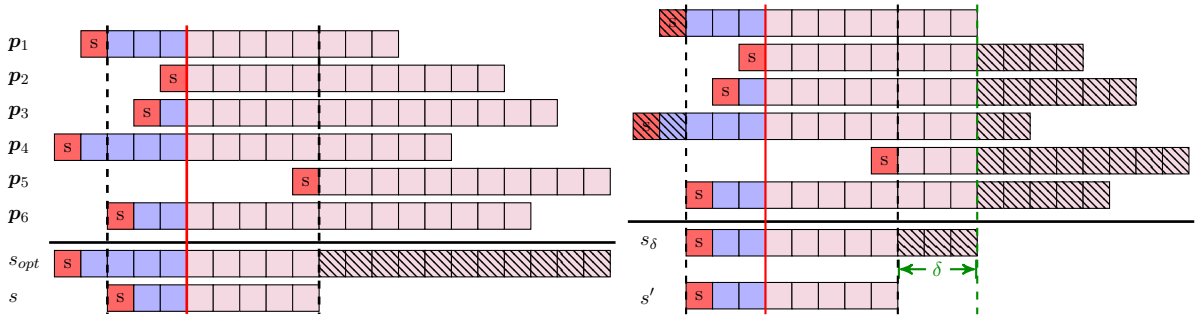
On sait déjà d'après la règle de l'arithmétique en complément à deux vue au chapitre 3 que l'on peut supprimer des $\boldsymbol{p}_i$ les bits dont la position est strictement supérieure à m_s. On veut alors déterminer une règle similaire pour les bits dont la position est plus petite que ℓ_s. Or, on sait qu'arrondir des bits implique une dégradation de la valeur arrondie, et cette dégradation peut impacter le résultat final même si ces bits ont une position inférieure à ℓ_s. De manière générale, on ne pourra jamais garantir d'avoir un arrondi exact (selon le mode d'arrondi choisi) sauf en conservant tous les bits de $\boldsymbol{p}_i$.

Le but est alors de déterminer un nombre minimal de bits à conserver au-delà de la position ℓ_s, noté δ, de calculer une somme intermédiaire s_δ en ne

considérant que les bits $\boldsymbol{p}_{i,j}$ pour $1 \leqslant i \leqslant N$ et $\ell_s - \delta \leqslant j \leqslant m_s$, de telle sorte que l'arrondi de cette somme s_δ sur le format final (m_s, ℓ_s) soit un arrondi fidèle de s_{opt}. On notera s' cette seconde somme finale.

Le figure 5.1 illustre cette approche. La figure 5.1a montre la somme telle qu'on la calcule dans le cas optimal en se ramenant ensuite sur le format final, et la figure 5.1b montre la somme intermédiaire des termes $\boldsymbol{p}_i$ formatés sur le format $(m_s, \ell_s - \delta)$, puis ramenée sur le format final (m_s, ℓ_s).

En fait, on ne peut garantir un arrondi fidèle du résultat optimal que dans le cas de l'arrondi au plus proche. En effet, on peut facilement exhiber un exemple, dans le cas de l'arrondi par troncature, où $s_{opt} - s' = 2^{\ell_s}$ alors que l'arrondi fidèle signifierait $s_{opt} - s' < 2^{\ell_S}$. Il suffit pour cela de considérer, pour n'importe quel $\delta \in \mathbb{N}^*$, une somme s_δ où les bits en position $\ell_s - \delta$ à $\ell_s - 1$ (les δ bits en plus) sont tous égaux à 1, et de considérer que la retenue engendrée par les bits tronqués (en position strictement inférieure à $\ell_s - \delta$) est strictement positive. Ainsi, dans le calcul de s_{opt} la retenue se propagerait au moins jusqu'au bit en position ℓ_s tandis que dans le calcul de s_δ on n'aurait pas de retenue et tous les bits intermédiaires à 1 seraient tronqués lors du passage à s'. On n'aurait alors bien $s_{opt} - s' = 2^{\ell_s}$.



(a) La somme des p_i est calculée sur le format optimale, puis ramenée sur le format final attendu.

(b) Les p_i sont formatés, la somme calculée sur un format intermédiaire, et cette somme est ramené sur le format final attendu.

Figure 5.1 – Illustration du formatage de bits.

5.2.2 Calcul de δ et résumé de la méthode

La proposition 5.1 donne la valeur de δ pour les deux modes d'arrondi considérés lors de la suppression des bits $\{\boldsymbol{p}_{ij}\}_{j \leqslant \ell_s - \delta}$.

Proposition 5.1. *Pour le mode d'arrondi au plus proche, le plus petit entier δ tel que $s' = \star_{\ell_s}(s_{opt})$ est donné par :*

$$\delta \triangleq \lfloor \log_2(N) \rfloor + 1 \tag{5.3}$$

Pour le mode d'arrondi par troncature, le plus petit entier δ qui garantit $s - s' \leqslant 2^{\ell_s}$ (ce qui est aussi équivalent à avoir $s_{opt} - s' \leqslant 2^{\ell_s+1}$) est donné par :

$$\delta \triangleq \lfloor \log_2(N-1) \rfloor + 1 \tag{5.4}$$

Démonstration. La preuve complète pour les deux modes d'arrondi se trouve en annexe C.4. □

Une fois que l'on a calculé δ, on peut alors formater tous les p_i, pour $1 \leqslant i \leqslant N$, au format $(m_s, \ell_s - \delta)$. Par conséquent, puisque tous les termes ont le même format, on peut les sommer dans n'importe quel ordre sans avoir à considérer d'alignement (et donc sans avoir à prendre en compte les erreurs impliquées par ces alignements). De ce fait, les seules dégradations numériques sont celles engendrées par les formatages avant de calculer la somme, et la dégradation cumulée globale, commune à tous les ordres possibles, peut être évaluée (sous forme de bruit ou d'intervalle) avant le choix du schéma d'évaluation.

Proposition 5.2. *L'intervalle cumulé de l'erreur, noté $[\underline{e}; \overline{e}]$, est, en utilisant la méthode de formatage, donné par :*

- *Arrondi par troncature :*

$$\underline{e} = \sum_{i \in I} (-2^{\ell_s - \delta} + 2^{\ell_i}) - 2^{\ell_s} + 2^{\ell_s - \delta} \tag{5.5a}$$

$$\overline{e} = 0 \tag{5.5b}$$

- *Arrondi au plus proche :*

$$\underline{e} = \sum_{i \in I} (-2^{\ell_s - \delta - 1} + 2^{\ell_i}) - 2^{\ell_s - 1} + 2^{\ell_s - \delta} \tag{5.6a}$$

$$\overline{e} = \sum_{i \in I} (2^{\ell_s - \delta - 1}) + 2^{\ell_s - 1} \tag{5.6b}$$

Le bruit cumulé de l'erreur est donné par ses moments d'ordre un et deux, notés respectivement μ_e et σ_e, avec :

- *Arrondi par troncature :*

$$\mu_e = \sum_{i \in I} (2^{\ell_s - \delta - 1} - 2^{\ell_i - 1}) + 2^{\ell_s - 1} - 2^{\ell_s - \delta - 1} \tag{5.7}$$

$$\sigma_e^2 = \sum_{i \in I} \frac{2^{2(\ell_s - \delta)}}{12} (1 - 2^{-2(\ell_s - \delta - \ell_i)}) \tag{5.8}$$

- *Arrondi au plus proche :*

$$\mu_e = \sum_{i \in I} (2^{\ell_i - 1}) + 2^{\ell_f - \delta - 1} \tag{5.9}$$

$$\sigma_e^2 = \sum_{i \in I} \frac{2^{2(\ell_s - \delta)}}{12} (1 - 2^{-2(\ell_s - \delta - \ell_i)}) \tag{5.10}$$

L'ensemble I est l'ensemble des indices i des p_i qui ont un formatage à droite positif, c.-à-d.

$$I = \{i \mid 1 \leqslant i \leqslant N \text{ et } \ell_s - \delta > \ell_i\} \tag{5.11}$$

avec ℓ_i le lsb de p_i pour $1 \leqslant i \leqslant N$.

Démonstration. Il suffit d'appliquer les équations vues dans les parties 4.3.1 et 4.3.2. □

La méthode peut se résumer ainsi :

1. on calcule δ selon le mode d'arrondi choisi,
2. on formate tous les produits p_i pour $1 \leqslant i \leqslant N$ sur le format intermédiaire $(m_s, \ell_s - \delta)$, où (m_s, ℓ_s) est le format du résultat final s,
3. on calcule la somme intermédiaire s_δ sur le format intermédiaire,
4. on arrondit s_δ sur le format final (en supprimant les δ bits de garde et en arrondissant selon le mode d'arrondi).

5.2.3 Exemple

On reprend la forme directe I de notre exemple fil rouge (voir section 3.4) afin d'illustrer cette méthode. Le mode d'arrondi choisi est l'arrondi par troncature et on considère dans cet exemple que les coefficients et les variables sont représentés sur 16 bits. Il y a sept produits à calculer et en appliquant la proposition 5.1 on obtient $\delta = 3$. Par conséquent, puisque $y(k) \in [-23.7637; 19.9824]$ pour tout $k \geqslant 0$, le format de $y(k)$ est $(m_y, \ell_y) = (5, -10)$ et le format de la somme intermédiaire s_δ est $(m_{s_\delta}, \ell_{s_\delta}) = (5, -13)$. La figure 5.2 illustre l'oSoP obtenu en appliquant le formatage. Chaque multiplieur possède son propre décalage pour mettre le résultat sur le format intermédiaire commun. On peut alors évaluer l'intervalle des erreurs de calcul commise sur $y(k)$ à chaque instant $k \geqslant 0$ en utilisant la proposition 5.2, on obtient $[\underline{e}; \overline{e}] = [-0.00170892; 0]$. On peut constater que comme attendu, la borne sur l'erreur de calcul à tout instant k est plus petite en valeur absolue que $2^{\ell_y+1} = 2^{-9} = 0.001953125$.

Finalement, à partir de cet intervalle on peut calculer l'impact de cette erreur sur le résultat final en appliquant le corollaire 3.1 avec les valeurs données par l'équation (3.39). Si $\delta y(k)$ désigne la différence entre la sortie calculée en appliquant le formatage et la sortie de référence à tout instant $k \geqslant 0$, avec $\delta y(k) \in [\underline{\delta y}; \overline{\delta y}]$, alors on a :

$$[\underline{\delta y}; \overline{\delta y}] = [-0.00235806; 0.000584955]. \tag{5.12}$$

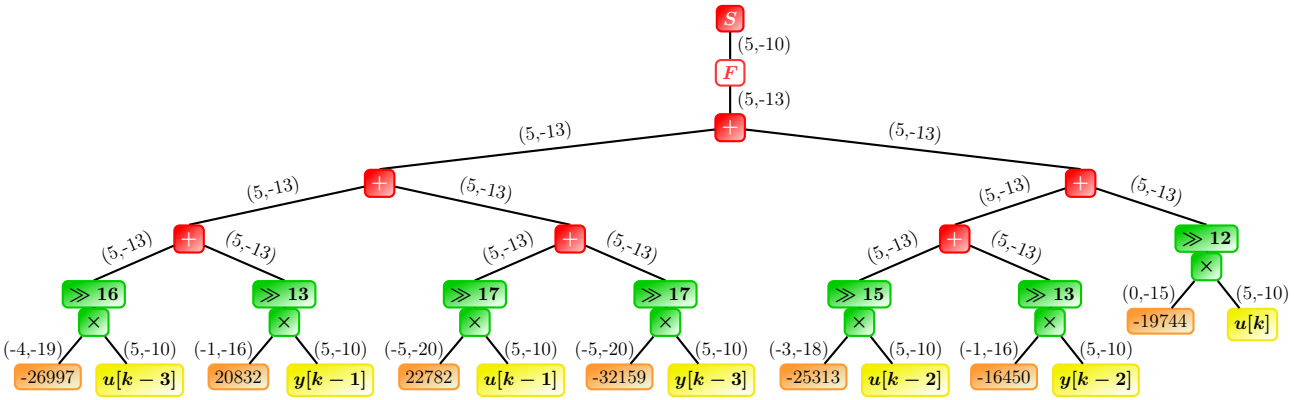


FIGURE 5.2 – oSoP obtenu en appliquant le formatage.

5.3 Formalisation du problème d'optimisation

On l'a vu en début de chapitre, le but de l'optimisation des largeurs est de minimiser une fonction de coûts dépendant des largeurs, tout en respectant des contraintes de précision.

Le système considéré est un filtre $\mathcal{H}$ exprimé sous forme implicite dont l'algorithme, donné par les équations 1.48, peut s'écrire ainsi :

$$\boldsymbol{v}' = \boldsymbol{Z}' \cdot \boldsymbol{v} \tag{5.13}$$

avec

$$\boldsymbol{v}' \triangleq \begin{pmatrix} \boldsymbol{t}(k+1) \\ \boldsymbol{x}(k+1) \\ \boldsymbol{y}(k) \end{pmatrix}, \quad \boldsymbol{Z}' \triangleq \begin{pmatrix} -\boldsymbol{J}' & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix}, \quad \text{et} \quad \boldsymbol{v} \triangleq \begin{pmatrix} \boldsymbol{t}(k) \\ \boldsymbol{x}(k) \\ \boldsymbol{u}(k) \end{pmatrix}. \tag{5.14}$$

On notera également $n' \triangleq n_t + n_x + n_y$ et $n \triangleq n_t + n_x + n_u$, on a ainsi $\boldsymbol{v}' \in \mathbb{R}^{n'}$, $\boldsymbol{v} \in \mathbb{R}^n$ et $\boldsymbol{Z}' \in \mathbb{R}^{n' \times n}$.

Les vecteurs $\boldsymbol{w}_{\boldsymbol{v}'}$ et $\boldsymbol{w}_{\boldsymbol{v}}$ désignent les vecteurs de largeurs des termes de $\boldsymbol{v}'$ et $\boldsymbol{v}$ respectivement, c'est-à-dire $(\boldsymbol{w}_{\boldsymbol{\nu}})_i \triangleq \boldsymbol{w}_{\boldsymbol{\nu}_i}$ est la largeur de $\boldsymbol{\nu}_i$, pour $\boldsymbol{\nu} \in \{\boldsymbol{v}', \boldsymbol{v}\}$. De la même manière, $\boldsymbol{w}_{\boldsymbol{Z}'}$ désigne la matrice des largeurs des termes de $\boldsymbol{Z}'$, c'est-à-dire $(\boldsymbol{w}_{\boldsymbol{Z}'})_{ij} \triangleq \boldsymbol{w}_{\boldsymbol{Z}'_{ij}}$ est la largeur de $\boldsymbol{Z}'_{ij}$.

On veut alors optimiser les largeurs $\boldsymbol{w}_{\boldsymbol{v}'}$ et $\boldsymbol{w}_{\boldsymbol{Z}'}$ tout en répondant à nos contraintes.

Un problème d'optimisation combinatoire, avec ces variables, s'exprimera

généralement ainsi :

$$\begin{aligned} \max / \min \quad & f(\boldsymbol{w_{Z'}}, \boldsymbol{w_{v'}}) && (5.15a) \\ \text{tel que} \quad & g_1(\boldsymbol{w_{Z'}}, \boldsymbol{w_{v'}}) \leqslant 0 && (5.15b) \\ & g_2(\boldsymbol{w_{Z'}}, \boldsymbol{w_{v'}}) \leqslant 0 && (5.15c) \\ & \vdots \\ & g_{N_c}(\boldsymbol{w_{Z'}}, \boldsymbol{w_{v'}}) \leqslant 0, && (5.15d) \end{aligned}$$

où la fonction f est appelé fonction objectif et représenter l'objectif à réaliser, les équations $g_i(\boldsymbol{w_{Z'}}, \boldsymbol{w_{v'}}) \leqslant 0$ pour $1 \leqslant i \leqslant N_c$ sont les contraintes à respecter et N_c est le nombre de contraintes.

Remarque 5.1. *Nous allons voir par la suite que les largeurs du vecteur $\boldsymbol{v}$ n'apparaissent pas dans les contraintes décrites. En fait, puisque les vecteurs $\boldsymbol{t}$ et $\boldsymbol{x}$ apparaissent à la fois dans les vecteurs $\boldsymbol{v'}$ et $\boldsymbol{v}$ (à des instants différents mais la largeur d'une variable reste la même quelque soit l'instant), seul le vecteur d'entrées $\boldsymbol{u}$ n'apparaît pas dans ces contraintes. Cela vient du fait que $\boldsymbol{u}$ est l'entrée du système et est donc considéré comme une donnée exacte du problème, c'est donc à l'utilisateur de choisir les largeurs du vecteur $\boldsymbol{w_u}$.*

La fonction objectif f peut par exemple évaluer le coût d'implémentation étant donné un jeu de largeurs. Sur un FPGA, le coût peut être le nombre de LUT nécessaires pour implémenter le filtre pour un ensemble de largeurs donné (ce nombre de LUT peut être approché par une fonction en fonction des largeurs, et les résultats sont stockés dans une table [38, 81], ou bien mesuré *a posteriori*). Minimiser le nombre de LUT revient à minimiser la surface d'implémentation, ce que le modèle utilisé dans [20] considère.

Étant donnée que ces travaux constituent pour nous une première approche dans le domaine de l'optimisation combinatoire, nous avons opté pour un modèle plus simple et linéaire, où l'objectif est de minimiser l'ensemble des largeurs de notre problème, c'est-à-dire on considère la fonction objectif f suivante :

$$f(\boldsymbol{w_Z}, \boldsymbol{w_{v'}}) = \sum_{i=1}^{n'} \boldsymbol{w_{v'}}_i + \sum_{i=1}^{n'} \sum_{j=1}^{n} \boldsymbol{w_{Z'}}_{ij}. \tag{5.16}$$

Cette fonction objectif a par exemple été utilisée dans les travaux de Blenovic *et al.* [87].

Nous avons maintenant besoin des contraintes de précision à respecter pour définir l'ensemble des solutions réalisables pour nos largeurs. Pour notre problème, on considère deux types de contraintes de précision, les contraintes liées au formatage de bits, et les contraintes de précision sur l'erreur finale, décrites dans la suite.

5.3.1 Formalisation des contraintes de formatage de bits

On l'a vu, la méthode du formatage de bits impose, étant donné le format virgule fixe d'une somme, un format intermédiaire pour les termes à sommer, c'est-à-dire pour les produits constante/variable. On a également vu (partie 2.4.2.4) qu'on connait, lors d'un produit constante/variable, la position du bit potentiellement faux le plus significatif du produit. L'idée est alors de combiner ces deux informations en écrivant une contrainte telle que les bits des produits sur le format intermédiaire (imposé par le formatage de bits) soient tous exacts. Cela revient en effet à choisir les largeurs des constantes et des variables telles que tous les bits potentiellement faux de leurs produits soient formatés par le format intermédiaire.

La position du bit potentiellement faux le plus significatif d'un produit $\boldsymbol{c}_i \times \boldsymbol{v}_i$, où $\boldsymbol{c}_i$ est une constante et $\boldsymbol{v}_i$ une variable, est $\boldsymbol{\ell}_{\boldsymbol{c}_i} + \boldsymbol{m}_{\boldsymbol{v}_i}$, d'après l'équation (2.60). De plus, dans une somme de produits $s = \sum_{i=1}^{N} \boldsymbol{c}_i \times \boldsymbol{v}_i$, le formatage de bits impose à chaque produit d'avoir son lsb en position $\ell_s - \delta$, avec ℓ_s le lsb final de s et δ donné par la proposition 5.1. Pour que le formatage ne considère, dans son format intermédiaire, que les bits exacts des produits, il faut alors vérifier :

$$\ell_s - \delta > \boldsymbol{\ell}_{\boldsymbol{c}_i} + \boldsymbol{m}_{\boldsymbol{v}_i} \quad \forall 1 \leqslant i \leqslant N, \tag{5.17}$$

qui devient, en utilisant l'égalité $w = m - \ell + 1$ et le fait que l'inégalité n'implique que des entiers (donc $a > b$ est équivalent à $a \geqslant b + 1$) :

$$\boldsymbol{w}_{\boldsymbol{c}_i} - w_s \geqslant \boldsymbol{m}_{\boldsymbol{c}_i} + \boldsymbol{m}_{\boldsymbol{v}_i} - m_s + \delta + 1 \quad \forall 1 \leqslant i \leqslant N. \tag{5.18}$$

On peut déjà noter deux choses. Premièrement, il y a une contraintes de formatage par produit d'un SoP. Ensuite, le membre de droite est une constante, et donc seules les largeurs des constantes et du résultat du produit scalaire sont prises en compte, les largeurs des variables $\boldsymbol{v}_i$ n'ont aucune incidence sur la contrainte que l'on souhaite respecter. De plus, on précise que ces contraintes sont à considérer uniquement pour des constantes non nulles. En effet, si $\boldsymbol{c}_i = 0$ pour un certains $1 \leqslant i \leqslant N$, alors le produit $\boldsymbol{c}_i \times \boldsymbol{v}_i$ est nul et il n'y a pas lieu d'exprimer une contrainte pour ce produit.

Si le produit scalaire consiste en une simple multiplication par une constante égale à 1, alors il s'agit d'une affectation et donc la contrainte n'est pas considérée. Pour les constantes égales à des puissances de 2, dont 1 si la somme de produits est constituée de plus d'un produit, il n'y a pas de produit à faire, un simple décalage suffit, mais ce cas n'est pas détecté dans notre implémentation. De plus, dans ce cas, les équations des contraintes sont toujours vraies, bien que moins fines qu'un traitement particulier.

Appliqué au filtre $\mathcal{H}$ et à son implémentation sous forme SIF, avec les notations de l'équation (5.13), l'équation (5.18) donne les inégalités suivantes :

$$\boldsymbol{w}_{\boldsymbol{Z'}_{i,j}} - \boldsymbol{w}_{\boldsymbol{v}'_i} \geqslant \boldsymbol{m}_{\boldsymbol{Z'}_{i,j}} + \boldsymbol{m}_{\boldsymbol{v}_j} - \boldsymbol{m}_{\boldsymbol{v}'_i} + \boldsymbol{\delta}_{\boldsymbol{v}'_i} + 1, \tag{5.19}$$

pour $\boldsymbol{Z'}_{i,j} \neq 0$ et pour tout $1 \leqslant i \leqslant n'$ et tout $1 \leqslant j \leqslant n$. Si $\boldsymbol{Z'}_{i,j} = 0$ pour un couple (i, j) tel que $1 \leqslant i \leqslant n'$ et $1 \leqslant j \leqslant n$, on n'a pas besoin de contrainte faisant intervenir $\boldsymbol{w}_{\boldsymbol{Z'}_{i,j}}$ puisque ce coefficient ne sera pas pris en compte dans l'algorithme implémenté. En effet, on rappelle qu'on ne veut pas stocker toutes la matrices $\boldsymbol{Z'}$ des coefficients sur la cible, mais uniquement implanter l'algorithme qui aura été implémenté à partir de la réalisation sous forme implicite, donc seuls les coefficients non nuls seront pris en compte (et ont donc besoin de contraintes pour minimiser leurs largeurs).

5.3.2 Formalisation des contraintes d'erreurs

Ce second type de contraintes concerne l'erreur de précision sur la sortie finale $\boldsymbol{y}$. Lorsqu'on applique la méthode du formatage de bits sur un SoP, on sait évaluer les erreurs de calcul (voir proposition 5.2), et à partir de ces évaluations sur tous les SoPs d'un système on peut alors calculer l'intervalle théorique de l'erreur finale (ou les moments d'ordre 1 et 2 si l'erreur est modélisée par un bruit) par le corollaire 3.1.

On veut alors pouvoir exprimer nos différentes largeurs de façon à ce que les bornes de l'intervalle théorique de l'erreur finale soient majorées par une constante fixée par l'utilisateur.

Soit $[\underline{\boldsymbol{\delta y}}; \overline{\boldsymbol{\delta y}}]$ l'intervalle théorique de l'erreur finale donné par le corollaire 3.1 pour l'entrée $\boldsymbol{\varepsilon}$, avec :

$$\boldsymbol{\varepsilon} \triangleq \begin{pmatrix} [\underline{\varepsilon_1}; \overline{\varepsilon_1}] \\ \vdots \\ [\underline{\varepsilon_{n'}}; \overline{\varepsilon_{n'}}] \end{pmatrix}, \tag{5.20}$$

où $[\underline{\varepsilon_i}; \overline{\varepsilon_i}]$ est l'intervalle d'erreur sur $\boldsymbol{v}'_i$ donné par la proposition 5.2. Soit $\boldsymbol{\xi}$ la borne sur l'erreur finale choisie par l'utilisateur, on veut alors choisir nos largeurs pour vérifier :

$$\max\left(\left|\underline{\boldsymbol{\delta y}_i}\right|, \left|\overline{\boldsymbol{\delta y}_i}\right|\right) < \boldsymbol{\xi}_i \quad \forall 1 \leqslant i \leqslant n'. \tag{5.21}$$

Il faut donc faire apparaître les largeurs dans l'équation (5.21). Pour cela, on revient à l'expression des intervalles $[\underline{\varepsilon_i}; \overline{\varepsilon_i}]$. On peut en effet borner ces intervalles d'erreurs de calcul par un terme fonction des largeurs $\boldsymbol{w}_{\boldsymbol{v}'}$. Puisque les intervalles $[\underline{\varepsilon_i}; \overline{\varepsilon_i}]$ ont une expression différente selon le mode d'arrondis, il est nécessaire de gérer les deux cas séparément.

5.3.2.1 Mode d'arrondi par troncature

D'après les équations (5.5) et les notations de l'équation (5.13), on a les majorations suivantes dans le cas de l'arrondi par troncature :

$$\left|\underline{\varepsilon_i}\right| < 2^{\boldsymbol{\ell}_{\boldsymbol{v}'_i}} + 1, \tag{5.22a}$$

$$\left|\overline{\varepsilon_i}\right| = 0, \tag{5.22b}$$

pour tout $1 \leqslant i \leqslant n'$.

On en déduit les majorations suivantes pour l'intervalle $[\underline{\boldsymbol{\delta y}}; \overline{\boldsymbol{\delta y}}]$, d'après le corollaire 3.1 :

$$\begin{aligned}
\left|\underline{\boldsymbol{\delta y}_i}\right| &= \sum_{j=1}^{n'} \left| (\langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}})_{i,j} \frac{\overline{\boldsymbol{\varepsilon}_j} + \underline{\boldsymbol{\varepsilon}_j}}{2} - \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{i,j} \frac{\overline{\boldsymbol{\varepsilon}_j} - \underline{\boldsymbol{\varepsilon}_j}}{2} \right| && (5.23)\\
&< \sum_{j=1}^{n'} \left| (\langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}})_{i,j} + \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{i,j} \right| \cdot 2^{\boldsymbol{\ell}_{\boldsymbol{v}'_j}} && (5.24)\\
\left|\overline{\boldsymbol{\delta y}_i}\right| &< \sum_{j=1}^{n'} \left| (\langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}})_{i,j} - \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{ij} \right| \cdot 2^{\boldsymbol{\ell}_{\boldsymbol{v}'_j}} && (5.25)
\end{aligned}$$

pour tout $1 \leqslant i \leqslant n_y$. On peut également utiliser une écriture matricielle pour plus de lisibilité, on obtient alors :

$$\left|\underline{\boldsymbol{\delta y}}\right| < \left|\langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}} + \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle\right| \cdot 2^{\boldsymbol{\ell}_{\boldsymbol{v}'}} \tag{5.26a}$$

$$\left|\overline{\boldsymbol{\delta y}}\right| < \left|\langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}} - \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle\right| \cdot 2^{\boldsymbol{\ell}_{\boldsymbol{v}'}} \tag{5.26b}$$

où $2^{\boldsymbol{\ell}_{\boldsymbol{v}'}} \triangleq \begin{pmatrix} 2^{\boldsymbol{\ell}_{\boldsymbol{v}'_1}} \\ \vdots \\ 2^{\boldsymbol{\ell}_{\boldsymbol{v}'_{n'}}} \end{pmatrix}$, le symbole '$<$' désigne dans ce cas l'infériorité terme-à-terme et la valeur absolue d'un vecteur (resp. d'une matrice) désigne le vecteur (la matrice) des valeurs absolues.

Pour respecter la contrainte (5.21), il suffit alors de montrer, utilisant l'égalité $w = m - \ell + 1$ sur les équations (5.26) :

$$\left|\langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}} + \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle\right| \cdot 2^{\boldsymbol{m}_{\boldsymbol{v}'} - \boldsymbol{w}_{\boldsymbol{v}'} + 1} < \boldsymbol{\xi} \tag{5.27a}$$

$$\left|\langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}} - \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle\right| \cdot 2^{\boldsymbol{m}_{\boldsymbol{v}'} - \boldsymbol{w}_{\boldsymbol{v}'} + 1} < \boldsymbol{\xi} \tag{5.27b}$$

avec, pour $2^{\boldsymbol{m}_{\boldsymbol{v}'} - \boldsymbol{w}_{\boldsymbol{v}'} + 1}$, une notation analogue à celle de $2^{\boldsymbol{\ell}_{\boldsymbol{v}'}}$ donnée précédemment.

Finalement, l'expression des contraintes pour le mode d'arrondi par troncature est :

$$\boldsymbol{A} \cdot 2^{-\boldsymbol{w}_{\boldsymbol{v}'}} < \mathbf{1}_{n_y} \tag{5.28a}$$

$$\boldsymbol{D} \cdot 2^{-\boldsymbol{w}_{\boldsymbol{v}'}} < \mathbf{1}_{n_y} \tag{5.28b}$$

où le symbole $<$ désigne dans ce cas l'infériorité terme à terme, et

$$\boldsymbol{A}_{i,j} \triangleq \left| (\langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}})_{i,j} + \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{i,j} \right| \cdot \frac{2^{\boldsymbol{m}_{\boldsymbol{v}'_j} + 1}}{\boldsymbol{\xi}_i} \tag{5.29a}$$

$$\boldsymbol{D}_{i,j} \triangleq \left| (\langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}})_{i,j} - \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{i,j} \right| \cdot \frac{2^{\boldsymbol{m}_{\boldsymbol{v}'_j} + 1}}{\boldsymbol{\xi}_i} \tag{5.29b}$$

pour tout $1 \leqslant i \leqslant n_y$ et tout $1 \leqslant j \leqslant n'$, et $\mathbf{1}_{n_y}$ est un vecteur de 1 de taille n_y.

5.3.2.2 Mode d'arrondi au plus proche

On veut dans un premier temps majorer $\left|\underline{\boldsymbol{\delta y}}\right|$ par ξ. Pour cela, on rappelle la formule de $\left|\underline{\boldsymbol{\delta y}}\right|$ déduite du corollaire 3.1 :

$$\left|\underline{\boldsymbol{\delta y}}\right| = \left|\langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}}\, \boldsymbol{\varepsilon}_m - \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle\, \boldsymbol{\varepsilon}_r\right|, \tag{5.30}$$

où $\langle \boldsymbol{\varepsilon}_m, \boldsymbol{\varepsilon}_r \rangle$ est le vecteur d'intervalles $[\underline{\boldsymbol{\varepsilon}}; \overline{\boldsymbol{\varepsilon}}]$ en notation centre/rayon.

On peut déduire des équations (5.6) un encadrement de $|\boldsymbol{\varepsilon}_m|$ et de $|\boldsymbol{\varepsilon}_r|$:

$$0 < |\boldsymbol{\varepsilon}_m| < 2^{\boldsymbol{\ell}_{\boldsymbol{v}'}-1}, \tag{5.31a}$$

$$0 < |\boldsymbol{\varepsilon}_r| < 2^{\boldsymbol{\ell}_{\boldsymbol{v}'}} \tag{5.31b}$$

Donc, en utilisant l'inégalité triangulaire sur la valeur absolue, on obtient, d'après (5.30) et (5.31) :

$$\begin{aligned}
\left|\underline{\boldsymbol{\delta y}}\right| &< \left|\langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}}\right| \cdot \boldsymbol{\varepsilon}_m + \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle \cdot \boldsymbol{\varepsilon}_r, && (5.32)\\
&< \left|\langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}}\right| \cdot 2^{\boldsymbol{\ell}_{\boldsymbol{v}'}-1} + \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle \cdot 2^{\boldsymbol{\ell}_{\boldsymbol{v}'}} && (5.33)\\
&< \left(\left|\langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}}\right| \cdot 2^{\boldsymbol{m}_{\boldsymbol{v}'}} + \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle \cdot 2^{\boldsymbol{m}_{\boldsymbol{v}'}+1}\right) \cdot 2^{-\boldsymbol{w}_{\boldsymbol{v}'}} && (5.34)
\end{aligned}$$

L'inégalité triangulaire utilisée pour majorer $\left|\underline{\boldsymbol{\delta y}}\right|$ implique qu'on obtient, par ce procédé, exactement la même majoration pour le vecteur des bornes supérieurs $\left|\overline{\boldsymbol{\delta y}}\right|$. Ainsi, on obtient l'expression suivante des contraintes sur l'erreur pour le mode d'arrondi au plus proche :

$$\boldsymbol{C} \cdot 2^{-\boldsymbol{w}_{\boldsymbol{v}'}} < \mathbf{1}_{n_y}, \tag{5.35}$$

avec

$$\boldsymbol{C}_{ij} \triangleq \left|(\langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{\mathrm{DC}})_{i,j}\right| \cdot \frac{2^{\boldsymbol{m}_{\boldsymbol{v}'_j}}}{\boldsymbol{\xi}_i} + \langle\!\langle \mathcal{H}_{\boldsymbol{\varepsilon}} \rangle\!\rangle_{ij} \cdot \frac{2^{\boldsymbol{m}_{\boldsymbol{v}'_j}+1}}{\boldsymbol{\xi}_i}, \tag{5.36}$$

pour tout $1 \leqslant i \leqslant n_y$ et tout $1 \leqslant j \leqslant n'$.

5.3.3 Discussion sur les différentes contraintes

Pour les deux modes d'arrondi considérés, le premier constat que l'on peut faire sur les contraintes d'erreurs est qu'elles n'impliquent que les largeurs $\boldsymbol{w}_{\boldsymbol{v}'_j}$ pour $1 \leqslant j \leqslant n'$, c'est-à-dire les largeurs des variables de sorties de la SIF de $\mathcal{H}$. De plus, si on regarde ce qu'implique ce constat sur les contraintes de formatage de bits (équation (5.19)), alors on remarque que la connaissance de $\boldsymbol{w}_{\boldsymbol{v}'}$ donne une borne minimale sur les largeurs des coefficients de la matrice

$\boldsymbol{Z'}$. En effet, en supposant que $\boldsymbol{w_{v'}}$ est connu, les contraintes de formatage de bits deviennent :

$$\boldsymbol{w_{Z'}}_{i,j} \geqslant \boldsymbol{w_{v'_i}} + \boldsymbol{m_{Z'}}_{i,j} + \boldsymbol{m_{v_j}} - \boldsymbol{m_{v'_i}} + \boldsymbol{\delta_{v'_i}} + 1, \quad \forall 1 \leqslant i \leqslant n', \quad \forall 1 \leqslant j \leqslant n, \tag{5.37}$$

avec le terme de droite de l'inégalité constant pour chaque $1 \leqslant i \leqslant n'$ et chaque $1 \leqslant j \leqslant n$. Or, puisqu'on cherche à minimiser nos largeurs et que l'équation précédente n'implique que des entiers, il suffit de prendre, pour chaque coefficient de la matrice $\boldsymbol{Z'}$, la borne minimale donnée par l'équation (5.37).

Notre problème d'optimisation peut alors s'exprimer uniquement en fonction des largeurs $\boldsymbol{w_{v'_j}}$ pour $1 \leqslant j \leqslant n'$, les largeurs des coefficients de la matrice $\boldsymbol{Z'}$ se déduisant directement à partir de $\boldsymbol{w_{v'}}$ et de l'équation (5.37).

$$\min \quad f(\boldsymbol{w_{v'}}) = \sum_{i=1}^{n'} \boldsymbol{w_{v'}}_i \tag{5.38a}$$

$$\text{tel que} \quad g_1(\boldsymbol{w_{v'}}) \leqslant 0, \tag{5.38b}$$

$$\vdots$$

$$g_N(\boldsymbol{w_{v'}}) \leqslant 0, \tag{5.38c}$$

où les contraintes $g_i(\boldsymbol{w_{v'}}) \leqslant 0$ sont définies soit à partir des équations (5.28) dans le cas du mode d'arrondi par troncature, soit par l'équation (5.35) dans le cas du mode d'arrondi au plus proche.

Une fois ce problème résolu, on déduit directement les largeurs à partir de l'équation (5.37) et de la solution trouvée pour le vecteur $\boldsymbol{w_{v'}}$.

Ce problème, ainsi formulé, fait partie de la classe de problème *Programmation Convexe Non-Linéaire en Nombre Entiers* (PCNLNE) [36]. En effet, il est clair que la fonction objectif choisie est convexe et que les contraintes d'erreurs sont non-linéaires. De plus les variables (les différentes largeurs) sont des nombres entiers.

Remarque 5.2. *Il est important de noter que cette simplification du problème d'optimisation n'est possible que par la simplicité de notre fonction objectif. En effet, avec une fonction objectif non linéaire, par exemple une fonction qui chercherait à minimiser le coût d'implémentation en fonction des largeurs, on ne pourrait pas forcément en extraire une sous-fonction objectif considérant uniquement les largeurs* $\boldsymbol{w_{v'}}$.

5.4 Résolution du problème

Dans la section précédente nous avons formalisé les contraintes de notre problème d'optimisation, nous allons maintenant abordé sa résolution. Dans

un premier temps un cas particulier est présenté dans lequel l'optimisation n'a plus lieu d'être car les contraintes donnent les bornes minimales sur les largeurs que l'on souhaite minimiser. La résolution du problème dans le cas général est décrite dans un second temps.

5.4.1 Cas particulier

Les équations (5.28a) et (5.28b) expriment, de façon matricielle, deux ensembles de contraintes qui peuvent être développées comme suit :

$$\frac{\boldsymbol{A}_{i1}}{2^{\boldsymbol{w}_{\boldsymbol{v}'_1}}} + \frac{\boldsymbol{A}_{i2}}{2^{\boldsymbol{w}_{\boldsymbol{v}'_2}}} + \ldots + \frac{\boldsymbol{A}_{in'}}{2^{\boldsymbol{w}_{\boldsymbol{v}'_{n'}}}} < 1, \quad \forall 1 \leqslant i \leqslant n_y, \tag{5.39a}$$

$$\frac{\boldsymbol{D}_{i1}}{2^{\boldsymbol{w}_{\boldsymbol{v}'_1}}} + \frac{\boldsymbol{D}_{i2}}{2^{\boldsymbol{w}_{\boldsymbol{v}'_2}}} + \ldots + \frac{\boldsymbol{D}_{in'}}{2^{\boldsymbol{w}_{\boldsymbol{v}'_{n'}}}} < 1, \quad \forall 1 \leqslant i \leqslant n_y. \tag{5.39b}$$

Il y a donc deux contraintes pour chaque sortie du filtre $\mathcal{H}$, qui s'écrivent comme des produits scalaires d'au plus n' termes, où n' correspond au nombre de produits scalaires dans le système (n' est défini comme le nombre de variables de sorties). Autrement dit, dans le cas d'un filtre SISO décrit par un seul produit scalaire, par exemple une forme directe I :

$$y(k) = \sum_{i=0}^{n} \boldsymbol{b}_i u(k-i) - \sum_{i=1}^{n} \boldsymbol{a}_i y(k-i), \tag{5.40}$$

on aurait

$$\frac{a}{2^{w_y}} < 1, \quad \frac{d}{2^{w_y}} < 1, \tag{5.41}$$

ce qui donne

$$w_y > \log_2(a), \quad w_y > \log_2(d), \tag{5.42}$$

où

$$a \triangleq \left| |\mathcal{H}_\varepsilon|_{\mathrm{DC}} + \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle_{\mathrm{wcpg}} \right| \cdot \frac{2^{m_y+1}}{\xi} \tag{5.43}$$

$$d \triangleq \left| |\mathcal{H}_\varepsilon|_{\mathrm{DC}} - \langle\!\langle \mathcal{H}_\varepsilon \rangle\!\rangle_{\mathrm{wcpg}} \right| \cdot \frac{2^{m_y+1}}{\xi} \tag{5.44}$$

De plus, l'objectif simplifié devient $\min f(w_y) = w_y$, on cherche donc le plus petit entier w_y qui respecte les deux contraintes précédentes, c'est-à-dire :

$$w_y = \lfloor \log_2(\max(a,d)) \rfloor + 1. \tag{5.45}$$

Les largeurs minimales des coefficients seraient alors, d'après l'équation (5.37) :

$$\boldsymbol{w}_{\boldsymbol{b}_i} = w_y + m_{\boldsymbol{b}_i} + m_u - m_y + \delta + 1, \quad \forall 0 \leqslant i \leqslant n, \tag{5.46a}$$

$$\boldsymbol{w}_{\boldsymbol{a}_i} = w_y + m_{\boldsymbol{a}_i} + \delta + 1, \quad \forall 1 \leqslant i \leqslant n, \tag{5.46b}$$

où δ est donné par la proposition 5.1.

Remarque 5.3. *En fait, les équations* (5.45) *et* (5.46) *devraient être des inégalités, et donnent ainsi les bornes minimales sur les largeurs des coefficients et de la sortie. Puisque l'on cherche à minimiser ses largeurs, on peut prendre ces bornes minimales et donc considérer des égalités.*

Dans ce cas là, on ne peut pas parler d'optimisation combinatoire puisque toutes les largeurs (exceptée celle de l'entrée u qui est choisie par l'utilisateur) sont directement données par des formules.

Remarque 5.4. *On peut également remarquer que si on décide d'utiliser la même largeur pour toutes les variables $w_{\boldsymbol{v}'_i}$ pour $1 \leqslant i \leqslant n'$, alors on peut directement déduire cette largeur, notée w', des équations* (5.39a) *et* (5.39b)*. En effet on a alors :*

$$w' > \log_2 \left(\sum_{j=1}^{n'} \boldsymbol{A}_{ij} \right), \quad w' > \log_2 \left(\sum_{j=1}^{n'} \boldsymbol{D}_{ij} \right), \quad \forall 1 \leqslant i \leqslant n_y \tag{5.47}$$

et donc

$$w' = \left\lfloor \max_{1 \leqslant i \leqslant n_y} \left(\log_2 \left(\max \left(\sum_{j=1}^{n'} \boldsymbol{A}_{ij}, \sum_{j=1}^{n'} \boldsymbol{D}_{ij} \right) \right) \right) \right\rfloor + 1. \tag{5.48}$$

Bien que n'entrant plus dans le cadre du paradigme des largeurs multiples, cette largeur commune répond aux contraintes d'erreurs, elle est donc réalisable, et peut donc servir de solution initiale à un algorithme de résolution du problème dans le cas général.

5.4.2 Cas général

Nous avons vu plus haut que le problème d'optimisation des équations (5.38) faisait partie de la classe PCNLNE. Le solveur Bonmin [4] [9] est dédié à la classe de problème *Programmation Convexe Non-Linéaire Mixtes* (PCNLM, *Mixtes* désigne le fait que certaines variables sont entières et les autres continues), dont la classe PCNLNE est une sous-classe.

En fait, si la fonction objectif et les contraintes sont convexes, alors Bonmin donne la solution exacte du problème relaxé (c'est-à-dire le problème où toutes les variables sont considérées comme continues), mais si les fonctions sont non-convexes, alors Bonmin peut quand même être utilisé en tant qu'heuristique. Le solveur Bonmin utilise trois différents algorithmes, un algorithme BaB, un algorithme d'*approximation externe* (*Outer Approximation*, OA), et un hybride OA/BaB, dont nous décrivons brièvement le principe.

4. Basic Open-source Nonlinear Mixed INteger programming, `https://projects.coin-or.org/Bonmin`

- Algorithme BaB : le problème initial est séparé en plusieurs sous-problèmes et on calcule des bornes inférieurs de ces sous-problèmes en résolvant leurs relaxations continues, c'est-à-dire en supposant toutes les largeurs comme des nombres réels (plus de détails dans [34]). Si un sous-problème ne donne pas de solution réalisable, on descend d'une nœud dans l'arbre en construisant de nouveaux sous-problèmes de taille inférieure, sinon on passe au sous-problème suivant. L'algorithme s'arrête quand une solution totalement entière est trouvée ou quand un critère d'arrêt est atteint.
- Algorithme OA [25] : à partir du problème de départ P non linéaire, on déduit un problème Q en linéarisant (au premier ordre) les contraintes de P. Ce problème Q est plus facile à résoudre et on sait obtenir la solution optimale mais pas forcément réalisable. En effet, certaines valeurs de la solution optimale sont continue et d'autres discrètes. À partir de cette solution, un nouveau problème R est construit en fixant sur le problème P les variables dont on a trouvé une solution discrète et en optimisant sur les variables restantes. Cette résolution va permettre d'exhiber de nouvelles contraintes pour le problème Q et le processus sera réitérer tant que Q ne trouve pas une solution optimale entièrement discrète ou jusqu'à ce qu'un critère d'arrêt (un certains temps fixé par l'utilisateur) soit atteint. Le problème Q est un problème linéaire mixte (variables discrètes et continues), tandis que le problème R est un problème convexe non linéaire.
- Algorithme hybride : cet algorithme utilise à la fois l'algorithme BaB et l'algorithme OA. L'utilisation de l'approximation externe dans le parcours de l'arbre permet de limiter le nombre de nœuds de celui-ci, et la construction de sous-problèmes rend plus facile la résolution de l'approximation externe.

Pour plus de détails le lecteur pourra se référer à [9].

Dans le solveur Bonmin, les instances de problèmes sont décrites dans le langage de description `AMPL`[5], qui est un language largement utilisé dans la plupart des solveurs de problèmes d'optimisation.

5.5 Exemples

La méthode pour l'implémentation matérielle et le choix des largeurs est appliquée dans cette section à notre exemple fil rouge (voir section 3.4), dans le cas de la forme directe I (pour illustrer le cas particulier de la section 5.4.1) et du state-space pour le cas général.

5. `http://ampl.com/`

5.5.1 Forme Directe I

Afin d'illustrer le cas particulier vu précédemment, reprenons ici la forme directe I de l'exemple fil rouge, dont on rappelle l'équation ici :

$$y(k) = \sum_{i=0}^{3} b_i u(k-i) - \sum_{i=1}^{3} a_i y(k-i) \tag{5.49}$$

avec

$$\begin{array}{rclrcl} b_0 & = & -0.602538, & a_1 & = & -0.317866 \\ b_1 & = & 0.0217266, & a_2 & = & 0.251000 \\ b_2 & = & -0.0965601, & a_3 & = & 0.0306691 \\ b_3 & = & -0.0514917, & & & \end{array} \tag{5.50}$$

(on rappelle que les valeurs données sont des valeurs approchées).

On rappelle également que l'entrée est donnée par $u(k) \in [-25; 30]$ et que, par la proposition 3.2, la sortie est donnée par $y(k) \in [-23.7637; 19.9824]$.

On choisit pour cet exemple d'avoir une borne sur l'erreur globale sur la sortie égale à $\xi = 2^{-3}$. Pour appliquer la formule donnée par l'équation (5.45), il faut d'abord calculer a et d d'après les équations (5.43) et (5.44), en utilisant les valeurs de l'équation (3.39) et $m_y = 5$ (puisque $y(k) \in [-25; 30]$ pour tout k) :

$$a = 1412.97, \quad d = 350.510. \tag{5.51}$$

On peut maintenant appliquer l'équation (5.45) et on obtient :

$$w_y = \lfloor \log_2(a) \rfloor + 1 = 10, \tag{5.52}$$

donc le signal y est au format $(5, -4)$.

Pour déterminer les largeurs des coefficients en appliquant les équations (5.46), il faut calculer les msb des coefficients et calculer également δ d'après la proposition 5.1. Les msb des coefficients sont donnés par :

$$\begin{array}{rclrcl} m_{b_0} & = & 0, & m_{a_1} & = & -1, \\ m_{b_1} & = & -5, & m_{a_2} & = & -1, \\ m_{b_2} & = & -3, & m_{a_3} & = & -5, \\ m_{b_3} & = & -4, & & & \end{array} \tag{5.53}$$

et d'après la proposition 5.1, on a $\delta = 3$. On applique alors les formules des équations (5.46), il vient :

$$\begin{array}{rclrcl} w_{b_0} & = & 14, & w_{a_1} & = & 13, \\ w_{b_1} & = & 9, & w_{a_2} & = & 13, \\ w_{b_2} & = & 11, & w_{a_3} & = & 9, \\ w_{b_3} & = & 10. & & & \end{array} \tag{5.54}$$

La largeur de l'entrée $u(k)$ est arbitraire (elle n'est pas soumise à une des contraintes), donc on peut la choisir, et on décide dans cet exemple d'affecter à $u(k)$ la même largeur que $y(k)$, on a donc $w_u = 10$.

On connait alors les formats des coefficients (on a les largeurs et les msb, on peut donc en déduire directement les formats), ainsi que ceux de l'entrée et de la sortie, on peut donc calculer les formats pour les produits et calculer la somme.

La figure 5.3 illustre la somme correspondant au calcul de $y(k)$. À chaque itération, une somme intermédiaire est calculée sur le format $(5, -7)$ avec les bits exacts des produits constantes/variables (les bits hachurés sur les produits représentent les bits tronqués par le formatage mais également les bits potentiellement faux de chaque produit), et le résultat est ensuite arrondi sur le format final $(5, -4)$ garantissant une erreur globale sur la sortie inférieure à $\xi = 2^{-3}$.

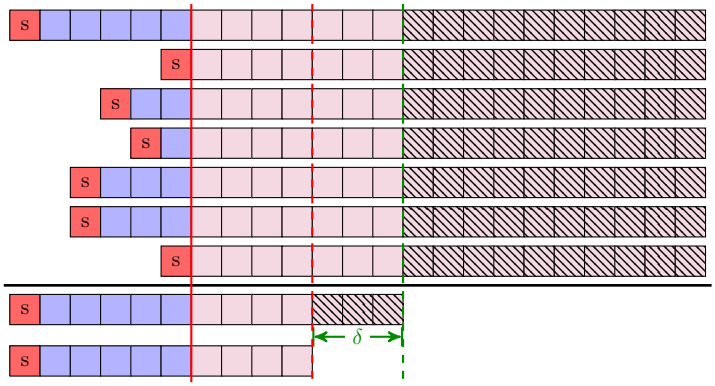


Figure 5.3 – Illustration sous forme d'une somme de la résolution du problème des largeurs minimales dans le cas d'une DFI.

La figure 5.4 illustre la solution trouvée sous forme d'un oSoP de hauteur minimale, puisque tous les oSoP produisent la même dégradation numériques à chaque itération.

5.5.2 State-Space

Cet exemple vise à illustrer, dans le cas général, la résolution de notre problème d'optimisation. On reprend alors notre exemple fil rouge exprimé par la réalisation state-space (voir équation (3.44)).

Comme dans le cas précédent, on rappelle que l'entrée est donnée par $u(k) \in [-25; 30]$ et que, par la proposition 3.2, la sortie est donnée par $y(k) \in [-23.7637; 19.9824]$. De plus, on souhaite toujours avoir une erreur globale sur la sortie inférieure à $\xi = 2^{-3}$.

Le problème d'optimisation, exprimé par les équations (5.38), correspond pour cet exemple au problème suivant :

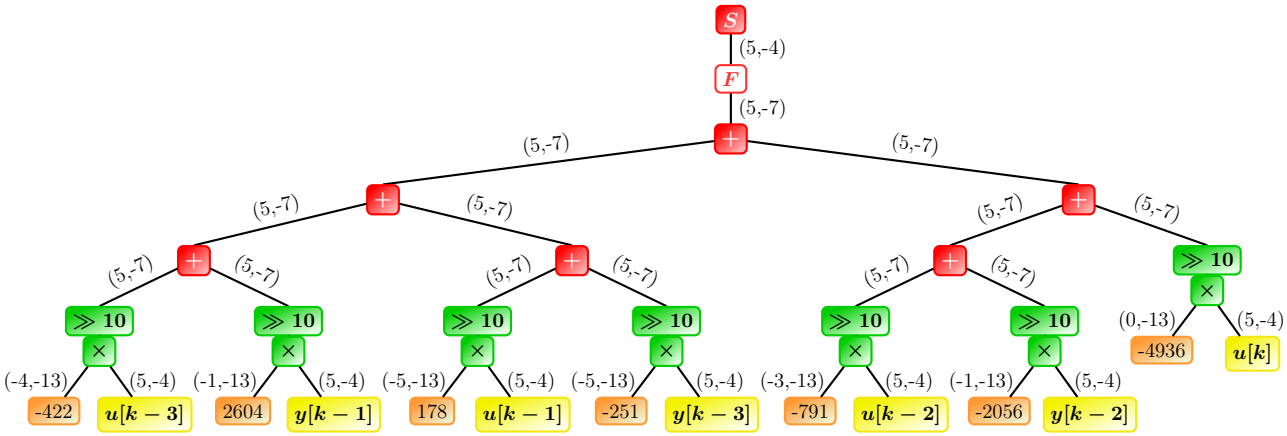


FIGURE 5.4 – Illustration sous forme d'un oSoP de la résolution du problème des largeurs minimales dans le cas d'une DFI.

$$\begin{aligned}
\min \quad & f(w_{\boldsymbol{x}_1}, w_{\boldsymbol{x}_2}, w_{\boldsymbol{x}_3}, w_{\boldsymbol{y}}) = w_{\boldsymbol{x}_1} + w_{\boldsymbol{x}_2} + w_{\boldsymbol{x}_3} + w_y && (5.55a)\\
\text{tel que} \quad & \frac{\boldsymbol{a}_1}{2^{w_{\boldsymbol{x}_1}}} + \frac{\boldsymbol{a}_2}{2^{w_{\boldsymbol{x}_2}}} + \frac{\boldsymbol{a}_3}{2^{w_{\boldsymbol{x}_3}}} + \frac{\boldsymbol{a}_4}{2^{w_y}} < 1 && (5.55b)\\
& \frac{\boldsymbol{d}_1}{2^{w_{\boldsymbol{x}_1}}} + \frac{\boldsymbol{d}_2}{2^{w_{\boldsymbol{x}_2}}} + \frac{\boldsymbol{d}_3}{2^{w_{\boldsymbol{x}_3}}} + \frac{\boldsymbol{d}_4}{2^{w_y}} < 1 && (5.55c)
\end{aligned}$$

avec $\boldsymbol{a}$ et $\boldsymbol{d}$ déterminés par les équations (5.28) en utilisant les valeurs données par les équations (3.49) et (3.50) :

$$\begin{aligned}
\boldsymbol{a} &= \begin{pmatrix} 4427.20 & 112053. & 158199. & 1024.0 \end{pmatrix} && (5.56)\\
\boldsymbol{d} &= \begin{pmatrix} 455.189 & 15193.0 & 21724.0 & 0.0 \end{pmatrix} && (5.57)
\end{aligned}$$

Le solveur Bonmin nous donne la solution optimale suivante pour les largeurs :

$$w_{\boldsymbol{x}_1} = 14, \quad w_{\boldsymbol{x}_2} = 20, \quad w_{\boldsymbol{x}_3} = 19, \quad w_y = 12. \tag{5.58}$$

À titre indicatif, la largeur commune pour les quatre variables, calculée d'après l'équation (5.48), est $w' = 19$.

On peut alors calculer les largeurs des coefficients d'après les largeurs des variables, en utilisant l'équation (5.37). Pour cela, on rappelle que dans cet exemple on calcule 4 SoP, et on doit donc calculer au préalable $\delta_{\boldsymbol{v}'_i}$ pour $1 \leqslant i \leqslant 4$. Dans notre exemple, on a, d'après la proposition 5.1 :

$$\delta_{\boldsymbol{v}'_i} = \lfloor \log_2(3) \rfloor + 1 = 2, \quad \forall 1 \leqslant i \leqslant 4 \tag{5.59}$$

On note $\boldsymbol{Z}$ la matrice correspondant à la SIF de notre spate-space, c'est-à-dire $\boldsymbol{Z} \triangleq \begin{pmatrix} \boldsymbol{A} & \boldsymbol{b} \\ \boldsymbol{c} & d \end{pmatrix}$, et on note également $\boldsymbol{W}_{\boldsymbol{Z}}$ la matrice des largeurs des

coefficients de $\boldsymbol{Z}$, c.-à-d. $(\boldsymbol{W_Z})_{i,j} \triangleq w_{\boldsymbol{Z}_{i,j}}$ pour $1 \leqslant i, j \leqslant 4$. On a alors, par application de la formule de l'équation (5.37) :

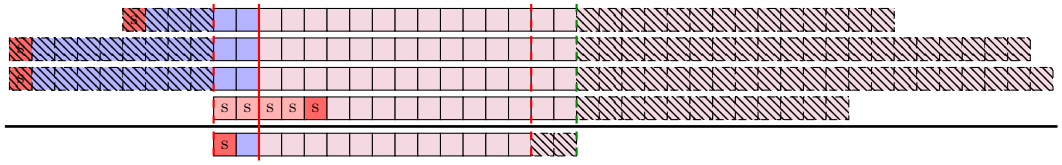
$$\boldsymbol{W_Z} = \left(\begin{array}{ccc:c} 22 & 27 & 27 & 13 \\ 28 & 33 & 33 & 16 \\ 26 & 31 & 32 & 15 \\ \hdashline 17 & 22 & 22 & 15 \end{array}\right) \tag{5.60}$$

Comme pour l'exemple précédent, on choisit d'affecter à w_u la valeur de la largeur de sortie, c'est-à-dire $w_u = w_y = 12$.

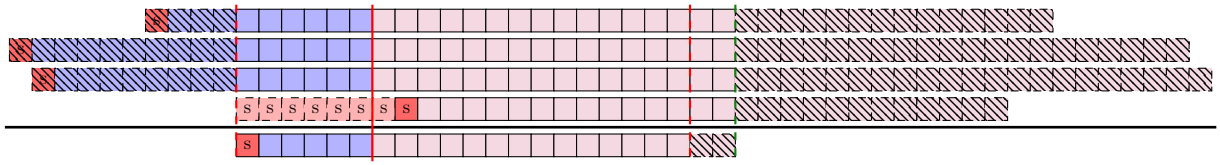
On calcule ensuite les formats des produits des différents SoP à partir des équations (2.42) et (2.43). La figure 5.5 illustre les représentations binaire des quatre sommes de produits. On peut constater que certains produits nécessitent un nombre important de bits, notamment sur la figure 5.5b illustrant le calcul de $x_2(k)$ où deux produits nécessitent 52 bits. Cependant, l'application de nos règles de formatage nous permettent de ne pas calculer le produits sur l'ensemble des bits. En effet, les bits en positions plus petites que le lsb imposé par l'opération de formatage (le lsb du résultat moins les δ bits de garde déterminés par la proposition 5.1) n'ont pas à être stockés, seule la retenue qu'ils impliquent sur les bits significatifs est nécessaire. De plus, les bits dont la position est plus grande que le msb du résultat n'ont pas à être calculés d'après la règle décrite dans la section 3.3.2.

La difficulté supplémentaire lors de la résolution du problème pour un state-space par rapport à une DFI est qu'ici la largeur d'un état est utilisée dans les expressions des contraintes des largeurs des autres états, et vice-versa. Par conséquent, on ne peut pas résoudre le problème pour SoP du chaque state-space, étant donné que les différentes contraintes dépendent les unes des autres.

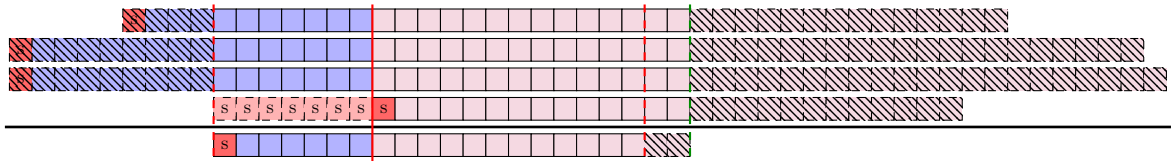
Afin de comparer ce résultat avec un code flottant double précision, un code est généré en `C++` en utilisant la librairie `ac_fixed`. Ce choix est justifié par l'utilisation de largeurs différentes pour chaque variable et chaque coefficient, la librairie permettant de définir des variables de largeur arbitraire. Afin de cibler la comparaison sur les erreurs de calculs (et non les erreurs de quantification des coefficients), les coefficients utilisés dans le code flottant double précision sont les mêmes coefficients que dans le code virgule fixe, avec les mêmes largeurs respectives. Le signal d'entrée u est tiré aléatoirement dans l'intervalle $[-25; 30]$. La figure 5.6 représente la différence $\delta y(k) = y^*(k) - y(k)$ pour $0 \leqslant k \leqslant 10000$. Les droites vertes illustrent les bornes attendues sur l'erreur (on souhaitait dans cet exemple borner l'erreur en valeur absolue par 2^{-3}), et les droites rouges illustrent les bornes théoriques sur l'erreur, calculées par le corollaire 3.1 à partir des valeurs déduites du résultat de l'optimisation. L'intervalle théorique de l'erreur (illustré par les droites rouges) est $[-0.0768329; 0.00575609]$. Si le code est loin d'atteindre les bornes attendues par l'utilisateur, c'est parce que la détermination des contraintes (5.28) est pessimiste par rapport à l'évaluation théorique de l'erreur, et l'évaluation théorique de l'erreur est elle-même



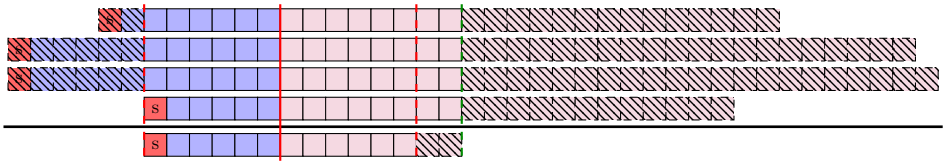
(a) Calcul de $x_1(k)$ sur le format intermédiaire $(1, -14)$.



(b) Calcul de $x_2(k)$ sur le format intermédiaire $(5, -16)$.



(c) Calcul de $x_3(k)$ sur le format intermédiaire $(6, -14)$.



(d) Calcul de $y(k)$ sur le format intermédiaire $(5, -8)$.

FIGURE 5.5 – Représentation des sommes avec les formats obtenus d'après la résolution du problème d'optimisation.

pessimiste (elle dépend du wors-case-peak-gain qui, même s'il peut être atteint, est généralement pessimiste).

5.6 Conclusion

Nous avons dressé dans ce chapitre un état de l'art autour de l'optimisation des largeurs en virgule fixe, un enjeu majeur dans le domaine de l'arithmétique virgule fixe de ce début de siècle, quand une implémentation matérielle est considérée. Dans la suite, nous avons proposé une nouvelle méthodologie, utilisant une méthode de formatage de bits permettant de limiter le nombre de bits impliqués dans le calcul d'une somme de plusieurs termes si l'on s'autorise un arrondi fidèle sur le résultat final.

Cette méthodologie repose également sur l'analyse de l'erreur utilisant le WCPG vu dans le chapitre 3, afin de pouvoir déterminer les largeurs pour une

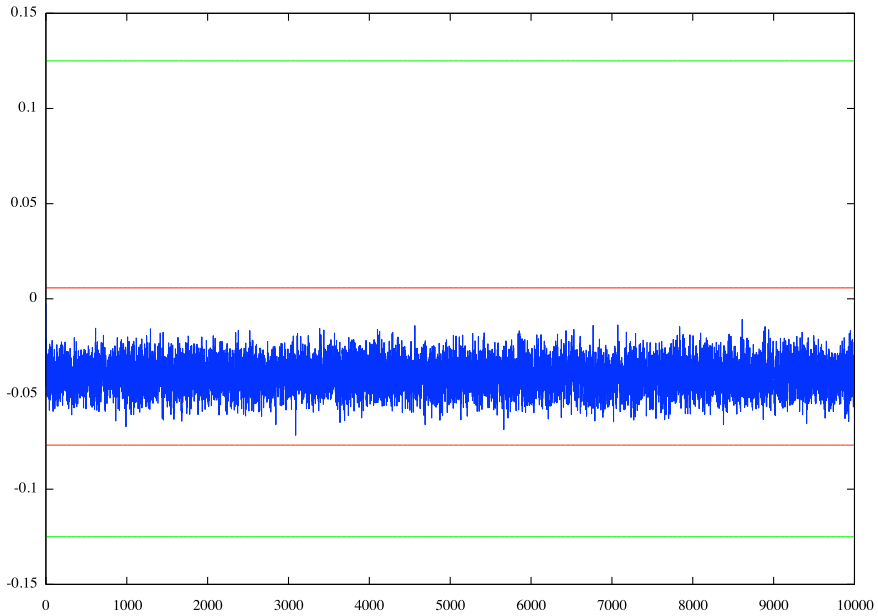


Figure 5.6 – Différence entre la sortie y flottante double précision et la sortie virgule fixe du code généré à partir des résultats de l'optimisation. Les droites vertes (extérieures) représentent les bornes attendues sur l'erreur, et les droites rouges (intérieures) illustrent les bornes théoriques sur l'erreur.

borne donnée sur l'erreur finale.

Le solveur Bonmin est utilisé pour résoudre le problème d'optimisation ainsi formalisé et il a été observé que, dans le cas où l'algorithme du filtre n'implique qu'un seul produit scalaire ou dans le cas où toutes les largeurs sont les mêmes, le problème se simplifie et on peut directement obtenir les largeurs souhaitées.

Chapitre 6

Exemple complet du flot `FiPoGen`

"It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiment, it's wrong."
- Richard P. Feynman

Depuis le début de cette thèse, un outil programmé en Python est en développement, appelé `FiPoGen` (pour **Fi**xed-**Po**int code **Gen**erator). Le but de cet outil est de répondre aux méthodologies décrites dans les chapitres précédents (pour une implémentation logicielle ou une implémentation matérielle) et ainsi de fournir automatiquement le meilleur code possible selon les critères définis. Cet outil devrait être disponible prochainement en *open-source* et également accessible via une interface web.

Après une description des méthodes employées par `FiPoGen` dans le cas logiciel et dans le cas matériel, un exemple sera introduit afin d'illustrer par la suite les capacités de l'outil.

6.1 Méthodologie `FiPoGen`

Nous avons proposé, dans les chapitres 4 et 5, une méthodologie pour répondre au problème de l'implémentation en virgule fixe dans le cas logiciel et le cas matériel, respectivement.

L'outil `FiPoGen` propose une implémentation qui reprend presque point par point cette méthodologie pour les deux types de cibles visées. On se propose dans cette section de décrire, non pas la méthodologie d'implémentation, déjà décrite dans les chapitres précédemment cités, mais l'implémentation faite dans `FiPoGen` de cette méthodologie.

6.1.1 Solution logicielle

La méthode d'implémentation pour un produit scalaire est résumée dans la section 4.5. La solution développée dans FiPoGen pour répondre à cette méthode est illustrée par la figure 6.1.

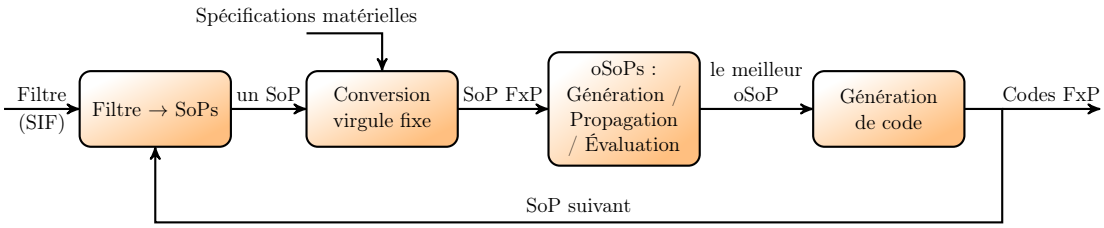


Figure 6.1 – Schéma du flot FiPoGen pour l'implémentation logcielle.

Les différentes étapes sont décrites ci-dessous :
- l'algorithme du filtre est décomposé en une succession de produits scalaires décrivant le calcul ;
- pour un SoP et en utilisant les spécifications matérielles (largeurs des opérateurs, des constantes, des variables, DC-gain et WCPG du filtre pour déterminer les formats FxP des variables, etc.), une conversion FxP est effectuée qui détermine les coefficients FxP mais aussi les formats des variables ;
- à partir de ce SoP virgule fixe, les oSoPs, sont générés et le meilleur oSoP (selon les critères définis) est retourné. Dans FiPoGen, les étapes de génération des oSoPs, de propagation des intervalles, d'évaluation des erreurs et de choix, décrites dans la section 4.5, sont effectuées simultanément (la propagation et l'évaluation s'effectuant sur chaque sous-oSoP lors de la génération) ;
- une fois le meilleur oSoP déterminé et tous les formats intermédiaires du calcul spécifiés, on génère le code correspondant à cet oSoP ;
- l'opération est répétée pour chaque SoP de l'algorithme.

6.1.2 Solution matérielle

De la même manière que pour la solution logicielle, le schéma de la solution matérielle développée dans FiPoGen pour répondre à la problématique décrite dans le chapitre 5 est illustré par la figure 6.2.

Les différentes étapes sont décrites ci-dessous :
- la première étape est la même que dans le cas logiciel, l'algorithme du filtre est décomposé en une succession de SoPs décrivant le calcul, sauf qu'ici l'ensemble des SoPs est considéré dans les étapes suivantes ;
- deux voies sont possibles suivant le nombre de SoPs dans l'algorithme :

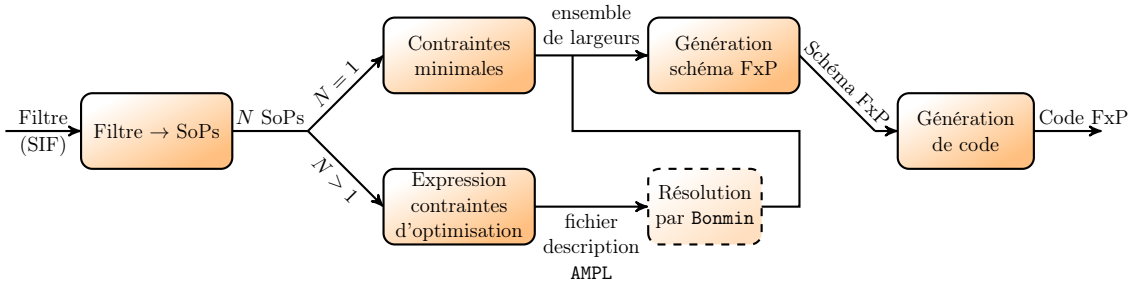


FIGURE 6.2 – Schéma du flot `FiPoGen` pour l'implémentation matérielle.

  - s'il n'y a qu'un seul SoP, on considère pour les variables et les coefficients les bornes minimales des contraintes (voir 5.4.1) ;
  - s'il y a plusieurs SoPs dans l'algorithme : les contraintes sont exprimées pour l'ensemble des variables des SoPs et une description de ces contraintes dans le langage `AMPL` est générée. À partir de cette description le solveur `Bonmin` résout le problème et fourni une solution optimale (ou sous-optimale si l'optimal ne peut pas être atteint) pour l'ensemble des largeurs ;
- à partir des largeurs (déterminées par les bornes minimales s'il y a un SoP ou par `Bonmin` s'il y en a plusieurs), un schéma FxP est généré. On rappelle ici que, puisque le formatage de bits est appliqué, tous les schémas possibles commentent les mêmes erreurs de calculs (voir proposition 5.2), et donc le critère utilisé par `FiPoGen` est celui du plus grand parallélisme du schéma ;
- le code FxP de cet oSoP est finalement généré.

6.2 Description de l'exemple

Le but de cet exemple est dans un premier temps d'illustrer l'ensemble de la méthodologie mise en place durant cette thèse. Cet exemple aura également pour but de montrer les performances de l'outil, en indiquant les temps d'exécution, les erreurs sur les résultats finaux, etc. Les avantages de notre approche globale, du choix de la bonne réalisation pour un filtre jusqu'à la génération de code, seront également mis en avant en considérant, pour un même filtre, tout un panel de réalisations différentes.

Pour cette étude, un filtre LTI SISO (afin de pouvoir considérer les réalisations ρDFIIt et LGS) aléatoire d'ordre 8, $\mathcal{H}$, a été généré avec `Matlab` par la commande `drss`, déjà utilisée pour l'exemple fil rouge du manuscrit.

La fonction de transfert H du filtre $\mathcal{H}$ est la suivante :

$$H(z) = \frac{\sum_{i=0}^{8} \boldsymbol{b}_i z^{-i}}{1 + \sum_{i=0}^{8} \boldsymbol{a}_i z^{-i}}, \quad \forall z \in \mathbb{C}, \tag{6.1}$$

avec

$$\begin{array}{lllllll}
\boldsymbol{b}_0 & = & -0.940140, & & & \\
\boldsymbol{b}_1 & = & 1.04326, & \boldsymbol{a}_1 & = & -2.06925, \\
\boldsymbol{b}_2 & = & 0.292108, & \boldsymbol{a}_2 & = & 1.95404, \\
\boldsymbol{b}_3 & = & -0.693159, & \boldsymbol{a}_3 & = & -1.04616, \\
\boldsymbol{b}_4 & = & 0.183105, & \boldsymbol{a}_4 & = & 0.0898125, \\
\boldsymbol{b}_5 & = & 0.270429, & \boldsymbol{a}_5 & = & 0.172354, \\
\boldsymbol{b}_6 & = & -0.296554, & \boldsymbol{a}_6 & = & -0.0730519, \\
\boldsymbol{b}_7 & = & -0.0311837, & \boldsymbol{a}_7 & = & -0.00696194, \\
\boldsymbol{b}_8 & = & 0.0363021, & \boldsymbol{a}_8 & = & 0.00556424.
\end{array} \tag{6.2}$$

Afin d'illustrer l'intérêt de choisir une bonne réalisation, l'étude est réalisée sur plusieurs réalisations. Ces réalisations sont brièvement décrites une à une et les matrices de leurs coefficients sont précisées.

Pour les state-space, les matrices de coefficients sont données sous la forme $\left(\begin{array}{c|c} \boldsymbol{A} & \boldsymbol{b} \\ \hline \boldsymbol{c} & d \end{array}\right)$, où $\boldsymbol{A}$ est une matrice de taille 8×8. De la même manière, pour les SIF, les matrices de coefficients sont données sous la forme $\left(\begin{array}{c|c|c} -\boldsymbol{J} & \boldsymbol{M} & \boldsymbol{N} \\ \hline \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{Q} \\ \hline \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{S} \end{array}\right)$, où $\boldsymbol{S}$ est un scalaire, $\boldsymbol{P}$ est de taille 8×8, et $\boldsymbol{J}$ est de taille variable selon la réalisation considérée.

Forme directe I (DFI) : Cette réalisation est obtenue à partir de l'équation classique d'une DFI (équation (1.14)) avec $n = 8$ et les coefficients de l'équation (6.2). Elle consiste en un SoP composé de 17 produits.

ρDFIIt ($SIF\rho$) : On considère également une Forme directe II transposée avec opérateur ρ (ρDFIIt), représentée sous forme de SIF, dont la description est donnée dans [41]. La matrice $\boldsymbol{Z}_\rho$ des coefficients de cette SIF est donnée par l'équation (6.3). On peut constater que la matrice, de taille 17×17, est très creuse, et implique peu de calculs : 40 produits et 24 additions, répartis en 17 SoP (le plus gros impliquant 4 produits et 3 additions).

$$\boldsymbol{Z}_\rho = \left(\begin{array}{cccccccc:cccccccc:c}
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.32727 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.94014 \\
0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.737549 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.718005 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.883582 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.719537 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.763098 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.761275 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.524466 & 0 \\
\hdashline
-0.197359 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0.249693 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.865237 \\
-0.271834 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0.294632 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0012346 \\
-0.217349 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0.18816 & 0 & 0 & 0 & 0 & 0 & 0.248451 \\
0.257446 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0.201781 & 0 & 0 & 0 & 0 & -0.0697342 \\
0.125632 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0.202961 & 0 & 0 & 0 & 0.449446 \\
0.0861154 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0.214659 & 0 & 0 & -0.0408021 \\
0.13416 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0.192965 & 0 & -0.481683 \\
0.132208 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.786348 & -0.388698 \\
\hdashline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right) \tag{6.3}$$

$$\boldsymbol{Z}_{eq} = \left(\begin{array}{cccccccc:c}
0.954393 & -0.160211 & -0.0196964 & 0.0619611 & -0.000592975 & 0.0079572 & -0.000768022 & 5.00511e-05 & -0.425856 \\
-0.160211 & 0.087364 & 0.578595 & 0.311007 & 0.0214818 & 0.0204804 & -0.00465003 & 0.000286284 & -0.936389 \\
0.0196964 & -0.578595 & 0.677352 & -0.23732 & -0.00300623 & -0.0252948 & 0.00300391 & -0.000192204 & 0.356125 \\
-0.0619611 & -0.311007 & -0.23732 & 0.0526454 & 0.434231 & -0.302408 & -0.0137928 & 0.000639639 & -0.168048 \\
-0.000592975 & 0.0214818 & 0.00300623 & -0.434231 & 0.512455 & 0.571766 & -0.035875 & 0.00231919 & -0.0295764 \\
-0.0079572 & -0.0204804 & -0.0252948 & -0.302408 & -0.571766 & -0.195775 & -0.0985609 & 0.00580496 & -0.0430786 \\
-0.000768022 & -0.00465003 & -0.00300391 & 0.0137928 & -0.035875 & 0.0985609 & -0.200501 & -0.309185 & -0.00107917 \\
5.00511e-05 & 0.000286284 & 0.000192204 & -0.000639639 & 0.00231919 & -0.00580496 & -0.309185 & 0.181319 & 8.96175e-05 \\
\hdashline
0.425856 & 0.936389 & 0.356125 & -0.168048 & 0.0295764 & -0.0430786 & 0.00107917 & -8.96175e-05 & -0.940140
\end{array}\right) \tag{6.4}$$

State-space équilibré ($SSeq$) : Le premier state-space considéré est un state-space équilibré (voir définition 1.12), dont la matrice des coefficients, notée $\boldsymbol{Z}_{eq}$, est donnée par l'équation (6.4).

Les prochains state-space sont obtenus en re-paramétrant celui-ci, c.-à-d. en lui appliquant une transformation $\boldsymbol{T}$.

Remarque 6.1. *Tous les state-space considérés ici sont pleinement paramétrés (aucun coefficient nul) et possède ainsi le même nombre de calculs : 9 SoP de taille 9 (9 produits et 8 additions chacun), soit un total de 81 produits et 72 additions.*

State-space aléatoire ($SSal$) : Cette réalisation est obtenue à partir de $SSeq$ en lui appliquant une transformation aléatoire, dans le but d'obtenir un state-space quelconque. On a la matrice de coefficients suivante :

$$\boldsymbol{Z}_{al} = \left(\begin{array}{cccccccc:c}
-0.0632406 & 0.8939 & 3.0728 & -1.61614 & 0.178363 & -0.845585 & 0.765506 & -1.87008 & -0.286985 \\
3.55187 & 5.47809 & 0.998207 & 4.98081 & 2.48082 & 4.89826 & 2.47933 & 6.43144 & 0.782005 \\
-0.169653 & -2.09311 & -1.1035 & -1.20609 & -1.82609 & -3.03564 & -2.00589 & -2.07307 & -0.449679 \\
-4.2914 & -5.42848 & -4.92528 & -2.77789 & -0.656006 & -0.757243 & -2.47233 & -2.31986 & 0.506792 \\
-13.7971 & -15.3266 & -9.05665 & -12.8906 & -2.12159 & -5.431 & -6.06198 & -11.7924 & 0.753966 \\
4.44396 & 6.71614 & 5.02911 & 4.33814 & 1.93579 & 3.34264 & 3.90283 & 4.21117 & 0.085371 \\
2.11671 & 2.72764 & 2.65081 & 1.51698 & 0.156384 & 0.424597 & 1.22339 & 1.17236 & -0.518907 \\
0.43289 & -1.64873 & -0.433148 & -0.563853 & -1.93156 & -3.03834 & -1.26397 & -1.90865 & -0.952559 \\
\hdashline
9.5539 & 10.424 & 3.17811 & 11.317 & 2.29594 & 6.62574 & 4.5831 & 11.0614 & -0.940140
\end{array}\right) \tag{6.5}$$

State-space $\mathcal{L}_2$ ($SS_{\mathcal{L}_2}$) : Cette re-paramétrisation du state-space $SSeq$ minimise la sensibilité à la norme $\mathcal{L}_2$. On a la matrice des coefficients $\boldsymbol{Z}_{\mathcal{L}_2}$ suivante :

$$\boldsymbol{Z}_{\mathcal{L}_2} = \left(\begin{array}{cccccccc:c}
0.467458 & 0.310338 & 0.197121 & -0.735333 & 0.0278436 & -0.00672839 & -0.067344 & -0.664021 & -0.00936502 \\
-0.240399 & 0.16769 & -0.265197 & 0.132307 & 0.793924 & -0.105099 & 0.0626461 & 0.193218 & -0.106941 \\
-0.223757 & -0.313438 & -0.262534 & 0.441753 & -0.360888 & 0.0713033 & 0.223862 & 0.0377591 & -0.0582334 \\
0.000539906 & -0.546026 & -0.309396 & 0.596688 & 0.142015 & 0.188554 & 0.0215739 & 0.082093 & -0.176416 \\
-0.38521 & -0.452078 & -0.574424 & -0.0800689 & 0.312343 & 0.391252 & -0.207539 & -0.20437 & 0.0547321 \\
-0.139317 & 0.417174 & 0.43892 & -0.369843 & 0.0856432 & 0.42537 & 0.456983 & -0.160648 & -0.0595954 \\
-0.100473 & 0.0299304 & 0.327077 & -0.144329 & -0.0349994 & 0.472064 & 0.12982 & 0.155027 & -0.114077 \\
0.326336 & -0.259867 & 0.265129 & 0.0149514 & 0.0329415 & -0.0883162 & 0.57605 & 0.232416 & -0.215465 \\
\hdashline
2.1134 & 2.88476 & -0.317584 & 1.36126 & 0.385345 & 1.61323 & -0.785983 & 1.70236 & -0.940140
\end{array}\right) \tag{6.6}$$

State-space ε_H ($SS\varepsilon_H$) : On minimise maintenant l'erreur sur la fonction de transfert. On a la matrice des coefficients $\boldsymbol{Z}_{\varepsilon_H}$ suivante :

$$\boldsymbol{Z}_{\varepsilon_H} = \left(\begin{array}{cccccccc:c}
-0.667553 & -0.678978 & -0.277133 & -1.395 & -0.129737 & -0.392609 & -0.476533 & -1.35659 & -0.00685491 \\
0.246859 & 0.579478 & -0.0754829 & 0.611096 & 0.0751157 & -0.214719 & -0.304638 & 0.516869 & 0.00646396 \\
0.405165 & 0.634561 & -0.242565 & 0.332917 & -0.407238 & -0.662181 & -0.578579 & -0.381738 & -0.0476695 \\
1 & 0.706014 & 0.498726 & 1.16488 & 0.0293219 & 0.483624 & 0.435732 & 0.928656 & -0.126046 \\
-4.80614 & -4.60383 & -1.91791 & -5.0454 & -0.0593067 & -1.35512 & -1.19145 & -4.50814 & 0.252832 \\
1.56596 & 2.38678 & 0.678404 & 1.9673 & 0.45611 & 1.26446 & 1.08701 & 1.23325 & 0.0419036 \\
-0.476016 & -0.424787 & -0.0151908 & -0.355471 & -0.188894 & 0.0926871 & -0.0326742 & 0.121052 & -0.115213 \\
-0.578343 & -1.10723 & -0.125448 & -0.585721 & 0.104804 & -0.0514263 & 0.214419 & 0.0625335 & 0.0273167 \\
\hdashline
9.53727 & 10.788 & 2.81306 & 11.3199 & 2.36002 & 6.83293 & 4.51911 & 10.6925 & -0.940140
\end{array}\right) \tag{6.7}$$

State-space ε_p ($SS\varepsilon_p$) : Cette réalisation minimise l'erreur sur les pôles du filtre. On a la matrice des coefficients $\boldsymbol{Z}_{\varepsilon_p}$ suivante :

$$\boldsymbol{Z}_{\varepsilon_p} = \left(\begin{array}{cccccccc:c}
-0.267075 & -0.542593 & -0.128242 & -0.875133 & -0.411657 & -0.597916 & -0.556239 & -0.730887 & -0.0869271 \\
-0.620391 & -0.109652 & -0.292396 & -0.373419 & 0.310999 & -0.235429 & -0.223878 & -0.164808 & 0.0842079 \\
-0.151488 & 0.587543 & 0.242128 & 0.234363 & -0.863583 & -1.02798 & -0.290766 & -0.0648477 & -0.0482105 \\
1.22583 & 0.832452 & 0.48337 & 1.49561 & 0.241016 & 0.691114 & 0.425144 & 1.13851 & -0.110295 \\
-2.03562 & -2.19214 & -1.18198 & -2.15955 & -0.00390481 & -0.872703 & -0.82956 & -1.84989 & 0.0891084 \\
0.913739 & 1.44872 & 0.58143 & 1.26123 & 0.429005 & 1.17488 & 0.932959 & 0.752039 & 0.0741525 \\
-1 & -0.348655 & 0.105839 & -0.401638 & -0.654041 & -0.465387 & 0.0449626 & 0.224698 & -0.148856 \\
0.0372506 & -0.707618 & -0.404017 & -0.589764 & 0.35874 & 0.347269 & 0.0534274 & -0.507698 & 0.032588 \\
\hdashline
9.55084 & 10.5361 & 2.76477 & 11.2003 & 2.36008 & 6.98261 & 4.47201 & 10.6515 & -0.940140
\end{array}\right) \tag{6.8}$$

State-space ε_{Hp} ($SS\varepsilon_{Hp}$) : La dernière re-paramétrisation que l'on considère ici est un compromis entre la minimisation de l'erreur sur la fonction de transfert et de l'erreur sur les pôles (voir équation (98) de [41]). On obtient ainsi la matrice $\boldsymbol{Z}_{\varepsilon_{Hp}}$ suivante :

$$\boldsymbol{Z}_{\varepsilon_{Hp}} = \left(\begin{array}{cccccccc:c}
-0.929397 & -1.14874 & -0.379761 & -1.65697 & -0.308237 & -0.597498 & -0.660425 & -1.55863 & -0.0145945 \\
-0.0726404 & 0.464209 & -0.0758013 & 0.3568 & 0.379573 & -0.0853479 & -0.119665 & 0.455788 & 0.0515328 \\
0.491203 & 0.70549 & -0.0298367 & 0.488557 & -0.533858 & -0.782556 & -0.529685 & -0.174433 & -0.0604004 \\
1.28886 & 0.949109 & 0.517961 & 1.45105 & -0.0457192 & 0.487466 & 0.391709 & 1.10436 & -0.146328 \\
-3.05212 & -2.97141 & -1.37129 & -3.19486 & -0.0864633 & -1.02892 & -0.987039 & -2.93595 & 0.151564 \\
0.655526 & 1.26301 & 0.41576 & 0.978725 & 0.354417 & 0.979585 & 0.844913 & 0.425266 & 0.0836914 \\
-0.636365 & -0.455317 & -0.0255871 & -0.408501 & -0.225859 & 0.0138523 & -0.00654007 & 0.119926 & -0.117447 \\
-0.0734973 & -0.5 & -0.0741442 & -0.205418 & 0.188245 & 0.200947 & 0.31811 & 0.226649 & 0.0113756 \\
\hdashline
9.52823 & 10.6977 & 2.89418 & 11.3407 & 2.34548 & 6.81605 & 4.49271 & 10.6935 & -0.940140
\end{array}\right) \tag{6.9}$$

Structure LGS (SIF_{LGS}) : La matrice $\boldsymbol{Z}_{LGS}$ des coefficients de la SIF de la structure LGS est donnée par l'équation (1.60). Cette matrice, très creuse, est de taille 185×185 et n'est donc pas représentable ici. On préfère indiquer les matrices non triviales, et on rappelle pour ça que la matrice $\boldsymbol{U}(i, j, x)$ est

définie comme la matrice identité sauf l'élément en position (i, j) qui vaut x (voir équation (A.17) en annexe).

$$\begin{array}{lllllllll}
\boldsymbol{A}^{(1)} & = & \boldsymbol{U}(2,1,-0.0859379), & \boldsymbol{A}^{(8)} & = & \boldsymbol{U}(5,5,0.831215), & \boldsymbol{A}^{(15)} & = & \boldsymbol{U}(7,8,1.47875), \\
\boldsymbol{A}^{(2)} & = & \boldsymbol{U}(2,2,0.992669), & \boldsymbol{A}^{(9)} & = & \boldsymbol{U}(6,5,-0.648844), & \boldsymbol{A}^{(16)} & = & \boldsymbol{U}(6,7,0.839682), \\
\boldsymbol{A}^{(3)} & = & \boldsymbol{U}(3,1,-0.346854), & \boldsymbol{A}^{(10)} & = & \boldsymbol{U}(6,6,0.740774), & \boldsymbol{A}^{(17)} & = & \boldsymbol{U}(5,6,0.539329), \\
\boldsymbol{A}^{(4)} & = & \boldsymbol{U}(3,3,0.893315), & \boldsymbol{A}^{(11)} & = & \boldsymbol{U}(7,6,-1.13352), & \boldsymbol{A}^{(18)} & = & \boldsymbol{U}(4,50.419595), \\
\boldsymbol{A}^{(5)} & = & \boldsymbol{U}(4,3,-0.414314), & \boldsymbol{A}^{(12)} & = & \boldsymbol{U}(7,7,0.512348), & \boldsymbol{A}^{(19)} & = & \boldsymbol{U}(3,4,0.370113), \\
\boldsymbol{A}^{(6)} & = & \boldsymbol{U}(4,4,0.867045), & \boldsymbol{A}^{(13)} & = & \boldsymbol{U}(8,7,-2.88622), & \boldsymbol{A}^{(20)} & = & \boldsymbol{U}(2,3,0.344311), \\
\boldsymbol{A}^{(7)} & = & \boldsymbol{U}(5,4,-0.483937), & \boldsymbol{A}^{(14)} & = & \boldsymbol{U}(8,8,0.095441), & \boldsymbol{A}^{(21)} & = & \boldsymbol{U}(1,2,0.0859379),
\end{array} \tag{6.10a}$$

$$\boldsymbol{A}^{(22)} = \begin{pmatrix}
1 & 0.0859379 & 0 & 0 & 0 & 0 & 0 & 0 \\
-0.0859379 & 1 & 0.346854 & 0 & 0 & 0 & 0 & 0 \\
0 & -0.346854 & 1 & 0.414314 & 0 & 0 & 0 & 0 \\
0 & 0 & -0.414314 & 1 & 0.483937 & 0 & 0 & 0 \\
0 & 0 & 0 & -0.483937 & 1 & 0.648844 & 0 & 0 \\
0 & 0 & 0 & 0 & -0.648844 & 1 & 1.13352 & 0 \\
0 & 0 & 0 & 0 & 0 & -1.13352 & 1 & 2.88622 \\
0 & 0 & 0 & 0 & 0 & 0 & -2.88622 & -4.20969
\end{pmatrix}, \tag{6.10b}$$

$$\boldsymbol{b}_{in} = \begin{pmatrix} 0.00134071 & 0.0156009 & 0.0453105 & 0.122423 & 0.291765 & 0.540979 & 0.644266 & 0.435683 \end{pmatrix}^{\top}, \tag{6.10c}$$

$$\boldsymbol{c}_{in} = \begin{pmatrix} -0.619531 & 0.305563 & 0.360364 & 0.381527 & -0.209683 & -0.713876 & -0.596989 & -0.314716 \end{pmatrix}, \tag{6.10d}$$

$$d = -0.940140. \tag{6.10e}$$

Entrée et sortie : Le signal d'entrée u est considéré comme une variable aléatoire uniformément répartie dans l'intervalle $[-237; 162]$. En appliquant la proposition 3.2 sur cet intervalle d'entrée, on obtient pour le signal de sortie y l'intervalle $[-1235.97; 1622.63]$, et ce pour toutes les réalisations (puisque toutes les réalisations ont la même fonction de transfert, et donc les mêmes DC-gain et WCPG).

Par conséquent, le msb de u et y sont $m_u = 8$ et $m_y = 11$, respectivement.

6.3 Implémentation logicielle

Le choix fait sur les largeurs pour l'implémentation logicielle est de 16 bits pour les coefficients et les variables (u, y mais aussi $\boldsymbol{x}$ et $\boldsymbol{t}$ pour les state-space et les SIF), et 32 bits pour les calculs intermédiaires (schémas $16 \times 16 \rightarrow 32$ pour la multiplication et $32 + 32 \rightarrow 32$ pour l'addition).

Ainsi, à partir des msb de u et y précédemment évalués, on obtient les formats $(m_u, \ell_u) = (8, -7)$ et $(m_y, \ell_y) = (11, -4)$ pour les signaux u et y respectivement.

On lance pour chacune des réalisations la méthodologie employée par `FiPoGen`, décrite précédemment et résumé par la figure 6.1. Pour une réalisation, les différents meilleurs oSoP sont générés en minimisant dans un premier temps les erreurs de calcul puis dans un second temps le parallélisme. Ensuite, du code entier `C` est généré à partir de chaque oSoP sélectionné.

Temps de génération : Le temps de génération et de choix des meilleurs oSoPs pour chaque réalisation est mesuré et retranscrit, avec le nombre d'opérations de chaque réalisation, dans le tableau 6.1. Le temps de génération pour la réalisation DFI n'a pas pu être estimé du fait de l'explosion combinatoire. En effet, cette réalisation est constituée d'un SoP de taille 17, or d'après l'équation (4.1), il y a environ $2 \cdot 10^{17}$ oSoP de taille 17 à considérer. À titre de comparaison, il faut plus d'une heure pour générer les 9 oSoPs de taille 9 (environ 18 millions d'oSoP au total à considérer) de la réalisation $SS_{\varepsilon_{Hp}}$. Il a donc été choisi, pour la DFI uniquement, de générer l'accumulation (aucun parallélisme dans l'arbre syntaxique) qui minimise uniquement les erreurs de calcul. Cette génération est quasi instantanée et le temps n'est pas indiquée dans le tableau 6.1 pour ne pas fausser la comparaison.

Réalisations	Temps ($mm'ss''cc$)	Nombre d'opérations
DFI	–	17 × et 16 +
SIF_p	$00'00''03$	40 × et 24 +
SIF_{LGS}	$02'27''61$	73 × et 34 +
SS_{eq}	$00'43''12$	81 × et 72 +
SS_{al}	$02'17''48$	81 × et 72 +
$SS_{\mathcal{L}_2}$	$5'48''87$	81 × et 72 +
SS_{ε_H}	$11'00''89$	81 × et 72 +
SS_{ε_p}	$9'31''61$	81 × et 72 +
$SS_{\varepsilon_{Hp}}$	$04'15''56$	81 × et 72 +

TABLE 6.1 – Comparaison des temps de génération et des nombres d'opérations pour chaque réalisation.

Remarque 6.2. *Il faut cependant noter que ces résultats sont propres au filtre choisit, ceci étant dû à la génération qui utilise une heuristique pour limiter le nombre d'oSoP générés qui dépend des formats virgule fixe des différents produits. Ce phénomène s'observe également pour les réalisations équivalentes en nombre d'opérations d'un même filtre, comme ici pour les différents state-space : le nombre de multiplications et d'additions est le même mais les coefficients différents impliquent des temps de calculs très variés.*

Intervalles d'erreur : Lors de la génération des différents oSoP de chaque réalisation, les erreurs de calculs ont été évaluées, il est donc possible de calculer l'intervalle de l'erreur finale sur le résultat de chaque réalisation, en appliquant le corollaire 3.1. Les résultats sont retranscrits dans le tableau 6.2.

On peut alors observer qu'en théorie, la réalisation $SS_{\mathcal{L}_2}$ est meilleure que les autres réalisations, en considérant la plus grande valeur absolue des deux bornes de chaque intervalle.

Réalisations	Intervalle d'erreurs ($[\underline{\delta y}; \overline{\delta y}]$)
DFI	$[-2.37205; 4.21885\texttt{e-}15]$
SIF_ρ	$[-1.62242; 0.176747]$
SIF_{LGS}	$[-0.608507; 5.66752]$
SS_{eq}	$[-1.43509; 0.110959]$
SS_{al}	$[-31.4617; 1.60702]$
$SS_{\mathcal{L}_2}$	$[-1.31393; 0.311331]$
SS_{ε_H}	$[-3.21514; 0.161485]$
SS_{ε_p}	$[-2.65737; 0.189219]$
$SS_{\varepsilon_{Hp}}$	$[-3.44109; 0.168142]$

Table 6.2 – Comparaison des intervalles d'erreur théoriques pour chaque réalisation.

Pour vérifier la validité de la théorie, on se propose de comparer les résultats des codes générés. Pour cela, on compare l'erreur entre une implémentation double précision de référence et le code généré pour chaque réalisation, sur 10000 échantillons du signal aléatoire u. On commence par comparer les différentes state-space entre eux afin de comparer par la suite le meilleur d'entree eux avec les autres réalisations. La figure 6.3 illustre les erreurs mesurées pour chaque state-space sur 10000 impulsions aléatoires.

Il apparaît clairement que le state-space aléatoire SS_{al} offre de mauvaises performances numériques par rapport aux autres state-space qui semblent assez proches les uns des autres. On réitère alors la simulation sans considérer SS_{al} afin d'avoir une observation plus nette sur les autres state-space. La figure 6.4 illustre ce procédé.

D'après ces observations, le state-space $SS_{\mathcal{L}_2}$ semble offrir les meilleures performances numériques par rapport aux autres state-space. Cela signifie que les coefficients du state-space équilibré de départ étaient plus sensibles à la mesure $\mathcal{L}_2$ et que la minimisation de cette mesure était le meilleur choix pour ce filtre.

Comparons maintenant le state-space $SS_{\mathcal{L}_2}$ aux autres réalisations non state-space, à savoir la structure LGS, la ρDFIIt et le DFI. Sur le même principe, la figure 6.5 illustre les résultats des erreurs produites par les réalisations par rapport à une implémentation double précision sur 10000 échantillons.

Le state-space $SS_{\mathcal{L}_2}$ reste, pour ce filtre, la réalisation qui offre les meilleures performances numériques. En effet, les réalisations DFI et SIF_ρ sont légèrement moins bonnes numériquement, mais restent proches des autres state-spaces en comparaison avec la figure 6.4. La réalisation SIF_{LGS} est également moins bonne (un facteur 2 avec la DFI ou avec les state-space SS_{ε_H} et $SS_{\varepsilon_{Hp}}$, en valeur absolue des bornes) mais offre tout de même une précision bien

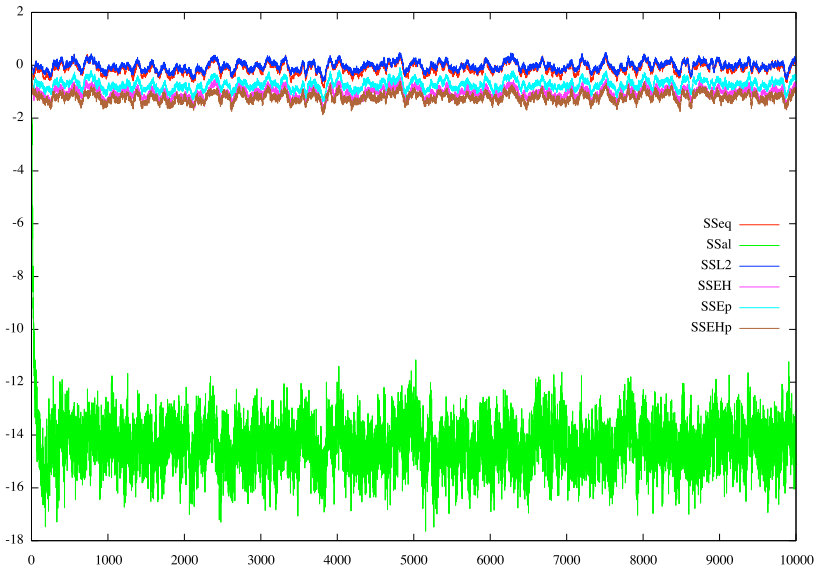


FIGURE 6.3 – Comparaison des erreurs produites par les différents state-space par rapport à une implémentation double précision de référence. Légende : SSeq $= SS_{eq}$, SSal $= SS_{al}$, SSL2 $= SS_{\mathcal{L}_2}$, SSEH $= SS_{\varepsilon_H}$, SSEp $= SS_{\varepsilon_p}$, SSEHp$= SS_{\varepsilon_{Hp}}$.

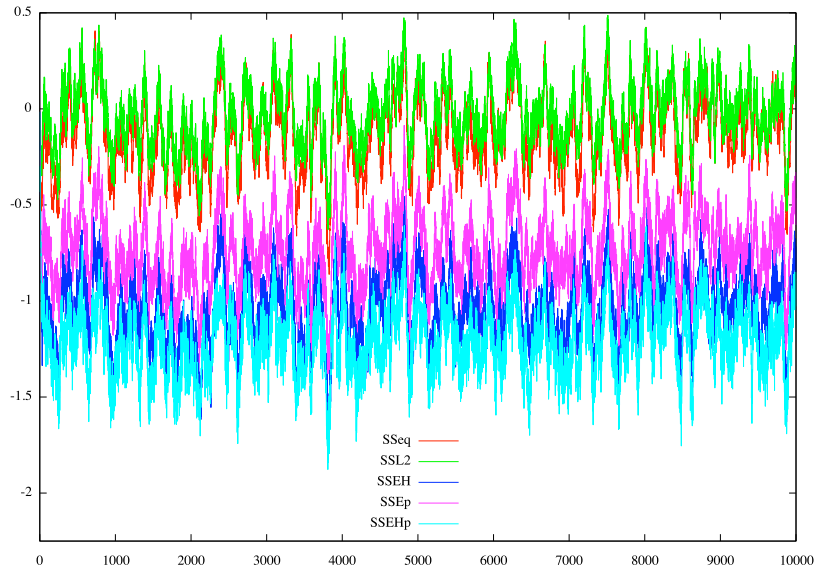


FIGURE 6.4 – Comparaison des erreurs produites par les différents state-space par rapport à une implémentation double précision de référence. Légende : SSeq $= SS_{eq}$, SSL2 $= SS_{\mathcal{L}_2}$, SSEH $= SS_{\varepsilon_H}$, SSEp $= SS_{\varepsilon_p}$, SSEHp$= SS_{\varepsilon_{Hp}}$.

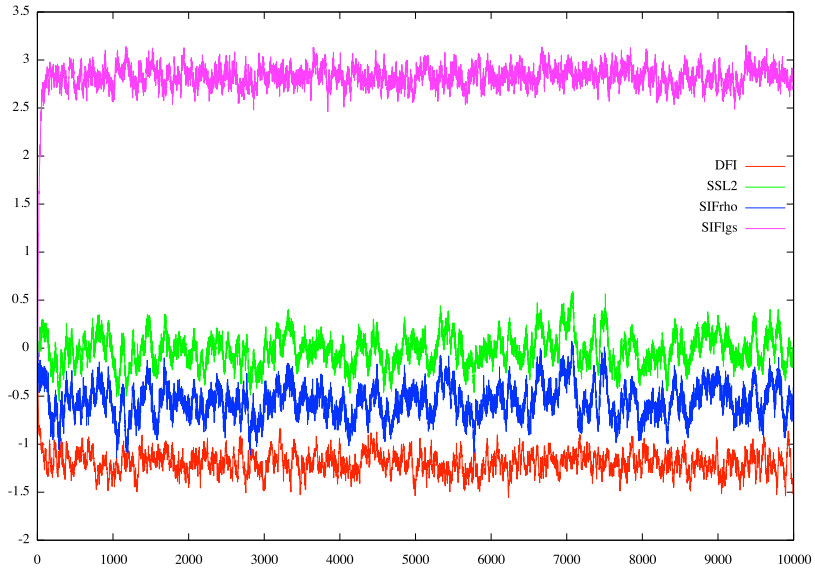


Figure 6.5 – Comparaison des erreurs produites par le state-space $SS_{\mathcal{L}_2}$ et les réalisations non state-space, par rapport à une implémentation double précision de référence. Légende : SSL2 = $SS_{\mathcal{L}_2}$, SIFrho = SIF_ρ, SIFlgs = SIF_{LGS}.

meilleurs que le state-space aléatoire.

Il faut cependant recouper les différentes informations et remarquer que si le state-space $SS_{\mathcal{L}_2}$ offre la meilleure précision numérique pour ce filtre, il est aussi l'un de ceux impliquant le plus long temps de génération du code virgule fixe. La réalisation SIF_ρ offre une précision relativement proche et la génération du code virgule fixe est quasi instantanée, pour un nombre d'opération également meilleur. Si l'on ne prend pas en compte le parallélisme de la cible, la DFI peut être très rapide et plutôt performante numériquement dans cet exemple, mais totalement irréalisable si l'on cherche à maximiser la précision et le parallélisme.

6.4 Implémentation matérielle

On souhaite, pour l'implémentation matérielle, déterminer les largeurs des coefficients et des variables de chaque réalisation pour avoir une erreur sur la sortie proche de celles observées précédemment pour l'implémentation logicielle. La borne sur l'erreur sera commune pour toutes les réalisations, afin de pouvoir en déduire des informations par comparaison des résultats. On choisit pour cela une borne égale à 2, c'est-à-dire on souhaite que l'erreur sur la sortie de chaque réalisation soit dans l'intervalle $[-2; 2]$.

Pour l'ensemble des réalisations, puisque la largeur du signal est u est arbitraire dans notre approche et afin de baser les comparaisons sur un signal commun, la largeur de u est fixé à 18 bits, ce qui correspond à une moyenne des largeurs calculées pour les variables de l'ensemble des réalisations.

Pour chacune des autres réalisations, les contraintes sont calculées par `FiPoGen` à partir des DC-gain et WCPG de chaque réalisation, et ces contraintes sont données au solveur `Bonmin`. On rappelle que les contraintes concernées par l'optimisation sont les contraintes des équations (5.28) dans le cas d'un arrondi par troncature et les contraintes de l'équation (5.35) dans le cas d'un arrondi au plus proche. Ainsi, pour un state-space l'optimisation donne une solution pour les largeurs des variables $\boldsymbol{x}_i$ et y, et pour une SIF la solution de l'optimisation comprend également les largeurs des variables $\boldsymbol{t}_i$.

Les résultats de l'optimisation pour chaque réalisation sont consignés dans le tableau 6.3. Le critère optimalité du tableau précise si la solution obtenue a été prouvée optimale par le solveur ou si la solution est sous-optimale. Le nombre total de bits correspond à la somme des largeurs des différentes variables du problème.

Réalisations	Temps de résolution (sec)	Optimalité	Nombre de bits des variables
SIF_ρ	836.15	`Oui`	293
SIF_{LGS}	600	`Non`	749
SS_{eq}	1.62	`Oui`	107
SS_{al}	3.87	`Oui`	182
$SS_{\mathcal{L}_2}$	0.28	`Oui`	142
SS_{ε_H}	8.03	`Oui`	157
SS_{ε_p}	4.76	`Oui`	155
$SS_{\varepsilon_{Hp}}$	8.12	`Oui`	158

TABLE 6.3 – Résultats retournés par l'optimisation du solveur `Bonmin`.

Le temps de résolution maximal était fixé à 600 secondes pour ces tests et `Bonmin` n'a pas trouvé de solution optimale pour la réalisation SIF_{LGS}. Ceci est dû au grand nombre de variables à déterminer, 38 dans ce cas. En fait, au bout de ces 10 minutes, la meilleure borne inférieure obtenue en considérant le problème continue a été 748.062, et la meilleure solution entière 749, on peut alors penser qu'on était très proche de la solution optimale mais toute la difficulté réside justement dans la preuve de cet optimalité par le solveur.

Cela souligne l'intérêt d'un algorithme de résolution spécifique à notre problème, plutôt qu'un solveur pour une classe globale dont fait partie notre problème. En effet, exploiter chaque information du problème dans un algorithme dédié pourrait permettre de résoudre plus efficacement le problème, et peut-être d'arriver plus rapidement une solution optimale.

Dans le cas de la réalisation SIF_ρ, vu le nombre de variables plus petits que pour la LGS (17 contre 38), nous savions qu'avec un peu plus de temps le solveur pourrait prouver l'optimalité de la solution, nous avons donc repoussé la limite de temps pour cette réalisation.

Pour la réalisation "DFI" il n'y a pas d'optimisation car cette réalisation composée d'un seul produit scalaire entre dans le cas particulier décrit dans la section 5.4.1. Ainsi, `FiPoGen` applique les formules (5.45) et (5.46) et détermine ainsi les largeurs de la sortie y et des coefficients $\boldsymbol{a}_i$ et $\boldsymbol{b}_j$ pour $1 \leqslant i \leqslant 8$ et $0 \leqslant j \leqslant 8$. Cette réalisation a également la particularité par rapport aux autres réalisations considérées d'utiliser directement les sorties y et les entrées u des 8 instants précédents à chaque itération. Cela signifie qu'il faut stocker ces valeurs d'une itération à l'autre et par conséquent les largeurs de ces variables sont également comptabilisées dans le nombre de bits total de la réalisation.

Une fois les largeurs des variables obtenues, `FiPoGen` en déduit les largeurs des coefficients à l'aide des contraintes de formatages de bits de l'équation (5.19). Le tableau 6.4 présente les résultats de ces calculs. Pour chaque réalisation, le nombre indiqué correspond à la somme des largeurs des variables obtenues par `Bonmin` et des largeurs des coefficients déduites par l'équation (5.19).

Réalisations	Nombre de bits total
DFI	670
SIF_ρ	1025
SIF_{LGS}	1702
SS_{al}	2318
$SS_{\mathcal{L}_2}$	1642
SS_{ε_H}	1898
SS_{ε_p}	1882
$SS_{\varepsilon_{Hp}}$	1891

Table 6.4 – Résultats finaux après déduction des largeurs des coefficients à partir des résultats trouvés par `Bonmin`.

Remarque 6.3. *La réalisation SS_{eq} semble intéressante, car la résolution de son problème est rapide et le nombre de bits obtenu pour ses variables est le plus petit, mais cela nous a justement posé problème par la suite. En effet, certaines largeurs trop petites (5 ou 3 bits), produisent, en appliquant les contraintes de formatage de bits, des coefficients dont la largeur est négative, et donc la suppression de certains produits de l'algorithme. Notre outil permet quand même de générer un code mais qui perd tout son sens du fait de cette perte d'information.*

Ceci peut signifier deux choses : soit nos contraintes ne sont pas assez fortes pour garantir la génération d'un code cohérent, soit cette instance possède une particularité qui la rend irréalisable par notre approche. Dans tous les cas, en l'absence d'une solution évidente à ce problème, il a été décidé de ne plus considérer cette réalisation dans le suite de notre exemple.

D'après le tableau 6.4 on peut observer que la DFI nécessite très peu de bits pour être implémentée par rapport aux autres réalisations, d'autant plus que ses largeurs sont calculées directement sans passer par un processus d'optimisation. Ceci dit, la DFI souffre du même problème que dans le cas logiciel, à savoir le temps de génération du code, une fois les largeurs connues. En effet, la prochaine étape, une fois les différentes largeurs de chaque réalisation connues, est de générer le code virgule fixe, en cherchant par exemple à minimiser le parallélisme. Dans le cas de l'optimisation des largeurs, notre algorithme de génération d'oSoP est utilisé et s'arrête dès qu'un oSoP de hauteur minimale est atteinte. Si pour l'ensemble des réalisations cela accélère la génération par rapport au cas logiciel, pour la DFI cela reste irréalisable, et on considère alors la même accumulation que précédemment. Il serait aussi possible de construire l'oSoP en largeur plutôt qu'en profondeur dans ce cas là mais ce type de solution n'est pas implémentée dans notre outil.

L'ensemble des codes générés pour les différentes réalisations possèdent une erreur sur la sortie dans l'intervalle $[-2; 2]$ (puisque c'était la contrainte fixée au départ), comme le montre la figure 6.6. Sur cette figure les code générés sont comparés à une implémentation double précision sur 1000 échantillons du signal d'entrée u.

6.5 Conclusion

Dans ce chapitre nous avons décrit puis illustré les fonctionnalités de l'outil `FiPoGen`, à travers un exemple comparant différentes réalisations d'un même filtre. Le but n'est bien sûr pas de comparer formellement les réalisations, puisque la sensibilité de chaque réalisation est propre au filtre choisit, et donc la réalisation idéale pour un filtre peut être différente de la réalisation idéale pour un autre filtre.

Cet exemple a cependant permis de valider, par l'utilisation de l'outil, la théorie développée au sein de ce manuscrit, notamment pour l'analyse de l'erreur et les contraintes de notre problème d'optimisation.

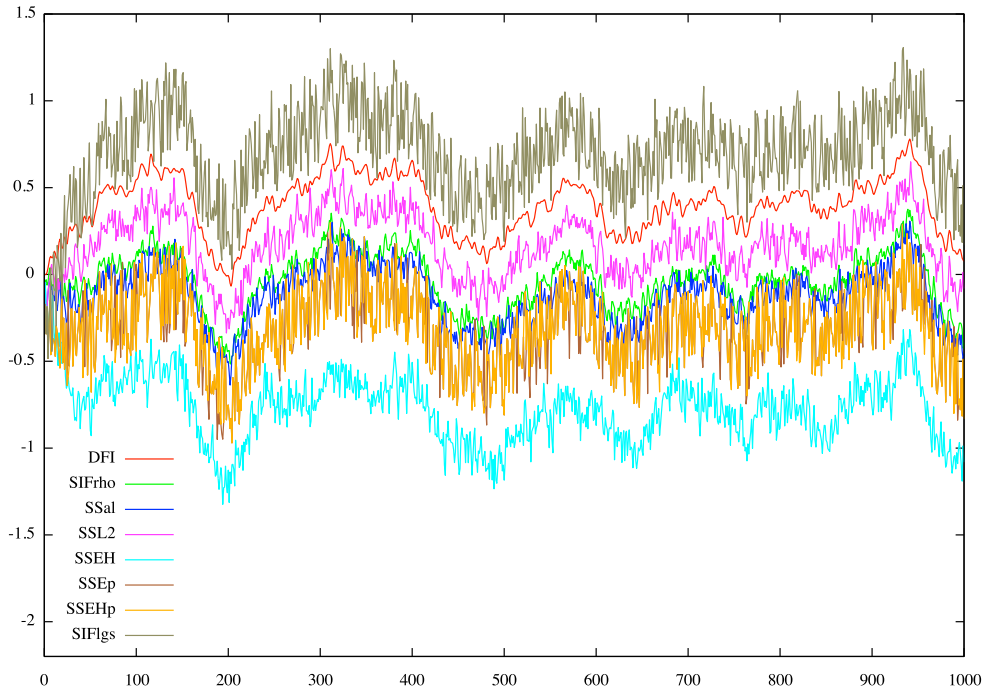


Figure 6.6 – Comparaison des erreurs produites par les différentes réalisations par rapport à une implémentation double précision de référence. Légende : SSL2 = $SS_{\mathcal{L}_2}$, SSEH = SS_{ε_H}, SSEp = SS_{ε_p}, SSEHp= $SS_{\varepsilon_{Hp}}$, SIFrho = SIF_ρ, SIFlgs = SIF_{LGS}.

Conclusion et perspectives

"*Science never solves a problem without creating ten more.*"
- George Bernard Shaw

Conclusion

Le choix de la meilleure implémentation de filtres LTI en virgule fixe est sujet à des contraintes diverses, et celles-ci dépendent de la cible visée. La conversion en virgule fixe implique des dégradations qui sont soit des erreurs de quantification des coefficients, soit des erreurs de calcul. Les erreurs de quantification dépendent des largeurs allouées aux coefficients et de la finesse de l'arithmétique virgule fixe utilisée (une mauvaise estimation de la position de la virgule peut conduire a une dégradation plus importante du coefficient), tandis que les erreurs de calcul dépendent des largeurs des opérateurs, de l'ordre des calculs et des modes d'arrondi. En logiciel, les largeurs sont fixées, l'action va donc se concentrer le formalisme virgule fixe et l'ordre des calculs, la meilleure implémentation sera celle qui sera le moins dégradée numériquement tout en exploitant au mieux les ressources (parallélisme quand cela est possible). En matériel, les largeurs ne sont pas fixées et l'enjeu est alors de minimiser ces largeurs pour respecter des contraintes numériques, on dira alors que la meilleure implémentation est celle qui minimisera les largeurs (ou la surface,la consommation, etc.), sous contraintes de précision numérique.

Pour répondre à cette problématique, une méthodologie a été mise en place durant cette thèse.

La première étape a été de rappeler les bases des filtres LTI afin de pouvoir en exploiter certaines propriétés et en déduire de nouvelles propriétés, utiles à notre approche. Parmi celles-ci, la linéarité du filtre nous a permis d'exprimer le filtre dégradé comme le filtre exact perturbé, sur la sortie, par la sortie d'un filtre alimenté par toutes les erreurs internes, permettant ainsi de résoudre le

problème du "rebouclage". La connaissance de ce filtre de transport des erreurs nous a permis, en utilisant une autre propriété des filtres LTI, d'évaluer les bornes théoriques (mais que l'on peut atteindre à un epsilon près) de l'erreur sur la sortie, ce qui, dans le contexte matériel de l'optimisation des largeurs, nous permet d'établir une contrainte sur cette erreur. Cette même propriété nous permet également de déterminer les bornes de la sortie d'un filtre, et donc d'en déduire son format virgule fixe. Cette information est utile, aussi bien dans le cas logiciel que dans le cas matériel, pour limiter la propagation des intervalles dans les sommes de produits quand elle devient inutile, en se débarrassant des bits de poids forts inutiles.

La méthodologie proposée diffère selon que la cible est logicielle ou matérielle. Pour une cible logicielle, la méthodologie consiste à générer tous les schémas d'évaluation virgule fixe possibles, à propager les intervalles à travers chacun d'eux pour en déduire les erreurs de calcul et ensuite en déduire quel est le meilleur d'entre eux (en prenant en compte d'autres critères comme le degré de parallélisme de chaque schéma). Afin de réaliser cette approche le plus finement possible, un formalisme de l'arithmétique virgule fixe a été proposé, utilisant l'arithmétique d'intervalle.

Pour une cible matérielle, on cherche à minimiser les largeurs pour respecter certaines contraintes numériques. Dans ce contexte, l'approche a consisté à formaliser le problème d'optimisation exprimant nos contraintes. Ces contraintes imposent, comme décrit précédemment, une borne sur l'erreur finale, mais aussi de calculer la somme de produits sur un format intermédiaire supprimant tous les bits potentiellement faux des différents produits.

Ces deux approches sont mise en œuvre dans un outil, FiPoGen, développé en `Python` durant ces trois ans, afin de répondre à la problématique de façon automatique. À partir des données d'un filtre et de certains autres paramètres comme les différentes largeurs en logiciel ou la borne sur l'erreur à respecter en matériel, FiPoGen est capable de générer le ou un des *meilleurs* codes virgule fixe implémentant ce filtre, au sens défini pour chaque approche.

Cependant, l'approche globale que nous avons proposé durant cette thèse souffre de quelques inconvénients. En effet, l'implémentation logicielle, si les produits scalaires considérés ont un nombre élevé de produits, peut être très longue. Cela est dû au nombre de schémas à générer pour un filtre de donné. Dans les faits, l'outil utilise une heuristique pour limiter ce nombre de schémas en excluant trivialement certains cas, mais le nombre de schémas effectivement générés reste sensiblement du même ordre de grandeur que le nombre de schémas générés sans l'heuristique.

Une autre limitation de notre approche réside dans l'utilisation d'un solveur externe pour l'optimisation des largeurs. Le solveur `Bonmin` est libre de droit mais le langage de description `AMPL` utilisé par le solveur ne l'est pas, ce qui peut entraîner des problèmes pour la portabilité de notre outil. De plus, le solveur `Bonmin` est dédié aux problèmes d'optimisation convexes et non-linéaires,

dont fait partie notre problème mais qui est une classe très générale. Cela signifie que la solution apportée par le solveur, qu'elle soit optimale ou sous-optimale, numériquement parlant, ne l'est pas en ce qui concerne le temps de résolution. Une meilleure connaissance des propriétés de notre problème pourrait permettre d'atteindre la solution optimale (ou une solution sous-optimale) plus rapidement. La solution idéale est de développer dans l'outil FiPoGen un solveur propre à notre problème d'optimisation, afin d'être plus rapide et indépendant d'autres outils.

Perspectives

L'implémentation de filtres linéaires en arithmétique virgule fixe est un sujet très vaste et de nombreuses pistes ont été abordées, que ce soit les propriétés de ces filtres, la formalisation de l'arithmétique virgule fixe, ou les deux approches logicielle et matérielle qui ont chacune leurs spécificités. De par cette diversité des pistes, certains points n'ont pas pu être complètement explorés.

Ainsi, un meilleur algorithme pour la génération des schémas est d'ores-et-déjà envisagé, se limitant aux seuls schémas efficaces numériquement, en somme utilisant une heuristique beaucoup plus forte que celle utilisée actuellement par FiPoGen. Celui-ci devrait, dans le pire des cas, se limiter à une centaine de schémas pour une somme de neuf produits, au lieu de plus de deux millions actuellement.

Une analyse plus complète de notre problème d'optimisation nous permettrait d'établir un algorithme d'optimisation dédié à ce problème et d'en déduire une implémentation dans l'outil FiPoGen.

Cet outil ayant par ailleurs vocation à être un logiciel libre et également disponible sous une interface en ligne, nécessiterait encore du développement pour être pleinement et ergonomiquement utilisable.

Un dernier exemple de perspective concerne encore une fois l'outil FiPoGen. Si le sigle FiPoGen est aussi vague (on rappelle que FiPoGen est une abréviation pour ***Fi**xed-**Po**int code **Gen**erator*) et ne fait pas intervenir le terme de *filtres*, c'est parce que l'objectif majeur est d'étendre les connaissances acquises sur l'implémentation en virgule fixe à diverses classes d'algorithmes, comme les algorithmes d'algèbre linéaire, et d'autres classes d'algorithmes de traitement du signal tel que les filtres adaptatifs.

Annexe A

Structures LGS et LCW

A.1 Paramétrisation JSS

Décrite dans [52, 68] par Johns, Snelgrove et Sedra (d'où le nom JSS donné à la paramétrisation par Li *et al.*), cette paramétrisation caractérise en fait un state-space dans le domaine continu. Elle offre l'avantage d'avoir une matrice de coefficients très creuse mais également un très bon conditionnement numérique par rapport à la norme L_2.

Soient $\mathcal{H}$ un filtre et $F(s)$ sa fonction de transfert dans le domaine continu, $F(s)$ est de la forme

$$F(s) = \frac{N(s)}{D(s)}, \tag{A.1}$$

où D et N sont deux polynômes en s. Le polynôme D peut se décomposer en une somme de deux polynômes, l'un composé des monômes de degré pair, noté E, et l'autre composé des monômes de degré impair, noté O.

On pose alors $Z(s) = \frac{O(s)}{E(s)}$ si le degré de D est impair et $Z(s) = \frac{E(s)}{O(s)}$ si le degré de D est pair. On note A le polynôme au numérateur et B le polynôme au dénominateur, on a alors $Z(s) = \frac{A(s)}{B(s)}$ et $deg(A) = deg(B) + 1$.

On peut alors écrire Z sous la forme d'une fraction continue :

$$Z(s) = r_n s + \cfrac{1}{r_{n-1}s + \cfrac{1}{r_{n-2}s + \cfrac{1}{\ddots \frac{1}{r_1 s}}}} \tag{A.2}$$

Pour cela, on décompose $A(s)$ en

$$A(s) = LM_A(s) + R_A(s) \tag{A.3}$$

où LM_A est le monôme de plus haut degré de A (leading monomial) et R_A est le polynôme formé des autres monômes de A. Soient Q et R, respectivement le

quotient et le reste de la division de LM_A par B. Puisque $deg(A) = deg(B)+1$, on a $Q = r_n s$, où r_n est une constante. On a alors :

$$\begin{aligned} Z(s) &= \frac{A(s)}{B(s)} = \frac{LM_A(s)}{B(s)} + \frac{R_A(s)}{B(s)} = Q(s) + \frac{R(s)}{B(s)} + \frac{R_A(s)}{B(s)} && \text{(A.4)} \\ &= Q + \frac{R + R_A}{B} && \text{(A.5)} \\ &= r_n s + \frac{R + R_A}{B} && \text{(A.6)} \\ &= r_n s + \frac{1}{\frac{B}{R+R_A}} && \text{(A.7)} \end{aligned}$$

Soit $d \triangleq deg(A)$. On a, par construction, $deg(B) = d - 1$ et $deg(R_A) = d - 2$. De plus, puisque R est le reste de la division de LM_A par B, on a $deg(R) < deg(B)$, et par conséquent on a $deg(B) = deg(R + R_A) + 1$, on peut alors répéter l'opération jusqu'à obtenir une décomposition complète.

Une fois la décomposition de Z sous forme de fraction continue complétée, et les $r_i \in \mathbb{R}$ tous obtenus, on calcule $\alpha_1, \alpha_2, \ldots, \alpha_n$:

$$\begin{aligned} \alpha_k &= \sqrt{\frac{1}{r_k r_{k+1}}}, \quad k = 1, 2, \ldots, n-1 && \text{(A.8)} \\ \alpha_n &= \frac{1}{r_n} && \text{(A.9)} \end{aligned}$$

On construit alors la matrice $\boldsymbol{\Phi}_{in}$ et le vecteur $\boldsymbol{k}_{in}$ de la façon suivante :

$$\boldsymbol{\Phi}_{in} \triangleq \begin{pmatrix} 0 & \alpha_1 & & & & \\ -\alpha_1 & 0 & \alpha_2 & & \text{\Large 0} & \\ & -\alpha_2 & 0 & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & \text{\Large 0} & & \ddots & 0 & \alpha_{n-1} \\ & & & & -\alpha_{n-1} & -\alpha_n \end{pmatrix}, \qquad \text{(A.10)}$$

$$\boldsymbol{k}_{in} \triangleq \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \sqrt{2\alpha_n} \end{pmatrix}. \qquad \text{(A.11)}$$

Si la fonction de transfert F est décrite sous la forme d'un state-space ayant pour paramétrisaiton $(\boldsymbol{\Phi}, \boldsymbol{k}, \boldsymbol{l}, m)$, alors de la même manière que dans le domaine discret, toute fonction de transfert ayant pour paramétrisation

$(\boldsymbol{T}^{-1}\boldsymbol{\Phi}\boldsymbol{T}, \boldsymbol{T}^{-1}\boldsymbol{k}, \boldsymbol{l}\boldsymbol{T}, m)$ est mathématiquement équivalente à F, pour toute transformation non singulière $\boldsymbol{T}$. Par conséquent, à partir de $\boldsymbol{\Phi}_{in}$ et $\boldsymbol{k}_{in}$, on peut trouver $\boldsymbol{T}_{in}$ telle que :

$$\boldsymbol{\Phi}_{in} = \boldsymbol{T}_{in}^{-1}\boldsymbol{\Phi}\boldsymbol{T}_{in} \tag{A.12}$$

$$\boldsymbol{k}_{in} = \boldsymbol{T}_{in}^{-1}\boldsymbol{k} \tag{A.13}$$

et en déduire $\boldsymbol{l}_{in} \triangleq \boldsymbol{l}\boldsymbol{T}_{in}$, de sorte que $(\boldsymbol{\Phi}_{in}, \boldsymbol{k}_{in}, \boldsymbol{l}_{in}, m)$ est équivalente à la paramétrisation $(\boldsymbol{\Phi}, \boldsymbol{k}, \boldsymbol{l}, m)$.

A.2 Structure LGS

On l'a vu, cette structure s'inspire de la paramétrisation JSS dans le domaine continue, il est donc nécessaire, étant données une fonction de transfert H dans le domaine discret et une paramétrisation $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, d)$, de passer dans le domaine continue pour appliquer la transformation qui nous donnera la paramétrisation JSS. Pour passer de la fonction de transfert $H(z)$ dans le domaine discret à son équivalent dans le domaine continue $F(s)$, on applique la transformée de Tustin [84], c'est-à-dire le changement de variable $s = \frac{z-1}{z+1}$, ou $z = \frac{1+s}{1-s}$, tel que :

$$F(s) = H(z)|_{z=\frac{1+s}{1-s}}. \tag{A.14}$$

Si $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, d)$ est une paramétrisation de H, il peut être montré qu'il existe une paramétrisation de F, $(\boldsymbol{\Phi}, \boldsymbol{k}, \boldsymbol{l}, m)$, telle que :

$$\boldsymbol{\Phi} = (\boldsymbol{I} + \boldsymbol{A})^{-1}(\boldsymbol{A} - \boldsymbol{I}) \tag{A.15a}$$

$$\boldsymbol{k} = \sqrt{2}(\boldsymbol{I} + \boldsymbol{A})^{-1}\boldsymbol{b} \tag{A.15b}$$

$$\boldsymbol{l} = \sqrt{2}\boldsymbol{c}(\boldsymbol{I} + \boldsymbol{A})^{-1} \tag{A.15c}$$

$$m = d - \boldsymbol{c}(\boldsymbol{I} + \boldsymbol{A})^{-1}\boldsymbol{b} \tag{A.15d}$$

À partir de la fonction de transfert $F(s)$, on peut calculer $\boldsymbol{\Phi}_{in}$ et $\boldsymbol{k}_{in}$ comme dans les équations (A.10) et (A.11), et en déduire la transformation $\boldsymbol{T}_{in}$ pour passer de la paramétrisation $(\boldsymbol{\Phi}, \boldsymbol{k}, \boldsymbol{l}, m)$ à la paramétrisation $(\boldsymbol{\Phi}_{in}, \boldsymbol{k}_{in}, \boldsymbol{l}_{in}, m)$, comme vu précédemment.

On repasse alors dans le domaine discret à partir de cette nouvelle paramétrisation en appliquant l'inverse de la transformations des équations (A.15), on a alors :

$$\boldsymbol{A}_{in} = (\boldsymbol{I} + \boldsymbol{\Phi}_{in})(\boldsymbol{I} - \boldsymbol{\Phi}_{in})^{-1} \tag{A.16a}$$

$$\boldsymbol{b}_{in} = \frac{\sqrt{2}}{2}(\boldsymbol{I} + \boldsymbol{A}_{in})\boldsymbol{k}_{in} \tag{A.16b}$$

$$\boldsymbol{c}_{in} = \frac{\sqrt{2}}{2}\boldsymbol{l}_{in}(\boldsymbol{I} + \boldsymbol{A}_{in}) \tag{A.16c}$$

$$d = m + \boldsymbol{c}_{in}(\boldsymbol{I} + \boldsymbol{A}_{in})^{-1}\boldsymbol{b}_{in} \tag{A.16d}$$

L'inversion de matrice dans l'équation (A.23a) implique que la matrice $\boldsymbol{A}_{in}$ est dense, contrairement à la matrice $\boldsymbol{\Phi}_{in}$ qui est creuse. On montre alors que $\boldsymbol{A}_{in}$ peut se factoriser en un produit de matrice creuse (un seul coefficient non trivial par matrice).

Soit $\boldsymbol{U}(i,j,x)$ la matrice définie comme la matrice identité sauf l'élément en position (i,j) qui vaut x, c'est-à-dire :

$$\boldsymbol{U}(i_0,j_0,x)_{i,j} \triangleq \begin{cases} 1 & \text{si } i=j \\ x & \text{si } i=i_0 \text{ et } j=j_0 \\ 0 & \text{sinon} \end{cases} \tag{A.17}$$

On peut montrer que $\boldsymbol{I}-\boldsymbol{\Phi}_{in}$ s'écrit :

$$\boldsymbol{I}-\boldsymbol{\Phi}_{in} = \boldsymbol{T}_1^{-1}\boldsymbol{T}_2^{-1}\dots\boldsymbol{T}_k^{-1}\dots\boldsymbol{T}_{n-1}^{-1}\boldsymbol{\Psi}, \tag{A.18}$$

avec

$$\boldsymbol{T}_k \triangleq \boldsymbol{U}(k+1,k+1,\gamma_k)\boldsymbol{U}(k+1,k,-\alpha_k), \tag{A.19}$$

$\boldsymbol{\Psi}$ la matrice définie par :

$$\boldsymbol{\Psi}_{i,j} \triangleq \begin{cases} 1 & \text{si } i=j \\ \beta_k & \text{si } j=i+1 \\ 0 & \text{sinon} \end{cases}, \tag{A.20}$$

les coefficients β_k sont définis par la récurrence suivante :

$$\begin{array}{rclrcll} \beta_{k+1} & \triangleq & -\frac{\alpha_{k+1}}{s_k}, & s_k & \triangleq & 1-\alpha_k\beta_k, & \forall 1 \leqslant k \leqslant n-2, \\ \beta_1 & \triangleq & -\alpha_1, & s_{n-1} & \triangleq & 1+\alpha_n-\alpha_{n-1}\beta_{n-1}, & \end{array}$$

et enfin les coefficients γ_k sont définis par

$$\gamma_k \triangleq s_k^{-1}, \quad \forall 1 \leqslant k \leqslant n-1. \tag{A.21}$$

On peut remarquer que l'inverse de $\boldsymbol{\Psi}$ peut s'écrire :

$$\boldsymbol{\Psi}^{-1} = \boldsymbol{U}(1,2,-\beta_1)\dots\boldsymbol{U}(n-1,n,-\beta_{n-1}) = \prod_{i=1}^{n-1}\boldsymbol{U}(i,i+1,-\beta_i).$$

Finalement, toutes ces notations permettent de factoriser la matrice $\boldsymbol{A}_{in}$:

$$\begin{aligned} \boldsymbol{A}_{in} &= (\boldsymbol{I}-\boldsymbol{\Phi}_{in})(\boldsymbol{I}+\boldsymbol{\Phi}_{in})^{-1} \\ &= (\boldsymbol{I}-\boldsymbol{\Phi}_{in})\boldsymbol{\Psi}^{-1}\boldsymbol{T}_{n-1}\boldsymbol{T}_{n-2}\dots\boldsymbol{T}_1 \\ &= \underbrace{(\boldsymbol{I}+\boldsymbol{\Phi}_{in})}_{\boldsymbol{A}^{(N)}}\underbrace{\boldsymbol{U}(1,2,-\beta_1)}_{\boldsymbol{A}^{(N-1)}}\dots\boldsymbol{U}(n-1,n,-\beta_{n-1}) \\ &\quad \times\boldsymbol{U}(n,n,\gamma_{n-1})\boldsymbol{U}(n,n-1,-\alpha_{n-1})\dots \\ &\quad \times\dots\underbrace{\boldsymbol{U}(2,2,\gamma_1)}_{\boldsymbol{A}^{(2)}}\underbrace{\boldsymbol{U}(2,1,-\alpha_1)}_{\boldsymbol{A}^{(1)}} \end{aligned} \tag{A.22}$$

On obtient donc

$$\boldsymbol{A}_{in} = \prod_{k=1}^{N} \boldsymbol{A}^{(k)} \quad \text{avec } N \triangleq 3n-1,$$

et le state-space caractérisé par la paramétrisation $(\boldsymbol{A}_{in}, \boldsymbol{b}_{in}, \boldsymbol{c}_{in}, d)$ devient, avec cette factorisation :

$$\begin{cases} \boldsymbol{x}^{(0)}(k) &= \boldsymbol{x}(k) \\ \boldsymbol{x}^{(m)}(k) &= \boldsymbol{A}^{(m)}\boldsymbol{x}^{(m-1)}(k), \quad k = 1, 2, \ldots, N \\ \boldsymbol{x}(k+1) &= \boldsymbol{x}^{(N)}(k) + \boldsymbol{b}_{in}u(k) \\ y(k) &= \boldsymbol{c}_{in}\boldsymbol{x}(k) + du(k) \end{cases}$$

L'algorithme ci-dessus n'est plus un state-space, et correspond à ce que l'on appelle la structure LGS.

A.3 Structure LCW

On reprend l'écriture de la DJSS décrite pour la structure LGS (équations (A.16)), rappelée ici :

$$\boldsymbol{A}_{in} = (\boldsymbol{I} + \boldsymbol{\Phi}_{in})(\boldsymbol{I} - \boldsymbol{\Phi}_{in})^{-1} \tag{A.23a}$$

$$\boldsymbol{b}_{in} = \frac{\sqrt{2}}{2}(\boldsymbol{I} + \boldsymbol{A}_{in})\boldsymbol{k}_{in} \tag{A.23b}$$

$$\boldsymbol{c}_{in} = \frac{\sqrt{2}}{2}\boldsymbol{l}_{in}(\boldsymbol{I} + \boldsymbol{A}_{in}) \tag{A.23c}$$

$$d = m + \boldsymbol{c}_{in}(\boldsymbol{I} + \boldsymbol{A}_{in})^{-1}\boldsymbol{b}_{in} \tag{A.23d}$$

On précise que les notations pour la paramétrisation JSS dans le domaine continu sont conservées ici.

On a alors :

$$\begin{aligned} \boldsymbol{A}_{in} &= (\boldsymbol{I} - \boldsymbol{\Phi}_{in})^{-1}(\boldsymbol{I} + \boldsymbol{\Phi}_{in}) \\ &= (2\boldsymbol{I} - (\boldsymbol{I} - \boldsymbol{\Phi}_{in}))(\boldsymbol{I} - \boldsymbol{\Phi}_{in})^{-1} \\ &= 2(\boldsymbol{I} - \boldsymbol{\Phi}_{in})^{-1} - I \end{aligned}$$

et on en déduit $\boldsymbol{B}_{in} = \sqrt{2}(\boldsymbol{I} - \boldsymbol{\Phi}_{in})^{-1}\boldsymbol{k}_{in}$.

On utilise maintenant une propriété des state-space dans le cas d'un filtre SISO. Cette propriété stipule que la paramétrisation duale d'une paramétrisation $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, d)$, obtenue par $(\boldsymbol{A}^\top, \boldsymbol{c}^\top, \boldsymbol{b}^\top, d)$, est une paramétrisation du même filtre. On note alors $(\boldsymbol{A}_t, \boldsymbol{b}_t, \boldsymbol{c}_t, d)$ la paramétrisation duale de $(\boldsymbol{A}_{in}, \boldsymbol{b}_{in}, \boldsymbol{c}_{in}, d)$

et on a :

$$\begin{aligned}
\boldsymbol{A}_t &\triangleq \boldsymbol{A}_{in}^\top = (\boldsymbol{I}+\boldsymbol{\Phi}_{in})^\top(\boldsymbol{I}-\boldsymbol{\Phi}_{in})^{-\top} = (\boldsymbol{I}-\boldsymbol{\Phi}_{in})^{-\top}(\boldsymbol{I}+\boldsymbol{\Phi}_{in})^\top && \text{(A.24)}\\
\boldsymbol{b}_t &\triangleq \boldsymbol{c}_{in}^\top && \text{(A.25)}\\
\boldsymbol{c}_t &\triangleq \boldsymbol{b}_{in}^\top = \sqrt{2}\boldsymbol{k}_{in}^\top(\boldsymbol{I}-\boldsymbol{\Phi}_{in})^{-\top} && \text{(A.26)}
\end{aligned}$$

On applique ensuite la transformation $\boldsymbol{T}^* \triangleq (\boldsymbol{I}-\boldsymbol{\Phi}_{in})^\top$ sur cette réalisation pour obtenir $(\boldsymbol{A}^*, \boldsymbol{b}^*, \boldsymbol{c}^*, d)$

$$\begin{aligned}
\boldsymbol{A}^* &= (\boldsymbol{I}-\boldsymbol{\Phi}_{in})^{-\top}(\boldsymbol{I}+\boldsymbol{\Phi}_{in})^\top = \boldsymbol{A}_{in}^\top\\
\boldsymbol{b}^* &= (\boldsymbol{I}-\boldsymbol{\Phi}_{in})^{-\top}\boldsymbol{c}_{in}^\top\\
\boldsymbol{c}^* &= \sqrt{2}\boldsymbol{k}_{in}^\top
\end{aligned}$$

On définit maintenant

$$\begin{array}{rclrclll}
\beta_{k+1} & \triangleq & -\alpha_{k+1}\gamma_k, & \gamma_k & \triangleq & \frac{1}{1-\alpha_k\beta_k}, & & \forall 1 \leqslant k \leqslant n-2,\\
\beta_1 & \triangleq & -\alpha_1, & \gamma_{n-1} & \triangleq & \frac{1}{1+\alpha_n-\alpha_{n-1}\beta_{n-1}}, & &
\end{array}$$

on reprend également les notations définies pour la structure LGS, $\boldsymbol{U}(i,j,x)$ (voir équation (A.17)) et $\boldsymbol{T}_k$ pour $1 \leqslant k \leqslant n-1$ (voir équation (A.19)).

On peut montrer que l'on a :

$$\begin{aligned}
((\boldsymbol{I}-\boldsymbol{\Phi}_{in})^{-1})^\top &= \boldsymbol{\Gamma}^{-1}\boldsymbol{T}_{n-1}\dots\boldsymbol{T}_2\boldsymbol{T}_1\\
&= \boldsymbol{\Gamma}^{-1}\boldsymbol{U}(n,n,\gamma_{n-1})\boldsymbol{U}(n,n-1,\alpha_{n-1})\dots\boldsymbol{U}(2,2,\gamma_1)\boldsymbol{U}(2,1,\alpha_1)\\
&= \boldsymbol{\Gamma}^{-1}\boldsymbol{A}_{2(n-1)}\boldsymbol{A}_{2(n-1)-1}\dots\boldsymbol{A}_2\boldsymbol{A}_1 && \text{(A.27)}
\end{aligned}$$

où

$$\boldsymbol{\Gamma}_{i,j} = \begin{cases} 1 & \text{si } i=j\\ -\beta_k & \text{si } j=i+1\\ 0 & \text{sinon}\end{cases}$$

On a la décomposition suivante pour $\boldsymbol{\Gamma}^{-1}$:

$$\begin{aligned}
\boldsymbol{\Gamma}^{-1} &= \boldsymbol{U}(1,2,\beta_1)\boldsymbol{U}(2,3,\beta_2)\dots\boldsymbol{U}(k,k+1,\beta_k)\dots\boldsymbol{U}(n-1,n,\beta_{n-1})\\
&= \boldsymbol{A}_{3(n-1)}\boldsymbol{A}_{3(n-1)-1}\dots\boldsymbol{A}_{3(n-1)-k}\dots\boldsymbol{A}_{2(n-1)+1} && \text{(A.28)}
\end{aligned}$$

D'après (A.27) et (A.28), on a :

$$\begin{aligned}
((\boldsymbol{I}-\boldsymbol{\Phi}_{in})^{-1})^\top &= \boldsymbol{\Gamma}^{-1}\boldsymbol{T}_{n-1}\dots\boldsymbol{T}_2\boldsymbol{T}_1\\
&= \boldsymbol{A}_{3(n-1)}\boldsymbol{A}_{3(n-1)-1}\dots\boldsymbol{A}_m\dots\boldsymbol{A}_2\boldsymbol{A}_1\\
&= \prod_{m=1}^{3(n-1)}\boldsymbol{A}_m && \text{(A.29)}
\end{aligned}$$

On a donc

$$\begin{aligned} \boldsymbol{A}^* &= 2((\boldsymbol{I}-\boldsymbol{\Phi}_{in})^{-1})^\top - \boldsymbol{I} \\ &= 2\prod_{m=1}^{3(n-1)} \boldsymbol{A}_m - \boldsymbol{I} \end{aligned}$$

Soient $\boldsymbol{W}_c^*$ le grammien de commandabilité défini d'après la paramétrisation $(\boldsymbol{A}^*, \boldsymbol{b}^*, \boldsymbol{c}^*, d)$, et $s_k^2 \triangleq (\boldsymbol{W}_c^*)_{k,k}$. On peut montrer (voir [67]) que, généralement, la paramétrisation $(\boldsymbol{A}^*, \boldsymbol{b}^*, \boldsymbol{c}^*, d)$ n'est pas $\mathcal{L}_2$-échelonnée, c'est-à-dire qu'il existe un entier k tel que $s_k^2 \neq 1$. Li *et al.* appliquent alors une dernière transformation pour obtenir une paramétrisation $\mathcal{L}_2$-échelonnée ($\mathcal{L}_2$-*scaled*). La transformation appliquée est $\boldsymbol{T}_s = diag\{s_1, s_2, \ldots, s_n\}$ ($(\boldsymbol{T}_s)_{i,j} = s_i$ si $j = i$ et $(\boldsymbol{T}_s)_{i,j} = 0$ sinon), on obtient ainsi $(\tilde{A}, \tilde{B}, \tilde{C}, d)$:

$$\begin{aligned} \tilde{\boldsymbol{A}} &\triangleq \boldsymbol{T}_s^{-1}\boldsymbol{A}^*\boldsymbol{T}_s = 2\prod_{m=1}^{3(n-1)} \tilde{\boldsymbol{A}}_m - I \\ \tilde{\boldsymbol{b}} &\triangleq \boldsymbol{T}_s^{-1}\boldsymbol{c}_{in}^\top \\ \tilde{\boldsymbol{c}} &\triangleq \sqrt{2}s_n\boldsymbol{k}_{in}^\top \end{aligned}$$

où $\tilde{\boldsymbol{A}}_m = \boldsymbol{T}_s^{-1}\boldsymbol{A}_m\boldsymbol{T}_s$, $\forall m$. Le vecteur $\tilde{\boldsymbol{c}}$ a la même structure creuse que le vecteur $\boldsymbol{k}_{in}$ (un seul élément non nul, voir équation (A.11)). De plus, $\tilde{\boldsymbol{A}}_m$ a la même structure que $\boldsymbol{A}_m$, où les coefficients $\{\alpha_k, \beta_k, \gamma_k\}$ sont remplacés par les coefficients $\{\tilde{\alpha}_k, \tilde{\beta}_k, \tilde{\gamma}_k\}$ avec

$$\begin{aligned} \tilde{\alpha}_k &\triangleq s_{k+1}^{-1}s_k\alpha_k, \\ \tilde{\beta}_k &\triangleq s_k^{-1}s_{k+1}\beta_k, \\ \tilde{\gamma}_k &\triangleq \gamma_k, \end{aligned}$$

pour $1 \leqslant k \leqslant n-1$.

Finalement, on obtient la structure suivante :

$$\begin{cases} \boldsymbol{x}^{(0)}(k) & = 2\boldsymbol{x}(k) \\ \boldsymbol{x}^{(m)}(k) & = \tilde{\boldsymbol{A}}_m\boldsymbol{x}^{(m-1)}(k), \quad k = 1, 2, \ldots, N \\ \boldsymbol{x}(k+1) & = \boldsymbol{x}^{(N)}(k) - \boldsymbol{x}(k) + \tilde{\boldsymbol{b}}u(k), \\ y(k) & = \tilde{\boldsymbol{c}}\boldsymbol{x}(k) + du(k), \end{cases}$$

avec $N \triangleq 3(n-1)$. Cette structure est appelée structure LCW. De même que pour la structure LGS, l'utilisation de variables intermédiaires $\boldsymbol{x}^{(m)}(k)$ fait que cette structure n'est pas un state-space.

Les transformations nécessaires pour obtenir une LGS ou une LCW paraissent bien compliqués, toutefois il y a un réel intérêt à obtenir une réalisation d'espace d'état creuse (avec peu de coefficients, $5(n-1)$ pour la LGS et $4n$

pour la LCW) et présentant une faible sensibilité vis-à-vis de la quantification des coefficients. Nous mettrons cela en évidence dans le chapitre 6 et l'exemple final.

Annexe B

Codes

B.1 Code de l'exemple 4.5.1

Code B.1 – Code `C` du calcul de x_1 sur 32 bits.

```
int16_t C_int_x1(int16_t v0,int16_t v1,
        int16_t v2,int16_t v3)
{       // Registers declaration
        int32_t r0, r1;
        // Computation of c3*v3 in r0
        r0 = -31852*v3;
        r0 = r0 >> 3;
        // Computation of c0*v0 in r1
        r1 = 17455*v0;
        r1 = r1 << 6;
        // Computation of r0+r1 in r0
        r0 = r0 + r1;
        // Computation of c1*v1 in r1
        r1 = 20349*v1;
        r1 = r1 << 11;
        // Computation of r0+r1 in r0
        r0 = r0 + r1;
        // Computation of c2*v2 in r1
        r1 = 29059*v2;
        r1 = r1 << 11;
        // Computation of r0+r1 in r0
        r0 = r0 + r1;
        // The result is returned with
        // a final right shift
        return r0 >> 16;
}
```

Code B.2 – Code `C` du calcul de x_2 sur 32 bits.

```
int16_t C_int_x2(int16_t v0,int16_t v1,
        int16_t v2,int16_t v3)
{       // Registers declaration
        int32_t r0, r1;
        // Computation of c3*v3 in r0
        r0 = -23042*v3;
        r0 = r0 >> 6;
        // Computation of c0*v0 in r1
        r1 = -19089*v0;
        r1 = r1 << 6;
        // Computation of r0+r1 in r0
        r0 = r0 + r1;
        // Computation of c1*v1 in r1
        r1 = -22265*v1;
        r1 = r1 << 11;
        // Computation of r0+r1 in r0
        r0 = r0 + r1;
        // Computation of c2*v2 in r1
        r1 = -31795*v2;
        r1 = r1 << 11;
        // Computation of r0+r1 in r0
        r0 = r0 + r1;
        // The result is returned with
        // a final right shift
        return r0 >> 16;
}
```

Code B.3 – Code C du calcul de x_3 sur 32 bits.

```
int16_t C_int_x3(int16_t v0,int16_t v1,
        int16_t v2,int16_t v3)
{       // Registers declaration
        int32_t r0, r1;
        // Computation of c3*v3 in r0
        r0 = 18933*v3;
        r0 = r0 >> 6;
        // Computation of c0*v0 in r1
        r1 = 26083*v0;
        r1 = r1 << 5;
        // Computation of r0+r1 in r0
        r0 = r0 + r1;
        // Computation of c1*v1 in r1
        r1 = 30432*v1;
        r1 = r1 << 10;
        // Computation of r0+r1 in r0
        r0 = r0 + r1;
        // Computation of c2*v2 in r1
        r1 = 21729*v2;
        r1 = r1 << 11;
        // Computation of r0+r1 in r0
        r0 = r0 + r1;
        // The result is returned with
        // a final right shift
        return r0 >> 16;
}
```

Code B.4 – Code C du calcul de y sur 32 bits.

```
int16_t C_int_y(int16_t v0,int16_t v1,
        int16_t v2,int16_t v3)
{       // Registers declaration
        int32_t r0, r1;
        // Computation of c3*v3 in r0
        r0 = -19744*v3;
        r0 = r0 << 1;
        // Computation of c0*v0 in r1
        r1 = 30060*v0;
        r1 = r1 << 3;
        // Computation of r0+r1 in r0
        r0 = r0 + r1;
        // Computation of c1*v1 in r1
        r1 = 17267*v1;
        r1 = r1 << 8;
        // Computation of r0+r1 in r0
        r0 = r0 + r1;
        // Computation of c2*v2 in r1
        r1 = 24028*v2;
        r1 = r1 << 8;
        // Computation of r0+r1 in r0
        r0 = r0 + r1;
        // The result is returned with
        // a final right shift
        return r0 >> 16;
}
```

Annexe C

Preuves

C.1 Preuve de la proposition 3.1 (page 79)

Le filtre $\mathcal{H}$ est caractérisé par un state-space de paramétrisation $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$, sa réponse impulsionnelle $\boldsymbol{h}$ peut donc être définie par cette paramétrisation de la façon suivante :

$$\boldsymbol{h}(k) = \begin{cases} \boldsymbol{0} & \text{si } k < 0 \\ \boldsymbol{D} & \text{si } k = 0 \\ \boldsymbol{C}\boldsymbol{A}^{k-1}\boldsymbol{B} & \text{sinon} \end{cases} \tag{C.1}$$

Par définition de la sortie $\boldsymbol{y}$, on a $\boldsymbol{y} = \boldsymbol{h} * \boldsymbol{u}$, soit à tout instant k :

$$\boldsymbol{y}(k) = \sum_{\ell=-\infty}^{+\infty} \boldsymbol{h}(\ell)\boldsymbol{u}(k-\ell). \tag{C.2}$$

On peut donc calculer le moment de premier ordre de $\boldsymbol{y}$, en utilisant la définition 1.2 :

$$\begin{aligned} \boldsymbol{\mu}_{\boldsymbol{y}} = E\{\boldsymbol{y}(k)\} &= E\left\{\sum_{\ell=1}^{+\infty} \boldsymbol{C}\boldsymbol{A}^{\ell-1}\boldsymbol{B}\boldsymbol{u}(k-\ell) + \boldsymbol{D}\boldsymbol{u}(k)\right\} && \text{(C.3)} \\ &= \sum_{\ell=0}^{+\infty} \boldsymbol{C}\boldsymbol{A}^{\ell}\boldsymbol{B}E\left\{\boldsymbol{u}(k-\ell-1)\right\} + \boldsymbol{D}E\left\{\boldsymbol{u}(k)\right\} && \text{(C.4)} \\ &= \left(\sum_{\ell=0}^{+\infty} \boldsymbol{C}\boldsymbol{A}^{\ell}\boldsymbol{B} + \boldsymbol{D}\right)\boldsymbol{\mu}_{\boldsymbol{u}} && \text{(C.5)} \end{aligned}$$

Or par l'équation (1.33) on a $\langle\!\langle \mathcal{H} \rangle\!\rangle_{\mathrm{DC}} = \boldsymbol{C}(\boldsymbol{I}_n - \boldsymbol{A})^{-1}\boldsymbol{B} + \boldsymbol{D}$, que l'on peut écrire $\boldsymbol{C} \sum_{\ell=-\infty}^{+\infty} \boldsymbol{A}^{\ell}\boldsymbol{B} + \boldsymbol{D}$, on a donc bien

$$\boldsymbol{\mu}_{\boldsymbol{y}} = \langle\!\langle \mathcal{H} \rangle\!\rangle_{\mathrm{DC}} \cdot \boldsymbol{\mu}_{\boldsymbol{u}} \tag{C.6}$$

On calcule maintenant $\boldsymbol{\Psi_y}$, toujours d'après la définition 1.2 :

$$\begin{aligned}\boldsymbol{\Psi_y} &= E\left\{(\boldsymbol{y}(k)-\boldsymbol{\mu_y})(\boldsymbol{y}(k)-\boldsymbol{\mu_y})^\top\right\}\\ &= E\left\{\left(\sum_{\ell=0}^{+\infty}\boldsymbol{h}(\ell)\boldsymbol{u}(k-\ell)-\boldsymbol{\mu_y}\right)\left(\sum_{m=0}^{+\infty}\boldsymbol{h}(m)\boldsymbol{u}(k-m)-\boldsymbol{\mu_y}\right)^\top\right\}\end{aligned}$$

On peut alors remplacer $\boldsymbol{\mu_y}$ par $\sum_{i=0}^{+\infty}\boldsymbol{h}(i)\boldsymbol{\mu_u}$ et factoriser, on obtient :

$$\boldsymbol{\Psi_y} = E\left\{\left(\sum_{\ell=0}^{+\infty}\boldsymbol{h}(\ell)\Big(\boldsymbol{u}(k-\ell)-\boldsymbol{\mu_u}\Big)\right)\left(\sum_{m=0}^{+\infty}\boldsymbol{h}(m)\Big(\boldsymbol{u}(k-m)-\boldsymbol{\mu_u}\Big)\right)^\top\right\} \tag{C.7}$$

qui, en utilisant la linéarité de l'espérance, devient :

$$\boldsymbol{\Psi_y} = \sum_{\ell=0}^{+\infty}\sum_{m=0}^{+\infty}\boldsymbol{h}(\ell)E\left\{\Big(\boldsymbol{u}(k-\ell)-\boldsymbol{\mu_u}\Big)\Big(\boldsymbol{u}(k-m)-\boldsymbol{\mu_u}\Big)^\top\right\}\boldsymbol{h}(m)^\top \tag{C.8}$$

Par hypothèse, $\boldsymbol{u}$ est décorrélé, on a donc :

$$\boldsymbol{\Psi_y} = \sum_{\ell=0}^{+\infty}\sum_{m=0}^{+\infty}\boldsymbol{h}(\ell)\delta_{\ell,m}\boldsymbol{\Psi_u}\boldsymbol{h}(m)^\top \tag{C.9}$$

$$= \sum_{\ell=0}^{+\infty}\boldsymbol{h}(\ell)\boldsymbol{\Psi_u}\boldsymbol{h}(\ell)^\top \tag{C.10}$$

$$= \boldsymbol{D}\boldsymbol{\Psi_u}\boldsymbol{D}^\top + \sum_{\ell=0}^{+\infty}\boldsymbol{C}\boldsymbol{A}^\ell\boldsymbol{B}\boldsymbol{\Psi_u}\boldsymbol{B}^\top\boldsymbol{A}^{\ell^\top}\boldsymbol{C}^\top \tag{C.11}$$

On peut alors, à partir de cette dernière expression, calculer $\boldsymbol{\sigma_y^2}$, qui par définition est la trace de $\boldsymbol{\Psi_y}$:

$$\boldsymbol{\sigma_y^2} = \mathrm{tr}(\boldsymbol{\Psi_y}) \tag{C.12}$$

$$= \mathrm{tr}\left(\boldsymbol{D}\boldsymbol{\Psi_u}\boldsymbol{D}^\top\right) + tr\left(\boldsymbol{\Psi_u}\boldsymbol{B}^\top\sum_{\ell=0}^{+\infty}\boldsymbol{A}^{\ell^\top}\boldsymbol{C}^\top\boldsymbol{C}\boldsymbol{A}^\ell\boldsymbol{B}\right) \tag{C.13}$$

or par définition du grammien d'observabilité (équation (1.26)), on a :

$$\boldsymbol{\sigma_y^2} = \mathrm{tr}\left(\boldsymbol{D}\boldsymbol{\Psi_u}\boldsymbol{D}^\top\right) + tr\left(\boldsymbol{\Psi_u}\boldsymbol{B}^\top\boldsymbol{W_o}\boldsymbol{B}\right) \tag{C.14}$$

On utilise maintenant la décomposition de Choleski de $\boldsymbol{\Psi_u}$, c'est-à-dire $\boldsymbol{\psi_u} = \boldsymbol{\varphi_u}\boldsymbol{\varphi_u}^\top$, on peut ainsi écrire :

$$\boldsymbol{\sigma_y^2} = \mathrm{tr}\left((\boldsymbol{D}\boldsymbol{\varphi_u})(\boldsymbol{D}\boldsymbol{\varphi_u})^\top + (\boldsymbol{B}\boldsymbol{\varphi_u})^\top\boldsymbol{W_o}(\boldsymbol{B}\boldsymbol{\varphi_u})\right) \tag{C.15}$$

et donc $\boldsymbol{\sigma_y^2} = \|\mathcal{H}\cdot\boldsymbol{\varphi_u}\|_2^2$ d'après l'équation (1.30).

C.2 Preuve de la proposition 3.2 (page 80)

La preuve est faite ici dans le cas SISO pour simplifier les notations mais se généralise aisément au cas MIMO.

Soit $\langle u_m, u_r \rangle$ l'intervalle des valeurs prises par le signal u, c'est-à-dire $u(k) \in \langle u_m, u_r \rangle$, pour tout $k \geqslant 0$, et on veut déterminer y_m et y_r tels que $y(k) \in \langle y_m, y_r \rangle$ pour tout $k \geqslant 0$. D'après la proposition 3.1, on sait que la moyenne μ_y des $y(k)$ pour tout $k > 0$ est égale à $\langle\!\langle \mathcal{H} \rangle\!\rangle_{\mathrm{DC}} \cdot u_m$. En prenant μ_y comme centre de l'intervalle $\langle y_m, y_r \rangle$, c'est-à-dire $y_m = \mu_u$, on cherche y_r tel que $y(k) \in \langle y_m, y_r \rangle$ pour tout $k > K$ où K dépend d'un certains ε.

On notera que par définition du DC-gain, on a $y_m = \sum_{\ell=0}^{+\infty} h(\ell) \cdot u_m$.

D'après l'équation (1.10) définissant la relation entre l'entrée et la sortie via la réponse impulsionnelle, on a :

$$\begin{aligned}
y(k) &= \sum_{\ell=0}^{k} h(\ell) \cdot u(k-\ell) \\
y(k) &= \sum_{\ell=0}^{k} h(\ell) \cdot u_m + \sum_{\ell=0}^{k} h(\ell) \cdot (u(k-\ell) - u_m) \\
y(k) - \sum_{\ell=0}^{+\infty} h(\ell) \cdot u_m &= \sum_{\ell=0}^{k} h(\ell) \cdot (u(k-\ell) - u_m) - \sum_{\ell=k+1}^{+\infty} h(\ell) \cdot u_m \\
|y(k) - y_m| &\leqslant \sum_{\ell=0}^{k} |h(\ell)| \cdot u_r + \left| \sum_{\ell=k+1}^{+\infty} h(\ell) \cdot u_m \right| \\
|y(k) - y_m| &\leqslant (\sum_{\ell=0}^{+\infty} |h(\ell)| - \sum_{\ell=k+1}^{+\infty} |h(\ell)|) \cdot u_r + \sum_{\ell=k+1}^{+\infty} |h(\ell)| \cdot |u_m| \\
|y(k) - y_m| &\leqslant \langle\!\langle \mathcal{H} \rangle\!\rangle_{\mathrm{wcpg}} \cdot u_r + \sum_{\ell=k+1}^{+\infty} |h(\ell)| \cdot (|u_m| - u_r). \qquad \text{(C.16)}
\end{aligned}$$

On pose alors $\varepsilon_k = \sum_{\ell=k+1}^{+\infty} |h(\ell)| \cdot (|u_m| - u_r)$.

Ainsi, le membre de gauche de l'inéquation (C.16) correspond à la distance entre le signal de sortie y à un instant k et la sortie moyenne y_m. De plus, dans le membre de droite, seul ε_k dépend de k. On souhaite donc, pour un ε donné, calculer le plus petit $K \in \mathbb{N}$ tel que pour tout $k \geqslant K$ on a

$$\langle\!\langle \mathcal{H} \rangle\!\rangle_{\mathrm{wcpg}} \cdot u_r + \varepsilon \geqslant |y(k) - y_m| \,, \qquad \text{(C.17)}$$

c'est-à-dire calculer K tel que $\langle\!\langle \mathcal{H} \rangle\!\rangle_{\mathrm{wcpg}} \cdot u_r + \varepsilon$ majore y_r, avec $\varepsilon = \varepsilon_K$.

Pour cela, on suppose que le filtre $\mathcal{H}$ est décrit par un state-space $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, d)$, que $\boldsymbol{A}$ est diagonalisable et qu'on a la décomposition en éléments propres de $\boldsymbol{A}$ suivante :

$$\boldsymbol{A} = \boldsymbol{X}\boldsymbol{E}\boldsymbol{X}^{-1}, \qquad \text{(C.18)}$$

où $\boldsymbol{E}$ est la matrice diagonale composée des valeurs propres de $\boldsymbol{A}$ et $\boldsymbol{X}$ est la matrice composée des vecteurs propres de $\boldsymbol{A}$. On a alors :

$$\boldsymbol{c}\boldsymbol{A}^k\boldsymbol{b} = \boldsymbol{c}\boldsymbol{X}\boldsymbol{E}^k\boldsymbol{X}^{-1}\boldsymbol{b} \tag{C.19}$$

$$= \sum_{i=1}^{n} r_i\lambda_i^\ell \tag{C.20}$$

avec

$$\boldsymbol{R}_i \triangleq (\boldsymbol{c}\boldsymbol{X})_i(\boldsymbol{X}^{-1}\boldsymbol{b})_i, \tag{C.21}$$

et λ_i la i-ème valeur propre de $\boldsymbol{A}$, pour $1 \leqslant i \leqslant n$.

On a donc

$$\sum_{\ell=k+1}^{+\infty} |h(\ell)| = \sum_{\ell=k+1}^{+\infty} \left|\boldsymbol{c}\boldsymbol{A}^\ell\boldsymbol{b}\right| \tag{C.22}$$

$$= \sum_{\ell=k+1}^{+\infty} \left|\sum_{i=1}^{n} r_i\lambda_i^\ell\right| \tag{C.23}$$

Or, Volkova *et al.* ont montré dans [104] que cette somme pouvait se majorer par $\rho(A)^{k+1} \cdot m$ avec

$$m \triangleq \sum_{\ell=1}^{n} \frac{|r_\ell|}{1-|\lambda_\ell|}\frac{|\lambda_\ell|}{\rho(A)}. \tag{C.24}$$

Ainsi, en injectant $(|u_m| - u_r)$ dans l'équation (C.23), on obtient :

$$\sum_{\ell=k+1}^{+\infty} |h(\ell)| \cdot (|u_m| - u_r) \leqslant \rho(A)^{k+1} \cdot m \cdot (|u_m| - u_r) \tag{C.25}$$

Et puisqu'on veut que le membre de gauche soit plus petit qu'un ε donné, on a :

$$\rho(A)^{k+1} \cdot m \cdot (|u_m| - u_r) \leqslant \varepsilon \tag{C.26}$$

dont on peut déduire K, le plus petit entier tel que k qui vérifie l'équation (C.26) pour tout $k > K$:

$$K \geqslant \left\lceil \left| \frac{\log \frac{\varepsilon}{m\cdot(|u_m|-u_r)}}{\log \rho(A)} \right| \right\rceil. \tag{C.27}$$

C.3 Preuve de la proposition 4.2 (page 103)

Cette preuve utilise le lemme suivant.

Lemme 1. *Soient L une liste contenant des oSoPs, créée dans l'algorithme 3, et i un indice de L ($L[i]$ est un oSoP). Si $mg(L[i]) < mg(L[1])$, alors $L[1]$ est un oSoP provenant de deux sous-oSoPs générés à l'étape précédente de l'algorithme.*

Démonstration. Du fait de sa position (premier élément de la liste L), l'élément $L[1]$ est soit un produit soit un oSoP. S'il est un produit alors L est la liste initiale des produits triés, et donc $L[1]$ est p_1, le produit d'indice 1, et donc on aurait $mg(L[i]) \geqslant mg(L[1])$ pour tout $1 \leqslant i \leqslant Longueur(L)$. Donc $L[1]$ est un oSoP, et puisqu'il est en position 1 c'est qu'il a été généré à l'étape précédente de l'algorithme, étant donné que les oSoPs sont insérés à chaque étape en début de liste. □

On peut maintenant démontrer la proposition 4.2.

L'algorithme génère tous les oSoPs : Sans considérer la condition $mg(L[i]) \geqslant mg(L[1])$, l'algorithme génère toutes les paires possibles à partir de liste de sous-oSoPs, qui sont soit des arbres de tailles inférieures à celle des oSoPs finaux, soit des multiplieurs. Toutes les combinaisons sont prises en compte, donc on génère *au moins* tous les oSoPs.

La condition $mg(L[i]) \geqslant mg(L[1])$ garantie l'absence de doublons : On suppose, et on note H_1 cette hypothèse, qu'on a généré tous les oSoPs qui pouvait l'être à partir de la condition $mg(L[i]) \geqslant mg(L[1])$, pour toute liste L et tout indice $1 \leqslant i \leqslant Longueur(L)$. On cherche alors à démontrer que si, pour une sous-liste L et un indice i, on ne respecte pas la condition $mg(L[i]) \geqslant mg(L[1])$, alors la liste créée aura déjà été créée.

Soient, à une étape de l'algorithme, une liste L contenant des sous-oSoPs, et $i > 1$ un indice de L, tels que $mg(L[i]) < mg(L[1])$. On note cette hypothèse H_2. On applique alors la suite de l'algorithme, soit j tel que $i < j \leqslant Longueur(L)$, et on additionne les termes $L[i]$ et $L[j]$, la liste obtenue est notée L_d :

$$L_d \triangleq [(L[i] + L[j]), L[1], \ldots] \tag{C.28}$$

où le reste de la liste L_d est composé des autres éléments de L, c.-à-d. $L[k]$ pour $k \in \{2, \ldots, Longueur(L)\} \backslash \{i, j\}$.

D'après le lemme 1, à l'étape précédente de l'algorithme, on avait une liste L' qui a engendré la liste L et telle que

$$L[1] = (L'[k] + L'[\ell]) \tag{C.29}$$

pour $1 \leqslant k < \ell \leqslant Longueur(L')$. De plus, étant donné qu'à une étape on ne crée qu'une paire, il existe i_0 (resp. j_0) tel que $L[i] = L'[i_0]$ (resp. $L[j] = L'[j_0]$) avec $k \neq i_0 \neq \ell \neq j_0$ (sinon on ne pourrait pas avoir les éléments $L[1]$, $L[i]$ et $L[j]$ tous différents).

On peut alors différencier deux cas, soit $mg(L'[i_0]) < mg(L'[1])$, soit $mg(L'[i_0]) \geqslant mg(L'[1])$. Or, pour la construction de la liste L', par l'hypothèse H_1, la condition $mg(L'[i]) \geqslant mg(L'[1])$ était respectée, donc le premier cas n'était pas possible. Le second cas signifie qu'on pouvait créer des paires à partir de l'indice i_0 puisqu'il respectait la condition. En particulier la paire d'indice (i_0, j_0) a été considérée, et à donné la liste L_t suivante :

$$L_t \triangleq [(L'[i_0] + L'[j_0]), \ldots] \tag{C.30}$$

où le reste de la liste L_t est composé des autres éléments de L', en particulier $L'[k]$ et $L'[\ell]$. Il existe donc k_t et ℓ_t tels que

$$L_t[k_t] = L'[k], \tag{C.31}$$
$$L_t[\ell_t] = L'[\ell], \tag{C.32}$$
$$\tag{C.33}$$

et par construction de L_t on a :

$$L_t[1] = (L'[i_0] + L'[j_0]) = (L[i] + L[j]). \tag{C.34}$$

Or, d'après l'hypothèse H_2, $mg(L[i]) < mg(L[1])$, et de plus $mg(L[1]) = mg(L'[k])$ (d'après l'équation C.29 et l'hypothèse H_1 qui était respectée), donc $mg(L[i]) < mg(L'[k])$, qui s'écrit, par les équations (C.31) et (C.34) :

$$mg(L_t[1]) < mg(L_t[k_t]), \tag{C.35}$$

ce qui respecte la condition pour la liste L_t et l'indice k_t. On peut donc créer des paires à partir de l'indice k_t et de la liste L_t, en particulier la paire d'indice (k_t, ℓ_t) a été considérée, et à donné la liste L_f (liste *finale* qui parachèvera cette démonstration) suivante :

$$L_f \triangleq [(L_t[k_t] + L_t[\ell_t]), L_t[1], \ldots]. \tag{C.36}$$

Or, d'après les équations (C.29), (C.31) et (C.32), on a $(L_t[k_t] + L_t[\ell_t]) = L[1]$, et d'après l'équation (C.34) on a $L_t[1] = (L[i] + L[j])$, donc L_f peut s'écrire :

$$L_f = [L[1], (L[i] + L[j]), \ldots]. \tag{C.37}$$

On obtient alors que la liste L_f, qui avait déjà été généré d'après l'hypothèse H_1, est, à permutation près des ses deux premiers éléments, la liste L_d. Donc L_d est bien une liste *doublon* d'une liste déjà générée.

Remarque C.1. *En fait pour être tout à fait rigoureux, il aurait fallu montrer que les restes (cachés dans les "...") des listes L_d et L_f sont biens les mêmes. Ceci est évident puisque les deux listes découlent toutes deux de la liste L' (L' a engendré L qui a engendré et L_d par l'hypothèse H_2, et L' a engendré L_t qui a engendré et L_f par l'hypothèse H_1). En reprenant les notations, les deux restes sont composés de $L'[m]$ pour $m \in \{1, \ldots, Longueur(L')\} \backslash \{i_0, j_0, k, \ell\}$.*

C.4 Preuve de la proposition 5.1 (page 136)

C.4.1 Arrondi par troncature

Démonstration. On souhaite calculer une somme intermédiaire

$$s_\delta \triangleq \sum_{i=1}^{N} \nabla_{\ell_s-\delta-1}(p_i) \tag{C.38}$$

pour un $\delta > 0$ arbitraire, et déterminer ce δ pour vérifier $s_{opt} - s' \leqslant 2^{\ell_s}$, où $s' = \nabla_{\ell_s}(s_\delta)$.

Pour calculer s_δ on tronque des bits des termes p_i, donc *a fortiori* :

$$s_{opt} \geqslant s_\delta \tag{C.39}$$

Le cas $s_{opt} = s_\delta$ est trivial car il implique $s = s'$, on considère donc uniquement le cas $s_{opt} > s_\delta$. La différence $s_{opt} - s_\delta$ correspond à la somme des bits des p_i qui ont été tronqués pour calculer s_δ, c.-à-d. :

$$s_{opt} - s_\delta \triangleq \sum_{i=1}^{N} p_i - \nabla_{\ell_s-\delta}(p_i) = \sum_{i=1}^{N} \sum_{j=L}^{\ell_s-\delta-1} 2^j p_{i,j}, \tag{C.40}$$

où $L = \min(\ell_i)$, en supposant $p_{i,j} = 0$ pour $j < \ell_i$.
Puisque $\sum_{j=L}^{\ell_s-\delta-1} 2^j p_{i,j} < 2^{\ell_s-\delta}$ pour $1 \leqslant i \leqslant N$, on a :

$$s_{opt} - s_\delta < N \cdot 2^{\ell_s-\delta} \tag{C.41}$$

De plus, puisque par définition $s' = \nabla_{\ell_s}(s_\delta)$, on a :

$$s_\delta - s' < 2^{\ell_s}. \tag{C.42}$$

En fait on peut être plus précis sur cette majoration car on sait que le *lsb* de s_δ est $\ell_s - \delta$, on a donc :

$$s_\delta - s' \leqslant 2^{\ell_s} - 2^{\ell_s-\delta}. \tag{C.43}$$

En ajoutant terme à terme les équations (C.41) et (C.43), on a :

$$s_{opt} - s' < 2^{\ell_s} + (N-1) \cdot 2^{\ell_s-\delta} \tag{C.44}$$

Pour avoir $s' = s$, c.-à-d. $\nabla_{\ell_s}(s_\delta) = \nabla_{\ell_s}(s_{opt})$, il faudrait majorer le terme droit de l'équation (C.44) par 2^{ℓ_s}, ce qui n'est pas possible (dans l'optique de supprimer des bits), et on a vu qu'il était facile d'exhiber un exemple montrant qu'il était impossible de garantir $s' = s$. On veut donc garantir dans le pire des cas $s' = \nabla_{\ell_s}(s_{opt}) - 2^{\ell_s}$, c.-à-d. le nombre FxP qui précède immédiatement s sur le format (m_s, ℓ_s), et pour cela on majore le terme droit de l'équation par 2^{ℓ_s+1} :

$$\begin{aligned} 2^{\ell_s} + (N-1) \cdot 2^{\ell_s - \delta} &< 2^{\ell_s + 1} \\ (N-1) \cdot 2^{-\delta} &< 1 \\ \log_2(N-1) &< \delta \end{aligned} \tag{C.45}$$

Le plus petit entier δ qui respecte l'équation (C.45) est donné par :

$$\delta \triangleq \lfloor \log_2(N-1) \rfloor + 1. \tag{C.46}$$

□

C.4.2 Arrondi au plus proche

Démonstration. Par définition de $\circ_d(x)$, pour $x \in \mathbb{R}^*$ et $d \in \mathbb{N}$, on a (voir le tableau 2.4 et l'arrondi au plus proche en précision infinie) :

$$x - \circ_d(x) \in [-2^{d-1}; 2^{d-1}[. \tag{C.47}$$

En réalité si x est un nombre virgule fixe (ce qui nous intéresse ici) dont on arrondit k bits, alors on a (voir le tableau 2.4 et l'arrondi au plus proche en précision finie) :

$$x - \circ_d(x) \in [-2^{d-1}; 2^{d-1} - 2^{d-k}]. \tag{C.48}$$

Dans cette preuve, une telle précision sur la borne supérieure est inutile puisque en valeur absolue la borne inférieure est plus grande et atteignable, c'est donc elle que l'on va vouloir majorer. Pour les termes à sommer p_i, on a donc :

$$p_i - \circ_{\ell_s - \delta}(p_i) \in [-2^{\ell_s - \delta - 1}; 2^{\ell_s - \delta - 1}[, \tag{C.49}$$

et par définition de s_{opt} et s_δ :

$$s_{opt} - s_\delta \in [-N \cdot 2^{\ell_s - \delta - 1}; N \cdot 2^{\ell_s - \delta - 1}[. \tag{C.50}$$

De même, par définition de s', arrondi de s_δ au ℓ_s-ième bit, il vient :

$$s_\delta - s' \in [-2^{\ell_s - 1}; 2^{\ell_s - 1}[\tag{C.51}$$

En ajoutant terme à terme (et en utilisant l'arithmétique d'intervalle, ou en combinant les précédentes équations, ce qui revient au même) les équations (C.50) et (C.51), on obtient :

$$s_{opt} - s' \in [-2^{\ell_s - 1} - N \cdot 2^{\ell_s - \delta - 1}; 2^{\ell_s - 1} + N \cdot 2^{\ell_s - \delta - 1}[. \tag{C.52}$$

Or on veut que s' soit un arrondi fidèle de s_{opt}, c.-à-d. :

$$\left| s_{opt} - s' \right| < 2^{\ell_s}, \tag{C.53}$$

on doit déterminer δ qui vérifie la majoration suivante :

$$\begin{aligned} 2^{\ell_s-1} + N \cdot 2^{\ell_s-\delta-1} &< 2^{\ell_s} \\ N \cdot 2^{-\delta-1} &< 1 \\ \log_2(N) &< \delta \end{aligned} \tag{C.54}$$

Le plus petit entier δ qui respecte l'équation (C.54) est donné par :

$$\delta \triangleq \lfloor \log_2(N) \rfloor + 1. \tag{C.55}$$

□

Bibliographie

[1] A Ahmadi et M. Zwolinski. "Area word-length trade off in DSP algorithm implementation and optimization". Dans : *The 2nd IEE/EURASIP Conference on DSPenabledRadio, 2005 (Ref. No. 2005/11086)*. Sept. 2005, 8 pp.–.

[2] A Ahmadi et M. Zwolinski. "A Symbolic Noise Analysis Approach to Word-Length Optimization in DSP Hardware". Dans : *International Symposium on Integrated Circuits. ISIC '07*. Sept. 2007, p. 457–460.

[3] A Ahmadi et M. Zwolinski. "Word-Length Oriented Multiobjective Optimization of Area and Power Consumption in DSP Algorithm Implementation". Dans : *25th International Conference on Microelectronics*. 2006, p. 614–617.

[4] M. Ahmad. "Implementable Decimal Arithmetic Algorithms for Micro/Minicomputers". Dans : *Microprocess. Microprogram.* 19.2 (fév. 1987), p. 119–128.

[5] P. J. Ashenden. *The Designer's Guide to VHDL, Volume 3, Third Edition (Systems on Silicon)*. 3e éd. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2008.

[6] V. Balakrishnan et S. Boyd. "On computing the worst-case peak gain of linear systems". Dans : *Proceedings of the 31st IEEE Conference on Decision and Control, 1992*. 1992, 2191–2192 vol.2.

[7] J.E. Bertram. "The effects of quantization in sampled-feedback systems". Dans : *Trans. of the American Institute of Electrical Engineers* 77 (1958), p. 177–182.

[8] J.-L. Beuchat. "Etude et conception d'opérateurs arithmétiques optimisés pour circuits programmables". Thèse de doct. Ecole Polythechnique Fédérale de Lausanne, 2001.

[9] P. Bonami et al. “An Algorithmic Framework for Convex Mixed Integer Nonlinear Programs”. Dans : *Discret. Optim.* 5.2 (mai 2008), p. 186–204.

[10] A. Booth. “A Signed Binary Multiplication Technique”. Dans : *Quarterly Journal of Mechanics and Applied Mathematics* 4.2 (juin 1951), p. 236–240.

[11] N. Brunie et al. “Arithmetic core generation using bit heaps”. Dans : *Field-Programmable Logic and Applications.* Sept. 2013.

[12] S. Cai et al. “NuMVC : An Efficient Local Search Algorithm for Minimum Vertex Cover”. Dans : *CoRR* abs/1402.0584 (2014).

[13] M.-A. Cantin, Y. Savaria et P. Lavoie. “A comparison of automatic word length optimization procedures”. Dans : *IEEE International Symposium on Circuits and Systems, 2002. ISCAS 2002.* T. 2. 2002,

[14] M.-A. Cantin et al. “A Metric for Automatic Word-Length Determination of Hardware Datapaths”. Dans : *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 25.10 (oct. 2006), p. 2228–2231.

[15] M.-A. Cantin et al. “An automatic word length determination method”. Dans : *The 2001 IEEE International Symposium on Circuits and Systems, 2001. ISCAS 2001.* T. 5. 2001, 53–56 vol. 5.

[16] V. Černý. “Thermodynamical approach to the traveling salesman problem : An efficient simulation algorithm”. Dans : *Journal of Optimization Theory and Applications* 45.1 (1985), p. 41–51.

[17] G.A. Constantinides. “Perturbation analysis for word-length optimization”. Dans : *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.* Avr. 2003, p. 81–90.

[18] G. A. Constantinides, Peter Y. K. Cheung et Wayne Luk. *Synthesis and optimization of DSP algorithms.* Kluwer, 2004, p. I–XI, 1–164.

[19] G. A. Constantinides, P. Y. K. Cheung et W. Luk. “Wordlength optimization for linear digital signal processing”. Dans : *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 22.10 (oct. 2003), p. 1432–1442.

[20] G.A Constantinides, P. Y K Cheung et W. Luk. “The Multiple Wordlength Paradigm”. Dans : *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2001. FCCM ’01.* Mar. 2001, p. 51–60.

[21] G.A. Constantinides, P.Y.K. Cheung et W. Luk. “Optimum wordlength allocation”. Dans : *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2002. Proceedings.* 2002, p. 219–228.

[22] G.A. Constantinides et G.J. Woeginger. “The complexity of multiple wordlength assignment”. Dans : *Applied Mathematics Letters* 15.2 (2002), p. 137 –140.

[23] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. McGraw-Hill Higher Education, 2001.

[24] F. de Dinechin et B. Pasca. “Designing Custom Arithmetic Data Paths with FloPoCo”. Dans : *Design Test of Computers, IEEE* 28.4 (juil. 2011), p. 18–27.

[25] M. A.. Duran et I. E. Grossmann. “An outer-approximation algorithm for a class of mixed-integer nonlinear programs”. English. Dans : *Mathematical Programming* 36.3 (1986), p. 307–339.

[26] M.D. Ercegovac et T. Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.

[27] Thomas A Feo et Mauricio G. C Resende. “A Probabilistic Heuristic for a Computationally Difficult Set Covering Problem”. Dans : *Oper. Res. Lett.* 8.2 (avr. 1989), p. 67–71.

[28] M. Gendreau et J.-Y. Potvin. *Handbook of Metaheuristics*. 2nd. Springer Publishing Company, Incorporated, 2010.

[29] M. Gevers et G. Li. *Parametrizations in Control, Estimation and Filtering Probems*. Springer-Verlag, 1993.

[30] F. Glover. “Tabu Search - Part II”. Dans : *INFORMS Journal on Computing* 2.1 (1990), p. 4–32.

[31] Fred Glover. “Tabu Search - Part I”. Dans : *INFORMS Journal on Computing* 1.3 (1989), p. 190–206.

[32] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1989.

[33] M.S. Grewal et A.P. Andrews. “Applications of Kalman Filtering in Aerospace 1960 to the Present [Historical Perspectives]”. Dans : *Control Systems, IEEE* 30.3 (juin 2010), p. 69–78.

[34] O. K. Gupta et A. Ravindran. “Branch and Bound Experiments in Convex Nonlinear Integer Programming”. Dans : *Management Science* 31.12 (1985), p. 1533–1546. eprint : http://dx.doi.org/10.1287/mnsc.31.12.1533.

[35] L. Hafer. *BonsaiG - Algorithms & Design*. Rap. tech. 1999.

[36] R. Hemmecke et al. “Nonlinear Integer Programming.” Dans : *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*. Springer, 2010, p. 561–618.

[37] N. Hervé. "Contributions à la synthèse d'architecture virgule fixe à largeurs multiples". Thèse de doct. ENSSAT de l'Université de Rennes 1, 2007.

[38] N. Herve, D. Menard et O. Sentieys. "Data wordlength optimization for FPGA synthesis". Dans : *IEEE Workshop on Signal Processing Systems Design and Implementation, 2005*. Nov. 2005, p. 623–628.

[39] T. Hilaire. "Analyse et synthèse de l'implémentation de lois de contrôle-commande en précision finie (Étude dans le cadre des applications automobiles sur calculateur embarquée)". Thèse de doct. Université de Nantes, juin 2006.

[40] T. Hilaire. "On the Transfer Function Error of State-Space Filters in Fixed-Point Context". Dans : *IEEE Transactions on Circuits and Systems II : Express Briefs* 56.12 (déc. 2009), p. 936–940.

[41] T. Hilaire et P. Chevrel. "Sensitivity-based Pole and Input-Output Errors of Linear Filters as Indicators of the Implementation Deterioration in Fixed-Point Context". Dans : *EURASIP Journal on Advances in Signal Processing* special issue on Quantization of VLSI Digital Signal Processing Systems (jan. 2011).

[42] T. Hilaire, P. Chevrel et J-P. Clauzel. "Pole Sensitivity Stability Related Measure of FWL Realization with the Implicit State-Space Formalism". Dans : *5th IFAC Symposium on Robust Control Design (ROCOND'06)*. Juil. 2006.

[43] T. Hilaire, P. Chevrel et Y. Trinquet. "Implicit State-Space Representation : a Unifying Framework for FWL Implementation of LTI Systems". Dans : *Proc. of the 16th IFAC World Congress*. Sous la dir. de P. Piztek. Elsevier, juil. 2005.

[44] T. Hilaire, P. Chevrel et J.F. Whidborne. "A Unifying Framework for Finite Wordlength Realizations". Dans : *IEEE Trans. on Circuits and Systems* 8.54 (août 2007), p. 1765–1774.

[45] T. Hilaire et B. Lopez. "Reliable Implementation of Linear Filters with Fixed-Point Arithmetic". Dans : *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*. 2013.

[46] T. Hinamoto et al. "Analysis and Minimization of L_2-Sensitivity for Linear Systems and Two-Dimensional State-Space Filters Using General Controllability and Observability Gramians". Dans : *IEEE Transactions on Circuits and Systems, Fundamental Theory and Applications*. T. 49. Sept. 2002.

[47] J. H. Holland. *Adaptation in Natural and Artificial Systems*. Cambridge, MA, USA : MIT Press, 1992.

[48] "IEEE Standard for Binary Floating-Point Arithmetic". Dans : *ANSI/IEEE Std 754-1985* (1985).

[49] L. Ingber. “Adaptive simulated annealing (ASA) : Lessons learned”. Dans : *CoRR* cs.MS/0001018 (2000).

[50] L.B. Jackson, J. Kaiser et H. McDonald. “An approach to the implementation of digital filters”. Dans : *IEEE Transactions on Audio and Electroacoustics* 16.3 (sept. 1968), p. 413–421.

[51] HAO J.-K., GALINIER P. et HABIB M. “Métaheuristiques pour l’optimisation combinatoire et l’affectation sous contraintes”. Dans : *Revue d’intelligence artificielle* 13.2 (1999). Sous la dir. de Lavoisier. fre, p. 283–324.

[52] D.A. Johns, W.M. Snelgrove et A. S. Sedra. “Orthonormal ladder filters”. Dans : *Circuits and Systems* 36.3 (mar. 1989), p. 337–343.

[53] Brian W. Kernighan. *The C Programming Language*. Sous la dir. de Dennis M. Ritchie. 2nd. Prentice Hall Professional Technical Reference, 1988.

[54] S. Kirkpatrick, D. Gelatt Jr. et M. P. Vecchi. “Optimization by Simmulated Annealing”. Dans : *Science* 220.4598 (1983), p. 671–680.

[55] K.-I. Kum et W. Sung. “Combined word-length optimization and high-level synthesis of digital signal processing systems”. Dans : *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 20.8 (août 2001), p. 921–930.

[56] K.-I. Kum et W. Sung. “Word-length optimization for high-level synthesis of digital signal processing systems”. Dans : *1998 IEEE Workshop on Signal Processing Systems, 1998. SIPS 98*. Oct. 1998, p. 569–578.

[57] O. Kupriianova, C. Q. Lauter et J.-M. Muller. “Radix conversion for IEEE754-2008 mixed radix floating-point arithmetic”. Dans : *ACSSC*. 2013, p. 1134–1138.

[58] M. Labarrere, JP. Krief et B. Gimonet. *Le filtrage et ses applications*. Cepadues Edition, sept. 1993.

[59] A. H. Land et A. G. Doig. “An Automatic Method of Solving Discrete Programming Problems”. English. Dans : *Econometrica* 28.3 (1960), pp. 497–520.

[60] P. Langlois, M. Martel et L. Thévenoux. “Accuracy Versus Time : A Case Study with Summation Algorithms”. Dans : *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*. PASCO ’10. Grenoble, France : ACM, New York, USA, 2010, p. 121–130.

[61] A.J. Laub et al. “Computation of system balancing transformations and other applications of simultaneous diagonalization algorithms”. Dans : *IEEE Trans. on Automatic Control* 32.2 (fév. 1987), p. 115–122.

[62] M. Leban et J.F. Tasic. “Word-length optimization of LMS adaptive FIR filters”. Dans : *10th Mediterranean Electrotechnical Conference, 2000. MELECON 2000.* T. 2. 2000, 774–777 vol.2.

[63] D.-U. Lee et al. “Accuracy-Guaranteed Bit-Width Optimization”. Dans : *IEEE Trans on Computer-Aided Design of Integrated Circuits and Systems* 25.10 (oct. 2006), p. 1990–2000.

[64] S. Lee et A Gerstlauer. “Fine grain word length optimization for dynamic precision scaling in DSP systems”. Dans : *2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC).* Oct. 2013, p. 266–271.

[65] F.L. Lewis. “A survey of linear singular systems”. English. Dans : *Circuits, Systems and Signal Processing* 5.1 (1986), p. 3–36.

[66] Gang Li. “On pole and zero sensitivity of linear systems”. Dans : *Circuits and Systems I : Fundamental Theory and Applications, IEEE Transactions on* 44.7 (juil. 1997), p. 583–590.

[67] G. Li, J. Chu et J. Wu. “A Matrix Factorization-Based Structure for Digital Filters”. Dans : *Signal Processing* 55.10 (oct. 2007), p. 5108–5112.

[68] G. Li, M. Gevers et Y. Sun. “Performance analysis of a new structure for digital filter implementation”. Dans : *IEEE Trans. on Circuits and Systems I : Fundamental Theory and Applications* 47.4 (avr. 2000), p. 474–482.

[69] G. Li et Z. Zhao. “On the generalized DFIIt structure and its state-space realization in digital filter implementation”. Dans : *IEEE Trans. on Circuits and Systems* 51.4 (avr. 2004), p. 769–778.

[70] B. Lopez, T. Hilaire et L.-S. Didier. “Formatting Bits to Better Implement Signal Processing Algorithms”. Dans : *PECCS 2014 - Proceedings of the 4th International Conference on Pervasive and Embedded Computing and Communication Systems, Lisbon, Portugal, 7-9 January.* 2014, p. 104–111.

[71] B. Lopez, T. Hilaire et L.-S. Didier. “Sum-of-products evaluation schemes with fixed-point arithmetic, and their application to IIR filter implementation”. Dans : *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing, DASIP 2012, Karlsruhe, Germany, October 23-25.* 2012, p. 1–8.

[72] J.A. Lopez, C. Carreras et O. Nieto-Taladriz. “Improved Interval-Based Characterization of Fixed-Point LTI Systems With Feedback Loops”. Dans : *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 26.11 (nov. 2007), p. 1923–1933.

[73] W.J. Lutz et S.L. Hakimi. “Design of multi-input multi-output systems with minimum sensitivity”. Dans : *IEEE Trans. on Circuits and Systems* 35.9 (sept. 1988), p. 1114–1122.

[74] S. J. Mason et H. J. Zimmerman. *Electronic Circuits, Signals, and Systems*. Wiley, 1960.

[75] D. Ménard. “Méthodologie de compilation d’algorithmes de traitement du signal pour les processeurs en virgule fixe sous contrainte de précision”. Thèse de doct. Université Rennes 1, déc. 2002.

[76] D. Menard, R. Rocher et O. Sentieys. “Analytical Fixed-Point Accuracy Evaluation in Linear Time-Invariant Systems”. Dans : *IEEE Transactions on Circuits and Systems I : Regular Papers* 55.10 (nov. 2008), p. 3197–3208.

[77] U. Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays*. Signals and Communication Technology. Springer, 2007.

[78] C. Mouilleron, A. Najahi et G. Revy. *Automated Synthesis of Target-Dependent Programs for Polynomial Evaluation in Fixed-Point Arithmetic*. Rap. tech. RR-13006. 2013.

[79] J.-M. Muller. *Arithmétique des ordinateurs*. Français. Masson, 1989, p. 214.

[80] J.-M. Muller et al. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010, p. 572.

[81] R. Nehmeh et al. “Integer word-length optimization for fixed-point systems”. Dans : *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP),* mai 2014, p. 8321–8325.

[82] H. N. Nguyen. “Optimisation de la précision de calcul pour la réduction d’énergie des systèmes embarqués”. Thèse de doct. ENSSAT de l’Université de Rennes 1, 2011.

[83] H. N. Nguyen, D. Ménard et O. Sentieys. “Novel Algorithms for Word-length Optimization”. Dans : *19th European Signal Processing Conference (EUSIPCO-2011)*. Barcelona, Espagne, 2011.

[84] A. V. Oppenheim et R. W. Schafer. *Discrete-Time Signal Processing*. 3rd. Upper Saddle River, NJ, USA : Prentice Hall Press, 2009.

[85] C. H. Papadimitriou et K. Steiglitz. *Combinatorial Optimization : Algorithms and Complexity*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1982.

[86] K.N. Parashar, D. Menard et O. Sentieys. “A polynomial time algorithm for solving the word-length optimization problem”. Dans : *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2013, p. 638–645.

[87] P. Belanovic et M. Rupp. "Automated Floating-point to Fixed-point Conversion with the fixify Environment". Dans : *International Workshop on Rapid System Prototyping RSP 05*. Montreal, Canada, juin 2005, p. 172–178.

[88] P. Belanovic et M. Rupp. "Fixify : A Toolset for Automated Floating-point to Fixed-point Conversion". Dans : *Proceedings of the International Conference on Computing, Communications and Control Technologies*. Austin, Texas, août 2004, p. 28–32.

[89] B. Poraz. *A course in digital signal processing*. John Wiley & Sons, 1997.

[90] P. Prandoni et M. Vetterli. *Signal Processing for Communications (Communication and Information Sciences)*. EPFL Press, 1st edition, 2008.

[91] C. Reyes. "Distributed Subspace Tracking in Wireless Sensor Networks". Thèse de doct. E389, Vienna University of Technology, 2013.

[92] S. M. Rump, T. Ogita et S. Oishi. "Accurate Floating-Point Summation Part I : Faithful Rounding". Dans : *SIAM J. Scientific Computing* 31.1 (2008), p. 189–224.

[93] Wadekar S. A. "Accuracy sensitive word–length selection for algorithm optimization". Thèse de doct. University of Southern California, 1998.

[94] D. Schlichthärle. *Digital Filters : Basics and Design*. Springer, 2nd ed., 2011.

[95] R. Sethi et J. D. Ullman. "The Generation of Optimal Code for Arithmetic Expressions". Dans : *J. ACM* 17.4 (oct. 1970), p. 715–728.

[96] A.B. Sripad et D. Snyder. "A necessary and sufficient condition for quantization errors to be uniform and white". Dans : *IEEE Transactions on Acoustics, Speech and Signal Processing* 25.5 (oct. 1977), p. 442–448.

[97] W. Sung et K.-I. Kum. "Simulation-based word-length optimization method for fixed-point digital signal processing systems". Dans : *IEEE Trans. on Signal Processing* 43.12 (déc. 1995), p. 3087–3090.

[98] W. Sung et K.-I. Kum. "Word-length determination and scaling software for a signal flow block diagram". Dans : *1994 IEEE International Conference on Acoustics, Speech, and Signal Processing, 1994. ICASSP-94*. T. ii. Avr. 1994, II/457–II/460 vol.2.

[99] M. Abe S. Yamaki et M. Kawamata. "Digital Filters and Signal Processing". Dans : InTech, 2013. Chap. Chapter 9 : Analytical Approach for Synthesis of Minimum L_2-Sensitivity Realizations for State-Space Digital Filters, p. 213–242.

[100] V. Tavsanoglu et L. Thiele. “Optimal design of state-space digital filters by simultaneous minimization of sensitivity and roundoff noise”. Dans : *IEEE Trans. on Circuits and Systems* 31.10 (oct. 1984), p. 884–888.

[101] S. Vakili, J.M.P. Langlois et G. Bois. “Enhanced Precision Analysis for Accuracy-Aware Bit-Width Optimization Using Affine Arithmetic”. Dans : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.12 (déc. 2013), p. 1853–1865.

[102] V. Balakrishnan et S. Boyd. “On Computing the Worst-Case Peak Gain of Linear Systems”. Dans : *Systems & Control Letters* 19 (1992), p. 265–269.

[103] M. Vetterli, J. Kovacevic et V. K. Goyal. *Foundations of Signal Processing*. Cambridge University Press, 2014.

[104] Anastasia Volkova, Thibault Hilaire et Q. Lauter Christoph. “Reliable evaluation of the Worst-Case Peak Gain matrix in multiple precision”. 2014.

[105] S.A Wadekar et AC. Parker. “Accuracy sensitive word-length selection for algorithm optimization”. Dans : *International Conference on Computer Design : VLSI in Computers and Processors, 1998. ICCD ’98. Proceedings*. Oct. 1998, p. 54–61.

[106] C. S. Wallace. “A Suggestion for a Fast Multiplier”. Dans : *IEEE Transactions on Electronic Computers* EC-13.1 (fév. 1964), p. 14–17.

[107] B. Widrow et I. Kollár. *Quantization Noise : Roundoff Error in Digital Computation, Signal Processing, Control, and Communications*. Cambridge, UK : Cambridge University Press, 2008.

[108] Z Zhao. “An Efficient State-Space Realization With Minimum Roundoff Noise Gain”. Dans : *IEEE Trans. on Circuits and Systems I : Regular Papers* 54.2 (fév. 2007), p. 432–440.

[109] Z. Zhao, G. Li et J. Zhou. “Efficient digital filter structures with minimum roundoff noise gain”. Dans : *IEEE International Symposium on Circuits and Systems, 2005. ISCAS 2005.* Mai 2005, 2587–2590 Vol. 3.

[110] Z. Zhao, G. Li et J. Zhou. “Roundoff noise analysis of two efficient digital filter structures”. Dans : *IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP ’05).* T. 4. Mar. 2005, iv/333–iv/336 Vol. 4.