Formalisation en Coq des algorithmes de filtre numérique calculés en précision finie

Diane Gallois-Wong

université PARIS-SACLAY

Formalisation en Coq des algorithmes de filtre numérique calculés en précision finie

Coq formalization of digital filter algorithms computed using finite precision arithmetic

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n°580 : Sciences et technologies de l'information et de la communication (STIC)

Spécialité de doctorat : Informatique

Unité de recherche : Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France

Référent : Faculté des sciences d'Orsay

Thèse présentée et soutenue à Paris-Saclay, le 4 mars 2021, par

Diane GALLOIS-WONG

Composition du jury

Florent HIVERT Professeur, LISN, Université Paris-Saclay	Président
Yves BERTOT Directeur de recherche, Inria Sophia Antipolis – Méditerranée	Rapporteur & Examinateur
Eric FERON Professeur, School of Aerospace Engineering, Etats-Unis	Rapporteur & Examinateur
Jérôme FERET Chargé de recherche, Inria Paris & DI-ENS, ENS / CNRS / Université PSL	Examinateur
Mioara JOLDES Chargée de recherche, LAAS-CNRS, CNRS Toulouse	Examinatrice
Assia MAHBOUBI Chargée de recherche, Inria Rennes – Bretagne Atlantique, LS2N CNRS, Nantes	Examinatrice
Jean-Michel MULLER Directeur de recherche, LIP, CNRS Lyon	Examinateur

Direction de la thèse

Sylvie BOLDO Directrice de recherche, Inria Saclay – Île-de-France	Directrice de thèse
Thibault HILAIRE Maître de conférence, LIP6, Sorbonne Université	Co-encadrant

Thèse de doctorat

NNT : 2021UPASG016

Table des matières

Chapitre 1

Introduction

1.1 Motivation

Des télécommunications à l'aérospatiale en passant par la robotique, les domaines d'application des filtres numériques sont nombreux et variés. En effet, ils servent de briques de base en traitement du signal ainsi qu'en contrôle-commande, c'est-à-dire lorsqu'il faut déterminer des consignes pour piloter des actuateurs (moteur, vanne, etc.) à partir de mesures relevées (position, vitesse, température, etc.). Par exemple, un filtre peut modifier un signal audio afin d'atténuer le bruit ambiant au cours d'un appel téléphonique. Ou encore, un filtre peut recevoir en entrée des informations de nombreux capteurs sur l'état physique d'un avion, et produire en sortie des commandes à fournir aux moteurs ou pompes de l'avion.

Un écueil classique de la conception et de l'analyse des filtres numériques est la précision finie intrinsèque aux calculs effectués par les ordinateurs. En effet, pour utiliser des filtres en pratique, il faut les faire calculer par une machine, qu'il s'agisse d'un ordinateur généraliste ou d'un système embarqué. Les nombres sont alors représentés par des structures de données de taille finie, avec lesquelles seul un nombre fini de valeurs réelles sont représentables. On parle d'arithmétique des ordinateurs. Les deux systèmes de représentation les plus utilisés sont la virgule flottante et la virgule fixe. Lorsque le résultat d'un calcul ou une constante comme $\pi = 3{,}14...$ n'est pas représentable dans le système choisi, on doit l'arrondir vers une valeur représentable. Les erreurs d'arrondi dues à de telles approximations sont une source fréquente de problèmes dans les programmes informatiques. Un exemple souvent mentionné est l'échec de l'anti-missile Patriot MIM-104 [15] en 1991, qui rate l'interception d'un missile provoquant la mort de 28 soldats américains. Le système Patriot utilisait entre autres le temps en secondes depuis sa mise en route, stocké en virgule fixe sur 24 bits. Chaque mise à jour de ce temps était sujet à une minuscule erreur d'arrondi, mais ces erreurs successives s'étaient accumulées pendant plus de vingt heures au moment de l'arrivée du missile, résultant en une trop grande erreur sur la trajectoire d'interception calculée.

Généralement, les filtres numériques tolèrent assez bien les imprécisions liées à l'arithmétique des ordinateurs. Cependant, dans certains cas particuliers, les erreurs d'arrondi peuvent avoir un impact désastreux pour les raisons suivantes. Les algorithmes calculant des filtres sont fortement itératifs : de nombreux calculs sont enchaînés, chacun sujet à des erreurs d'arrondi qui peuvent donc se propager au fil des itérations. Cette propagation a un effet relativement négligeable dans la plupart des cas, mais quelquefois les erreurs s'accumulent et ont un impact conséquent. De plus, les implémentations de filtres numériques sont souvent limitées par des contraintes pratiques comme la taille du circuit électronique ou la consommation énergétique, notamment lorsqu'on les utilise dans des systèmes embarqués. On doit alors donner la priorité à l'efficacité (à la fois en temps et en mémoire) des calculs, parfois au détriment de leur précision. Les filtres sont donc souvent calculés avec la précision la plus faible possible (c'est-à-dire que les nombres sont représentés en virgule fixe en utilisant le moins de bits possible) permettant quand même d'obtenir des résultats suffisamment précis pour l'application considérée. Mais la propagation des erreurs fait que si on est légèrement trop optimiste sur cette précision minimale, on peut obtenir des résultats très différents de ce qu'on attendait. Notamment, on peut voir apparaître des comportement exceptionnels comme des *overflows* (dépassements de capacité quand un nombre est trop grand pour être représentable) qui changent complètement le comportement du filtre.

Si l'image d'une télévision est momentanément brouillée ou perdue, ce n'est pas grave. Mais un dysfonctionnement du système de contrôle d'un avion peut avoir des conséquences catastrophiques. Il est donc très important de bien connaître les effets de la précision finie sur le comportement d'un filtre. Pour choisir la précision minimale à utiliser dans les calculs et déterminer l'impact des erreurs d'arrondi sur la sortie d'un filtre, on a souvent recours à des simulations [6, 72] ou des études probabilistes [113]. De telles approches permettent de concevoir rapidement des implémentations de filtres efficaces (calculs rapides, peu d'énergie requise) qui ont un comportement adéquat dans la grande majorité des cas. Cependant, elles ne sont pas exhaustives : elles n'offrent aucune garantie sur ce qui se passe dans le pire cas. Pour les applications les plus critiques, par exemple en robotique ou pour les différentes formes de transport (automobile, aéronautique, aérospatial), on veut assurer le bon comportement du filtre dans la totalité des cas. On se tourne alors vers des analyses rigoureuses des erreurs d'arrondi même dans le pire cas [85, 108]. Mais il est possible de se tromper dans de telles analyses. La propagation des erreurs au cours des itérations, ainsi que l'optimisation agressive pour utiliser une précision tout juste suffisante dans les calculs, compliquent ces analyses d'erreurs et signifient que se tromper légèrement peut complètement fausser ces analyses.

Afin d'obtenir des garanties plus fortes sur les filtres utilisés dans des contextes critiques, on peut faire appel aux méthodes formelles [52]. C'est l'approche choisie dans cette thèse, plus particulièrement la preuve formelle interactive. Ainsi, ma thèse consiste à formaliser les filtres numériques en utilisant l'assistant de preuve Coq, et prouver formellement des résultats (pour la plupart préexistants dans la littérature) sur l'analyse des erreurs d'arrondi en précision finie dans les implémentations de ces filtres.

Cette thèse se trouve donc à l'intersection de trois domaines : les méthodes formelles, l'arithmétique des ordinateurs et le traitement du signal. Les sections 1.2 à 1.4 présentent des éléments de contexte pour mieux comprendre ces trois domaines respectifs, ainsi que les notions et définitions requises pour la suite du manuscrit. La section 1.5 propose un état de l'art de l'application des méthodes formelles aux filtres numériques implémentés en précision finie. La section 1.6 présente le plan de ce manuscrit, ainsi que l'organisation des fichiers de code composant ma formalisation en Coq. Elle donne également un lien vers ces fichiers. Enfin, la section 1.7 récapitule les notations et conventions utilisées dans le manuscrit (y compris certaines qui auront déjà été introduites dans les sections 1.2 à 1.4, et d'autres correspondant à des définitions qui ne seront présentées que dans les chapitres suivants).

1.2 Méthodes formelles et assistant de preuve Coq

Ma thèse s'inscrit dans le domaine des méthodes formelles, et plus particulièrement de la preuve formelle puisque j'utilise un assistant de preuve appelé Coq. La section 1.2.1 présente ce domaines général. La section 1.2.2 est une introduction à Coq destinée au lecteur qui n'est pas familier avec cet assistant de preuve. Elle explique entre autres comment lire les énoncés Coq que je donne tout au long de ce manuscrit. La section 1.2.3 présente les bibliothèques de Coq que j'utilise et les axiomes que je suppose. La section 1.2.4 introduit quelques notions et éléments syntaxiques de Coq apparaissant souvent dans le manuscrit. Enfin, la section 1.2.5 détaille les éléments de la bibliothèque MathComp que j'utilise beaucoup.

1.2.1 Méthodes formelles

Les programmes et systèmes informatiques sont omniprésents dans notre quotidien. Mais ils sont accompagnés d'un fléau bien connu : les bugs informatiques. En tant que programmeur, on a parfois l'impression de passer tout son temps à corriger des bugs, tout en sachant qu'il en restera probablement encore d'autres. Pour tenter de pallier ce problème, les programmes informatiques sont souvent testés sur de nombreuses entrées bien choisies et/ou générées aléatoirement. Cette approche, efficace en termes de ressources et temps requis, permet souvent de découvrir de nombreux bugs, mais ne garantit pas qu'il n'y a plus aucun bug. Et régulièrement, des bugs passent à travers les mailles du filet. En 2004, McConnell annonce qu'il y a environ 15-50 erreurs par 1000 lignes de code produit dans l'industrie [88].

Comme on peut s'y attendre, les bugs informatiques peuvent avoir des conséquences catastrophiques. Les exemples sont nombreux. Entre 1985 et 1987, la machine de radiothérapie Therac-25 administre une dose de radioactivité vingt fois trop élevée, causant la mort de cinq personnes [80]. J'ai déjà mentionné en section 1.1 l'échec de l'anti-missile Patriot MIM-104 [15] en 1991, causé par une accumulation d'erreurs d'arrondi. En 1996, la fusée Ariane 5 explose à cause d'un *overflow* : l'accélération dépasse la valeur maximale de la structure de données dans laquelle elle doit être stockée [9]. En plus des désastres occasionnellement médiatisés, les bugs informatiques sont une source continuelle de pertes financières.

Ces exemples et bien d'autres montrent l'importance de s'assurer que les programmes font bien ce qu'on attend d'eux. Pour les programmes utilisés dans des contextes critiques (où les dysfonctionnements mettent des vies en danger), ou pouvant causer des répercussions financières importantes, on veut des garanties fortes, qu'on ne peut généralement pas obtenir en se contentant de tester les programmes. On utilise alors les *méthodes formelles* [52]. Il s'agit d'une famille de techniques qui consistent à décrire un programme à l'aide d'un modèle mathématique, afin de pouvoir démontrer rigoureusement qu'il fonctionne correctement. Ce qu'on entend par *correctement* dépend de la méthode utilisée : on peut prouver que le programme ne va pas produire d'erreur, ou qu'il vérifie une *spécification* donnée, par exemple qu'il renvoie bien une certaine valeur. Parmi les différentes techniques consistuant les méthodes formelles, on rencontre l'interprétation abstraite [33], le *model checking* [7], la vérification déductive [39, 40] et la preuve formelle [51]. Des applications de ces différentes méthodes aux filtres numériques seront présentées en section 1.5. La preuve formelle inclut la preuve automatique et la preuve interactive à l'aide d'un assistant de preuve [43] comme PVS [97], Isabelle / HOL [93], Mizar [92] ou Coq [11]. Cette thèse utilise la preuve interactive avec Coq.

1.2.2 Introduction à Coq

Coq [11] est un assistant de preuve interactif. Il inclut un langage de spécification appelé *Gallina*, qui permet d'énoncer des définitions et théorèmes mathématiques. Ce langage est basé sur une théorie des types d'ordre supérieur appelée *calcul des constructions inductives* (*CIC* pour *Calculus of Inductive Constructions*) [100]. Ensuite, pour construire une preuve d'un théorème, l'utilisateur guide Coq à l'aide de *tactiques*, qui peuvent par exemple appliquer une règle de logique ou un théorème démontré précédemment. Enfin, Coq vérifie la preuve assemblée en utilisant le typage. En effet, Coq est basé sur la *correspondance de Curry-Howard* entre preuve mathématique et programme [60]. Le principe est que les types peuvent être vus comme des propositions, et les éléments ayant un certain type comme des preuves de la proposition correspondante. Coq est basé sur la logique intuitionniste : il n'y a pas d'axiome du tiers-exclu ($\forall P, P \vee \neg P$). Coq est par ailleurs muni d'un mécanisme d'extraction [79] vers OCaml et Haskell. Parmi les développements en Coq bien connus, on peut citer la preuve du théorème des quatre couleurs [45], celle du théorème de Feit-Thompson (*odd order theorem*) [46], ou encore CompCert [78], un compilateur C efficace et certifié.

La contribution principale de ma thèse est une formalisation en Coq des filtres numériques et de leurs erreurs d'arrondi en précision finie. Ce manuscrit est donc parsemé d'énoncés en Coq (plus précisément, dans le langage de spécification Gallina). Ceux-ci correspondent à des définitions ou théorèmes que je décris au préalable en langage mathématique courant, donc il n'est pas strictement nécessaire de pouvoir les lire. Néanmoins, il est intéressant de voir exactement ce qui a été prouvé en Coq. De plus, le langage de spécification Gallina est souvent assez lisible même par quelqu'un ne l'ayant jamais étudié. Je propose dans cette section une brève introduction à la lecture des énoncés Coq à l'aide de quelques exemples. L'objectif est que le lecteur puisse ensuite comprendre l'es-

sentiel des énoncés Coq apparaissant dans ce manuscrit. Pour une présentation plus générale et plus approfondie de Coq, voir par exemple [10, 11].

Exemple de définition. Voici un prédicat sur les fonctions réelles, c'est-à-dire une fonction qui prend en argument une fonction réelle et indique si celle-ci vérifie une certaine propriété (ici le fait d'être bornée) :

```
Definition bounded (f : R → R) : Prop :=
  exists M : R, forall r : R, Rabs (f r) ≤ M.
```

Même sans connaître Coq, on peut deviner qu'une fonction $f : \mathbb{R} \to \mathbb{R}$ satisfait le prédicat si :

$$\exists M \in \mathbb{R}, \forall r \in \mathbb{R},\ |f(r)| \leq M \tag{1.1}$$

Détaillons maintenant certains éléments syntaxiques de cette définition. Le type `R` est le type des réels de la bibliothèque standard de Coq. Les détails de ces réels axiomatiques seront présentés en section 1.2.3 mais n'ont pas d'importance pour cet exemple. La fonction `Rabs` de la bibliothèque standard implémente la valeur absolue. On peut demander son type à Coq à l'aide de la commande `Check`. Je noterai toujours `(* en commentaire *)` ce que répond Coq :

```
Check Rabs. (* Rabs : R → R *)
```

Le type `Prop` est le type des propositions, avec deux valeurs possibles : `True` et `False`. On est en train de définir un élément `bounded` qui prend un argument `f` de type `R → R` et renvoie un élément de type `Prop`, dont la construction est donnée après le symbole `:=`. On obtient bien un prédicat, c'est-à-dire une fonction à valeurs dans `Prop`. On peut vérifier le type de `bounded` en le demandant à nouveau à Coq :

```
Check bounded. (* bounded : (R → R) → Prop *)
```

On aurait d'ailleurs pu définir directement `bounded` comme une fonction avec la syntaxe `fun ... ⇒ ...`. La définition suivante est équivalente à la première :

```
Definition bounded : (R → R) → Prop :=
  fun f : R → R ⇒ exists M : R, forall r : R, Rabs (f r) ≤ M.
```

Ce n'est pas nécessaire d'indiquer les types lorsque Coq peut les inférer. Ici, il sait que `f` est à valeurs dans `R` et que `M` est de type `R` grâce à l'utilisation de `Rabs` et à l'inégalité. Il sait aussi que la valeur de retour de `bounded` est de type `Prop`. En revanche, le fait que les arguments de `f` soient dans `R` doit apparaître quelque part, car cela ne peut pas être inféré si on enlève toutes les indications de type (on a alors un message d'erreur : « Cannot infer `?T` in the partial instance `"?T → R"` found for the type of `f`. »). On peut par exemple écrire seulement le type de `bounded`, ou seulement celui de la variable `f`, ou encore de la variable `r`. Notons aussi qu'on peut utiliser la syntaxe `(expression : type)` n'importe où pour indiquer le type de l'expression. Par exemple, on aurait aussi bien pu écrire la définition suivante (le type des arguments de `f` est bien indiqué, même si cette fois c'est au moment où `f` est appliquée) :

```
Definition bounded :=
  fun f ⇒ exists M, forall r, Rabs (f (r : R)) ≤ M.
```

Si les détails d'une définition (tout ce qui vient après `:=`) sont trop complexes ou non pertinents pour mon propos, je me contenterai souvent de signaler qu'une définition existe en écrivant `Definition nom : type.`

Exemple de lemme et de preuve. Intéressons-nous maintenant à un exemple de lemme accompagné d'une preuve. C'est la seule preuve Coq qui apparaîtra explicitement dans le manuscrit, et il n'est pas nécessaire d'en comprendre tous les détails : mon objectif est simplement de donner un aperçu de ce à quoi une telle preuve ressemble. Par la suite, je me contenterai d'énoncer les lemmes en indiquant éventuellement l'idée générale de leur preuve (les preuves complètes sont accessibles en ligne : l'url sera donnée en section 1.6.3). Quelques exceptions s'adresseront au lecteur familier avec Coq : des mentions occasionnelles d'une tactique particulière, et la section 6.2 proposant des commentaires sur le développement de ma formalisation.

Prouvons que la fonction mathématique *sinus* est bornée. On utilise le mot clé `Lemma` suivi du nom du lemme (accompagné d'éventuelles variables comme on verra plus bas), du symbole « : » et de la proposition à prouver (c'est-à-dire le type du lemme, grâce à la correspondance de Curry-Howard [60]). Vient ensuite la preuve, souvent constituée d'une série de tactiques entre les commandes `Proof` et `Qed`. Ici, on affirme que le prédicat `bounded` ci-dessus est vérifié par la fonction `sin : R → R` de la bibliothèque standard.

```
Lemma bounded_sin : bounded sin.
Proof.
  exists 1.
  intro.
  apply Rabs_le.
  apply SIN_bound.
Qed.
```

Coq est interactif : entre chaque application de tactique, il affiche où en est la preuve, notamment quelles sont les propositions qu'il reste à prouver (appelées les *buts*). Au début de la preuve présentée ici, lorsqu'on n'a pas encore utilisé de tactique, Coq affiche un seul but :

```
bounded sin
```

Le prédicat `bounded` demande l'existence d'un réel `M` vérifiant la suite de la définition. On doit donc exhiber un tel `M`. À l'aide de la tactique `exists`, on affirme qu'il suffit de choisir `M = 1`. Suite à l'application de cette tactique, le but affiché devient :

```
forall r : R, Rabs (sin r) ≤ 1
```

La tactique `intro` fait passer la variable `r` dans le contexte. Les détails n'ont pas d'importance ici, si ce n'est qu'on obtient juste le but suivant, où l'on sait que `r` est quantifiée universellement :

```
Rabs (sin r) ≤ 1
```

On utilise alors la tactique `apply` qui permet d'appliquer un lemme (prouvé précédemment ou provenant d'une bibliothèque). Affichons le type du lemme `Rabs_le` de la bibliothèque standard (c'est-à-dire la proposition à laquelle il correspond).

```
Check Rabs_le.
  (* Rabs_le : forall x y : R, - y ≤ x ≤ y → Rabs x ≤ y *)
```

Suite à l'application de ce lemme, notre but devient :

```
- 1 ≤ sin r ≤ 1
```

Or la bibliothèque standard contient justement le lemme `SIN_bound` suivant.

```
Check SIN_bound. (* SIN_bound : forall x : R, -1 ≤ sin x ≤ 1 *)
```

Ce lemme prouve le but qu'il nous restait. Suite à son application, Coq annonce que la preuve est terminée et accepte alors la commande `Qed`.

Notons que le mot clé `Lemma` a de nombreuses variantes comme `Theorem`, `Corollary`, `Fact` ou `Remark`. Tous ces mots sont synonymes pour Coq, et servent seulement à indiquer différentes nuances au lecteur.

Par ailleurs, un lemme peut être paramétré par des variables indiquées à gauche du symbole « : », qui sont alors quantifiées universellement (encore une fois, on peut indiquer explicitement leur type ou laisser Coq l'inférer). Par exemple, les deux énoncés suivants sont équivalents, et affirment que pour n'importe quel réel `c`, la fonction constante égale à la valeur `c` est bornée.

```
Lemma bounded_const c : bounded (fun _ ⇒ c).
Lemma bounded_const : forall c, bounded (fun _ ⇒ c).
```

1.2.3 Bibliothèques et axiomes utilisés

Cette section présente les bibliothèques de Coq que j'utilise dans ma formalisation, ainsi que les axiomes que je suppose.

Réels de la bibliothèque standard. J'utilise les réels de la bibliothèque standard de Coq [87], dont le type est noté `R`. Ils reposent sur des axiomes décrivant les propriétés principales des nombres réels et de leurs opérations. Cette approche est très différente d'autres bibliothèques qui construisent les nombres réels à l'aide des suites de Cauchy (bibliothèque MathClasses de Coq [74]) ou des coupures de Dedekind (bibliothèque standard de Mizar [107]). Les réels de la bibliothèque standard sont basés sur deux éléments (`R0` et `R1`), quatre opérations élémentaires (addition `Rplus`, opposé `Ropp`, multiplication `Rmult` et inverse `Rinv`, à partir desquelles sont définies la soustraction `Rminus` et la division `Rdiv`), et l'ordre strict usuel `Rlt` (dont on dérive l'ordre large `Rle` ainsi que les ordres symmétriques respectifs `Rgt` et `Rge`). On utilise les notations usuelles comme `0`, `1`, `+`, `-` (unaire ou binaire), `*`, `/` (unaire pour l'inverse ou binaire) ou `<`. Ces notations demandent d'être dans le *scope* des réels identifié par `%R`, comme on expliquera en section 1.2.4. Voici quelques uns des 19 axiomes définissant les réels :

```
Axiom Rplus_comm : forall r1 r2 : R, r1 + r2 = r2 + r1.
Axiom Rplus_0_l : forall r : R, 0 + r = r.
Axiom Rmult_assoc :
  forall r1 r2 r3 : R, r1 * r2 * r3 = r1 * (r2 * r3).
Axiom Rinv_l : forall r : R, r ≠ 0 → / r * r = 1.
Axiom Rmult_plus_distr_l :
  forall r1 r2 r3 : R, r1 * (r2 + r3) = r1 * r2 + r1 * r3.
```

```
Axiom Rplus_lt_compat_l :
  forall r r1 r2 : R, r1 < r2 → r + r1 < r + r2.
Axiom total_order_T :
  forall r1 r2 : R, {r1 < r2} + {r1 = r2} + {r1 > r2}.
```

La notation `{ P } + { Q }` représente une disjonction constructive entre `P` et `Q`. Le dernier axiome signifie en particulier qu'on peut décider l'égalité de deux réels.

Coquelicot. La bibliothèque Coquelicot [18, 77] étend la bibliothèque standard des réels pour faciliter les raisonnements d'analyse réelle. Elle est basée sur les filtres topologiques (qui n'ont rien à voir avec les filtres numériques de traitement du signal !). Elle fournit notamment des outils (dont des fonctions totales) pour travailler avec des limites, des séries entières, des dérivées ou des intégrales.

Je n'utilise que très peu cette bibliothèque. Je me sers surtout des résultats sur les limites de suites de réels en section 4.3. De plus, le lemme `LPO` (pour *Limited Principle of Omniscience*), prouvé par Coquelicot à partir des axiomes définissant les réels de la bibliothèque standard, intervient en section 2.1.2.

Flocq. La bibliothèque Flocq [20, 21] formalise l'arithmétique des ordinateurs. Elle décrit des formats généraux de précision finie, qui peuvent notamment être spécialisés pour les formats classiques de virgule flottante ou de virgule fixe. J'utilise cette bibliothèque exclusivement et intensivement dans le chapitre 5 : je la présente donc en détail en section 5.1.2.

MathComp. La bibliothèque *Mathematical Components* [86] (souvent appelée *MathComp*) formalise de nombreuses notions mathématiques, en particulier en algèbre. Elle joue notamment un rôle central dans les démonstrations du théorème des quatre couleurs [45] et du théorème de Feit-Thompson [46]. Elle repose sur l'utilisation de *SSReflect* [47] (pour *Small Scale Reflection*), un langage de tactiques conçu pour faciliter l'écriture et le maintien des preuves Coq.

L'utilisateur est invité à exprimer des propriétés et prédicats à l'aide du type `bool` (dont les valeurs sont `true` et `false`) plutôt que `Prop` lorsque c'est possible. Une différence importante est que l'égalité sur `bool` est décidable, permettant de raisonner par disjonction de cas sur une assertion de type `bool`, ce qui n'est généralement pas possible avec `Prop`. La correspondance `is_true : bool → Prop` permet de fournir une expression de type `bool` à la commande `Lemma`. Il s'agit en effet d'une coercion comme on verra en section 1.2.4, ce qui signifie que Coq peut implicitement convertir une expression de type `bool` vers le type `Prop` exigé par cette commande. Des lemmes dits de *réflection*, ou *vues*, permettent aussi de lier des expressions dans `Prop` et dans `bool` du même concept.

Les éléments principaux de MathComp que j'utilise sont les structures algébriques, les matrices et les grands opérateurs. La section 1.2.5 introduira chacun de ces éléments. J'utilise aussi les opérations sur `nat` et le type `int` décrits ci-dessous.

Types pour les ensembles de nombres usuels. Il y a parfois plusieurs choix possibles pour la représentation en Coq des ensembles mathématiques usuels. Je liste donc ici lesquels j'utilise. Comme expliqué plus haut, je travaille avec les réels de la bibliothèque standard, de type `R`. Pour les entiers naturels, j'utilise

le type `nat` de la bibliothèque standard. Cependant, j'utilise les opérations et relations sur `nat` fournies par MathComp (comme l'addition `addn` ou la relation inférieur ou égal `leq : nat → nat → bool`), car elles sont plus adaptées aux raisonnements suggérés par SSReflect que celles de la bibliothèque standard.

Enfin, pour les entiers relatifs, j'utilise le type `int` de MathComp dans les chapitres 2 à 4, mais le type `Z` de la bibliothèque standard dans le chapitre 5. Le type `int` est plus facile à utiliser dans le cadre de SSReflect. Mais dans le chapitre 5, je me repose beaucoup sur la bibliothèque Flocq qui utilise déjà `Z`. Utiliser ces deux types différents ne m'a pas posé de problème car ils représentent des concepts différents : `int` est utilisé pour des indices de signaux numériques, tandis que `Z` représente des exposants pour des nombres à virgule fixe. J'envisage cependant d'utiliser `int` partout comme discuté en section 6.2.1.

Axiomes. Les axiomes que j'utilise dans cette formalisation sont :

- les axiomes de la bibliothèque standard qui définissent les réels [87] comme vu précédemment.
- `FunctionalExtensionality` : deux fonctions identiques point à point sont égales.
- `ProofIrrelevance` : deux preuves d'une même propriété sont égales.
- `R_epsilon_statement` et `fnv_epsilon_statement` : des instanciations de l'axiome *epsilon* de Hilbert aux types `R` (nombres réels) et `nat → VT` (suites à valeurs dans un espace vectoriel).

Les axiomes `FunctionalExtensionality` et `ProofIrrelevance` sont disponibles dans la bibliothèque standard lorsqu'on importe les modules de même nom, et sont généralement considérés comme sûrs par la communauté [1]. J'ai défini `R_epsilon_statement` et `fnv_epsilon_statement` en m'inspirant de `Epsilon.epsilon_statement` de la bibliothèque standard. Tous ces axiomes serviront en particulier à la formalisation des signaux numériques comme on verra en section 2.1.2.

1.2.4 *Scope*, coercion et section

Cette section présente quelques notions et éléments syntaxiques de Coq que j'utilise au cours du manuscrit.

Scope. Des notations comme `0`, `<`, `*` ou `+` peuvent représenter des objets différents : par exemple, le zéro réel de type `R` n'est pas le même que le zéro entier de `nat`. Coq utilise des *scopes* pour indiquer quel sens donner à ces notations. Ces *scopes* peuvent être délimités localement par `(...)%nat` ou `(...)%R` par exemple. Ainsi, `(x + y)%nat` est l'addition entière `addn` appliquée à `x` et `y` qui doivent donc être de type `nat`, tandis que `(x + y)%R` est l'addition réelle `Rplus` avec `x` et `y` de type `R`. Il est aussi possible d'ouvrir un *scope* pour plusieurs énoncés successifs (avec une commande `Open Scope`), mais je ne l'écrirai pas explicitement car le *scope* utilisé sera généralement évident (et dans les cas où il ne l'est pas, j'utiliserai les délimitations locales).

On a vu en section 1.2.3 que pour les entiers relatifs, j'utilise parfois le type `Z` de la bibliothèque standard et parfois `int` de MathComp [86]. Le *scope* pour le premier type est délimité par `%Z`. Pour le second, la plupart des opérations que

1. `https://github.com/coq/coq/wiki/The-Logic-of-Coq`

j'utilise sont en fait définies par MathComp pour n'importe quel anneau et sont donc associées à un scope de MathComp appelé `ring_scope`, habituellement délimité par `%R`. Mais pour ne pas confondre avec celui des réels, j'ai choisi de plutôt délimiter `ring_scope` par `%RT`, voire souvent par `%int` pour indiquer au lecteur que je suis en train d'appliquer des opérations de `ring_scope` à des éléments de type `int`.

Coercion. En mathématiques, l'ensemble $\mathbb{N}$ des entiers naturels est inclus dans l'ensemble $\mathbb{Z}$ des entiers relatifs. Mais en Coq, le type `nat` des entiers naturels et le type `int` des entiers relatifs (venant de la bibliothèque MathComp présentée plus bas) sont disjoints : un élément `n : nat` n'a pas le type `int`. Cependant, on dispose d'un plongement `Posz : nat → int`, c'est-à-dire une fonction injective qui, du point de vue mathématique, laisse inchangés les entiers naturels bien que leur type soit modifié. Un autre exemple de plongement est `INR : nat → R`. Pour faciliter l'écriture des énoncés, un plongement peut être déclaré comme une *coercion* : Coq effectue alors implicitement la conversion quand c'est nécessaire. Par exemple, si on écrit `(k + n)%int` où la variable `k` est de type `int` et `n` de type `nat`, Coq comprend qu'il s'agit en fait de `(k + Posz n)%int` car `Posz` est une coercion. En revanche, `INR` n'est pas déclaré comme une coercion donc il faut l'écrire explicitement si on veut utiliser le réel correspondant à un entier naturel.

Section. Lorsqu'on travaille sur plusieurs définitions et lemmes dépendant des mêmes arguments, on peut factoriser ces arguments grâce au mécanisme de *section*. Une section est délimitée par les commandes `Section` et `End` suivies du nom de la section. À l'intérieur de la section, on peut déclarer des variables avec la commande `Variable` ou `Variables`[2] (le type doit ici être donné explicitement). On peut aussi déclarer une hypothèse avec `Hypothesis`, qui est supposée vraie dans le reste de la section. Ces variables peuvent ensuite être utilisées dans les définitions et lemmes de la section. À la fermeture de la section, les variables et hypothèses de la section deviennent des variables et hypothèses ordinaires de chaque définition ou lemme de la section qui les a utilisées. Voici un exemple.

```
Section Exemple.

Variables (m n : nat).
Context (a : R) {b : R}. (* comme Variables (a : R) (b : R)
                            en première approximation *)
Hypothesis n_gt0 : (0 < n)%nat.

Definition f_affine r := (a * r + b)%R.
Lemma predn_lt : (n.-1 < n)%nat. (* .-1 prédécesseur unaire *)
Check f_affine. (* f_affine : R → R *)
Check predn_lt. (* predn_lt : (n.-1 < n)%nat *)

End Exemple.

Check f_affine. (* f_affine : R → R → R → R *)
Check predn_lt.
  (* predn_lt : forall n : nat, (0 < n)%nat → (n.-1 < n)%nat *)
```

2. J'utilise parfois `Context` au lieu de `Variable` pour pouvoir utiliser la syntaxe `{v : T}` des arguments implicites, comme `{b : R}` dans l'exemple proposé. Mais c'est un détail technique qui s'adresse au lecteur habitué à Coq. En première approximation, on peut considérer que `Context` est synonyme de `Variables` et qu'utiliser des accolades ou des parenthèses ne change rien.

Dans la section `Exemple`, on dispose de quatre variables `m`, `n`, `a` et `b`, avec l'hypothèse que `n` est strictement positif. On peut définir une fonction affine à partir des paramètres `a` et `b`, et montrer que le prédécesseur de `n` est strictement inférieur à `n` (cela nécessite l'hypothèse `0 < n` vu que le prédécesseur de `0` est `0`). Tant qu'on est dans la section, `f_affine` est une fonction réelle et `predn_lt` est simplement une inégalité. Mais après la section, on remarque que `f_affine` a trois arguments réels vu que `a` et `b` sont désormais aussi ses arguments, et `predn_lt` fait explicitement intervenir une variable `n` et l'hypothèse qu'elle est strictement positive (car cette hypothèse a vraiment été utilisée dans la preuve de `predn_lt` que je n'ai pas donnée ici). En revanche, `predn_lt` ne dépend toujours pas de `m`, `a` ou `b`, et `f_affine` ne dépend pas de l'hypothèse `0 < n`.

Dans le manuscrit, je n'explicite pas toujours les ouvertures et fermetures de sections Coq. Dès qu'une commande `Variable`, `Context` ou `Hypothesis` apparaît, cela suppose alors que tout le code présenté dans un paragraphe ou une sous-section du manuscrit est contenu dans une section Coq.

1.2.5 Structures algébriques, matrices et grands opérateurs de MathComp

Cette section présente les éléments de la bibliothèque MathComp [86] (introduite en section 1.2.3) les plus utilisés tout au long de ma formalisation.

Structures algébriques. MathComp décrit toute une hiérarchie de structures algébriques. Elles héritent les unes des autres grâce à des coercions (section 1.2.4). Je n'utilise que les quelques structures suivantes. Le type `ringType` est le type des anneaux (pas forcément unitaires ; on parle parfois de pseudo-anneau), `comRingType` celui des anneaux commutatifs et `comUnitRingType` celui des anneaux commutatifs unitaires (ayant un élément neutre pour la multiplication). J'utiliserai la notation `RT` pour un anneau de type `ringType` (ou souvent `comRingType` ou `comUnitringType`). Le type `vectType RT` est alors celui des modules à gauche de dimension finie sur l'anneau `RT`. Un tel module sera appelé un espace vectoriel par abus de langage (techniquement, un module est à un anneau ce qu'un espace vectoriel est à un corps, donc il faudrait que `RT` soit un corps pour pouvoir parler d'espace vectoriel). Un tel espace vectoriel de type `vectType RT` sera noté `VT`.

Matrices. MathComp inclut une bibliothèque très complète sur les matrices. Le type des matrices de taille $\mathtt{m} \times \mathtt{n}$ pour `(m n : nat)` à coefficients dans un type `T` est noté `'M[T]_(m, n)`. Une telle matrice correspond à une fonction vers `T` depuis l'ensemble fini `'I_m * 'I_n`, où `*` est le produit cartésien et `'I_m` est l'ensemble des ordinaux de `m` : $\{0, 1, \ldots, \mathtt{m}-2, \mathtt{m}-1\}$ (une coercion `nat_of_ord` envoie un ordinal vers l'entier correspondant dans `nat`). On dispose d'une coercion `fun_of_matrix` qui peut implicitement convertir une matrice de `'M[T]_(m, n)` en une fonction de type `'I_m → 'I_n → T`, si bien que pour une matrice `A` et des indices `i` et `j` dans les bons ensembles d'ordinaux, on peut simplement écrire `A i j` pour obtenir le coefficient de `A` à la position $(\mathtt{i}, \mathtt{j})$. On peut définir une matrice à partir d'un terme général pour le coefficient en $(\mathtt{i}, \mathtt{j})$ grâce à l'opérateur `\matrix_(i, j)`. Par exemple, voici la matrice avec le réel 1 sur la diagonale supérieure et 0 partout ailleurs.

```
Definition Mdiasup : 'M[R]_(m, n) :=
  \matrix_(i, j) (if (i.+1 == j) then 1 else 0)%R.
```

On remarque que `i` et `j`, dont les types respectifs sont `'I_m` et `'I_n`, ont été implicitement convertis vers `nat` (car `.+1` est une notation unaire pour le successeur dans `nat`, et l'égalité doit porter sur deux éléments de même type).

Pour les matrices carrées, on peut écrire `'M[T]_n` pour `'M[T]_(n, n)`. On a aussi les notations `'cV[T]_n = 'M[T]_(n, 1)` pour les vecteurs colonnes et `'rV[T]_n = 'M[T]_(1, n)` pour les vecteurs lignes (c pour *column* et r pour *row*). Souvent, le type des coefficients peut être inféré : on peut alors écrire `'M_(m, n)` ou `'cV_n` par exemple.

Le type `T` des coefficients peut être quelconque. Mais s'il possède des propriétés algébriques intéressantes, les matrices correspondantes peuvent les hériter. Ainsi, dès que `T` est un `zmodType` (type des groupes abéliens, qui peuvent être vus comme des modules sur l'anneau $\mathbb{Z}$ d'où le nom), on dispose de l'addition de matrices (coefficient par coefficient), notée `+`. Et si c'est un `ringType`, on a aussi accès à la multiplication matricielle usuelle, notée `*mx`.

MathComp inclut de nombreux concepts mathématiques sur les matrices. J'utilise notamment la transposition, les matrices par blocs, les permutations de lignes ou colonnes, et la notion de matrice inversible. Cette bibliothèque formalise aussi des notions avancées d'algèbre linéaire que je n'utilise pas actuellement, mais peuvent servir à étendre ma formalisation (voir les perspectives en section 6.3.3).

Grands opérateurs. MathComp fournit des *grands opérateurs* [12] permettant d'itérer un opérateur binaire sur un ensemble de valeurs. Par exemple, on peut les utiliser pour traduire des expressions mathématiques comme $\sum_{i=0}^{n-1} i^2$ ou $\prod_{x \in X} f(x)$, qu'on peut voir comme une itération d'additions ou de multiplications binaires, à partir de l'élément neutre 0 ou 1 respectivement. La syntaxe est `\big[op/e0]_(range) expr` où `op` est l'opérateur binaire, `e0` la valeur initiale de l'itération (généralement l'élément neutre de `op`), `range` introduit un indice et décrit l'ensemble parcouru par cet indice, et enfin `expr` donne le terme général en fonction de cet indice. Par exemple :

```
Definition sum_squares (n : nat) := \big[addn/0]_(0 ≤ i < n) i ^ 2.
```

où `addn` est l'addition binaire sur `nat` d'élément neutre `0`, et `(0 ≤ i < n)` signifie que l'indice `i` parcourt les entiers de `0` inclus à `n` exclu. Les descriptions d'indices que j'utilise dans ce manuscrit sont : celle que je viens de donner pour les entier ; `(i < n)` pour parcourir l'ensemble `'I_n` des ordinaux de `n` ; et enfin `(i ← s)` pour parcourir une liste `s` (l'indice `i` a alors le type `T` des éléments de `s`, qui est de type `seq T`).

Lorsque le type du terme général est muni d'un opérateur faisant clairement office d'addition (par exemple `nat`, ou n'importe quel `zmodType` vu précédemment), on peut utiliser la notation plus simple `\sum_(range) expr` :

```
Definition sum_squares n := \sum_(0 ≤ i < n) i ^ 2.
```

1.3 Arithmétiques en précision finie

L'arithmétique des ordinateurs est le domaine de l'informatique qui étudie les représentations des nombres (principalement entiers ou réels) dans les ordinateurs, ainsi que les calculs impliquant ces nombres. Souvent, on utilise une structure de donnée concise, qui ne peut cependant représenter qu'un nombre fini de nombres réels. Il faut alors indiquer comment arrondir un nombre qui n'est pas représentable vers un nombre qui l'est. On parle d'arithmétique en précision finie. Les deux telles arithmétiques les plus utilisées sont l'arithmétique en virgule flottante et l'arithmétique en virgule fixe. Elles couvrent à elles deux la grande majorité des programmes informatiques faisant intervenir des approximations de nombres réels. Elles sont présentées dans les sections 1.3.1 et 1.3.2 respectivement. Mais parfois, on veut étudier le comportement en précision finie d'un programme sans avoir besoin de savoir à l'avance quelle arithmétique sera utilisée. On considère alors une arithmétique en précision finie générique comme expliqué en section 1.3.3.

1.3.1 Virgule flottante

La virgule flottante n'est pas étudiée explicitement dans cette thèse. J'en donne cependant un aperçu ici car c'est un élément de contexte important. De plus, l'arithmétique en précision finie générique de la section 1.3.3 qui sera utilisée dans les chapitres 3 et 4 couvre aussi bien la virgule flottante que la virgule fixe, et est généralement destinée à être instanciée à l'une des deux. Par ailleurs, plusieurs notions présentées dans cette section, comme les modes d'arrondis ou les *overflows*, peuvent se généraliser à d'autres arithmétiques.

Les formats de représentation, les arrondis et les opérations de la virgule flottante sont régis par le standard IEEE-754 de 1985 [61], révisé en 2008 [63] et 2019 [65].

La base 2 est généralement utilisée : les nombres sont représentés à l'aide de chiffres qui valent 0 ou 1, appelés des *bits*. À partir de la révision de 2008 [63], la norme IEEE-754 inclut aussi des formats en base 10, notamment utilisés dans le domaine de la finance. La section courante ne présente que la base 2, mais l'arithmétique générique de la section 1.3.3 peut représenter des formats flottants dans n'importe quelle base.

Un nombre flottant est représenté par un *bit de signe* s (0 pour positif, 1 pour négatif), un *exposant* entier e et une *mantisse* positive m. La valeur du nombre est alors :

$$(-1)^s \times m \times 2^e \tag{1.2}$$

Un *format* de virgule flottante indique comment le bit de signe, l'exposant et la mantisse sont répartis dans le mot binaire sur lequel est codé le nombre flottant. Par exemple, la figure 1.1 montre cette répartition pour un format classique appelé *binary32* (ou *simple précision*) : l'exposant est représenté sur 8 bits et la mantisse (ou plutôt sa fraction comme on verra plus bas) sur 23 bits.

L'exposant est représenté sur les bits qui lui sont dédiés comme un entier positif en base 2, auquel il faut ensuite soustraire une constante appelée le *biais* qui dépend du format. Certaines combinaisons de bits d'exposant sont par

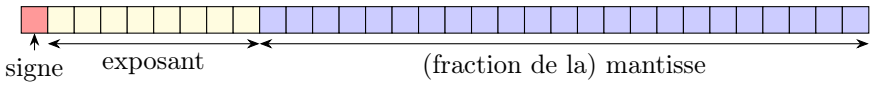


FIGURE 1.1 – Répartition des 32 bits d'un nombre flottant *binary32*

ailleurs réservées pour des flottants particuliers. Sans rentrer dans les détails, on peut retenir que chaque format a un exposant minimal $e_{\min}$ et un exposant maximal $e_{\max}$, et l'exposant e d'un nombre peut être n'importe quel entier tel que $e_{\min} \leq e \leq e_{\max}$.

La mantisse est un nombre qui s'écrit en base 2 sur p bits où p est appelée la *précision* du format, avec la virgule située entre le premier bit et les $p-1$ autres bits. Mais le premier bit est implicitement un 1 (sauf cas particuliers expliqués plus bas) et n'est donc pas stocké. Le segment réservé à la mantisse sur la figure 1.1 contient en fait la *fraction* de la mantisse, c'est-à-dire $p-1$ bits notés $m_1, \ldots, m_{p-1}$ tels que (où l'indice 2 à la fin indique un nombre écrit en base 2) :

$$m = 1{,}m_1 m_2 \ldots m_{p-1}{}_2 \tag{1.3}$$

On a donc $1 \leq m < 2$, et m est divisible par 2^{-p+1} (c'est-à-dire que $m \times 2^{p-1}$ est un entier).

La figure 1.2 indique les caractéristiques des deux formats flottants les plus communs : le format *binary32* (ou *simple précision*) sur 32 bits et le format *binary64* (ou *double précision*) sur 64 bits.

Format	Nombre de bits				Biais	$e_{\min}$	$e_{\max}$	p
	total	signe	exposant	fraction				
Binary32	32	1	8	23	127	-126	127	24
Binary64	64	1	11	52	1023	-1022	1023	53

FIGURE 1.2 – Caractéristiques des deux formats usuels de virgule flottante

Exemple. Représentons une approximation de $\pi = 3{,}14...$ en *binary32*. On commence par écrire π en base 2 en arrondissant (avec un arrondi au plus proche, défini plus bas) pour garder $p = 24$ bits :

$$\pi \approx 11{,}00100100001111111011011_2 \tag{1.4}$$

On introduit un exposant pour décaler la virgule afin d'avoir un seul chiffre avant celle-ci :

$$\pi \approx 1{,}00100100001111111011011_2 \times 2^1 \tag{1.5}$$

Le signe est 0 puisque π est positif. L'exposant est 1, donc en tenant compte du biais (qui vaut 127 en *binary32*), la valeur à stocker est 128 c'est-à-dire 10000000_2 sur 8 bits. Enfin, pour la mantisse, on écrit juste les 23 bits après la virgule, puisque le bit initial 1 est implicite. On obtient la représentation flottante de la figure 1.3.

Cas particuliers. En plus de la représentation expliquée ci-dessus, on définit des cas particuliers qui utilisent les valeurs réservées de l'exposant. Pour représenter des nombres de valeur absolue strictement inférieure à $2^{e_{\min}}$ (qui sont

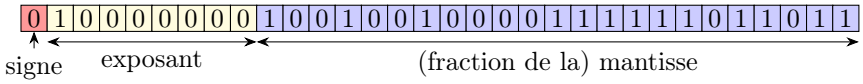


FIGURE 1.3 – Exemple : approximation de π en flottant *binary32*

dits *sous-normalisés* ou *dénormalisés*), on utilise un exposant particulier qui signale que le bit initial implicite de la mantisse vaut exceptionnellement 0 au lieu de 1. On définit aussi les flottants particuliers suivants : deux zéros $+0$ et -0, les infinis $+\infty$ et $-\infty$, et *NaN* (*Not a Number*) qui est par exemple renvoyé par $0/0$.

Arrondis. Pour un format donné, les nombres qu'on peut exprimer sous la forme ci-dessus sont dits *représentables* dans ce format. Ils forment un ensemble fini. Lorsque le nombre qu'on veut stocker n'est pas représentable, on utilise une *fonction d'arrondi* qui détermine quelle approximation représentable utiliser.

La norme IEEE-754 [65] définit plusieurs *modes d'arrondi* associant une fonction d'arrondi à un format flottant (les définitions suivantes ne prennent pas en compte les comportement exceptionnels comme les *overflows* présentés plus bas) :

- L'*arrondi vers le bas* ou *arrondi vers* $-\infty$ envoie un nombre réel r vers le plus grand nombre représentable inférieur ou égal à r, noté $\bigtriangledown(r)$.
- L'*arrondi vers le haut* ou *arrondi vers* $+\infty$ envoie un nombre réel r vers le plus petit nombre flottant supérieur ou égal à r, noté $\bigtriangleup(r)$.
- L'*arrondi vers 0* envoie r vers $\bigtriangledown(r)$ quand r est positif et vers $\bigtriangleup(r)$ quand r est négatif.
- L'*arrondi au plus proche* est le mode d'arrondi considéré par défaut. Comme son nom l'indique, il envoie r vers le nombre représentable le plus proche de r. Il y a une ambiguïté lorsque r est équidistant de deux nombres représentables : on dit que r est un *point milieu*. Une fonction d'arrondi au plus proche $\bigcirc^\tau$ dépend donc d'une *règle de bris d'égalité* τ, qui indique de quel côté arrondir chacun de ces points milieu (j'utiliserai la terminologie anglophone plus connue *tie-breaking rule*). La norme IEEE-754 définit deux règles en particulier :
  - Parmi les deux nombres représentables les plus proches, la règle *ties-to-even*, notée simplement *even*, choisit celui qui a une mantisse paire (se terminant par un 0 plutôt que par un 1 ; il y en a toujours exactement un parmi deux flottants consécutifs en base 2). On parle alors d'*arrondi au plus proche pair*, noté $\bigcirc^{even}(r)$.
  - La règle *away-from-zero* choisit celui des deux qui est le plus grand en valeur absolue.

Overflows. Un format de virgule flottante ne représente qu'un nombre fini de valeurs réelles : il a donc des valeurs mininale et maximale représentables (sans compter les deux infinis). Lorsque le nombre qu'on cherche à représenter (par exemple le résultat d'un calcul) est en dehors de ces valeurs extrêmes, on a ce qu'on appelle un *dépassement de capacité supérieur*. J'utiliserai le terme anglais plus concis *overflow*. Ce phénomène se retrouve dans plusieurs arithmétiques en précision finie et peut être géré de différentes manières. En virgule flottante,

la valeur renvoyée en cas d'*overflow* est généralement $+\infty$ ou $-\infty$ ou une des deux valeurs représentables extrêmes selon le mode d'arrondi.

Les *overflows* sont assez rares en virgule flottante vu que les valeurs extrêmes représentables sont très grandes en valeur absolue. Par exemple, en *binary64*, ces valeurs sont $\pm(2^{1024}-2^{971})$ (avec l'exposant maximal 1023 et tous les bits de la mantisse à 1) c'est-à-dire environ $\pm 10^{308}$. Mais si un *overflow* arrive tout de même, c'est souvent catastrophique. Une possibilité est de prouver qu'il ne peut pas y avoir d'*overflow* dans un programme considéré. Une méthode généralement efficace pour cela est l'interprétation abstraite [33].

1.3.2 Virgule fixe

La virgule fixe est l'arithmétique la plus souvent utilisée pour implémenter des filtres numériques dans des systèmes embarqués. Elle interviendra directement dans le chapitre 5. Elle est aussi couverte par l'arithmétique en précision finie générique de la section 1.3.3 qui sera utilisée dans les chapitres 3 et 4.

Il n'y a pas de description générale officielle pour la virgule fixe. Cependant, la norme IEEE-1666 standardisant SystemC, datant de 2005 [62] et révisée en 2011 [64], contient les propriétés attendues de la virgule fixe dans SystemC. La norme ISO/IEC TR 18037:2008 [66], qui décrit des extensions du langage C destinées aux systèmes embarqués, inclut aussi des spécifications concernant la virgule fixe. La virgule fixe utilise très principalement la base 2. C'est la seule base que je considère dans cette thèse (même si encore une fois, l'arithmétique générique de la section 1.3.3 couvre n'importe quelle base).

En virgule fixe, un nombre est seulement stocké sous la forme d'une mantisse entière m (qui peut être négative). En plus de décrire comment cet entier m est codé, notamment sur combien de bits, un format de virgule fixe comporte un *exposant implicite* ℓ. La valeur du nombre est alors le produit de m par le *facteur d'échelle implicite* 2^ℓ :

$$m \times 2^\ell \tag{1.6}$$

Ainsi, un algorithme en virgule fixe ne fait intervenir que les mantisses entières. Mais le développeur connaît les exposants implicites des formats utilisés dans le programme et c'est à lui de s'assurer que ces mantisses sont utilisées adéquatement (par exemple en ajoutant un décalage avant une addition pour mettre les deux termes au même format : on parle d'*alignement*).

Représentation en complément à deux pour la mantisse. On peut rencontrer différentes représentations pour la mantisse entière m. Une possibilité simple consiste à coder son signe sur un bit (comme ci-dessus pour les nombres flottants) puis sa valeur absolue en base 2 classique sur les bits restants. Mais la représentation en *complément à deux* définie ci-dessous est généralement préférée car ses opérations sont compatibles avec celles des entiers positifs habituels (en base 2 sans signe), c'est-à-dire qu'elles utilisent le même circuit. En outre, cela fait naturellement apparaître l'*overflow modulaire* qui sera au centre de la section 5.4. Dans cette thèse, je considère exclusivement la représentation en complément à deux.

Une mantisse m est codée sur w bits notés $m_0, \ldots, m_{w-1}$. La *largeur* w (notée ainsi d'après l'anglais *width*) est donnée par le format considéré. Dans la représentation habituelle des entiers positifs en base 2, l'entier $m_{w-1} \ldots m_1 m_{0_2}$ associé à ces bits est obtenu en multipliant chaque bit m_i par le facteur 2^i. La représentation en complément à deux est similaire, à part que le bit m_{w-1} de poids le plus fort est à la place multiplié par -2^{w-1}. La valeur de la mantisse m associée à ces bits est donc :

$$m = -m_{w-1}2^{w-1} + \sum_{i=0}^{w-2} m_i 2^i \tag{1.7}$$

Les nombres représentables ainsi sont donc les entiers entre -2^{w-1} et $2^{w-1}-1$ inclus, au lieu des entiers entre 0 et $2^w - 1$ inclus pour une représentation habituelle de nombre positif en base 2. On remarque que le codage des nombres entre 0 et $2^{w-1} - 1$ n'a pas changé (ce sont ceux dont le bit de poids fort est à 0 de toute façon, donc peu importe par quoi il est multiplié). En revanche, si un mot de w bits représentait un entier $x \geq 2^w$ en base 2 positive, il représente à la place le nombre $x - 2^w$ en complément à deux. On remarque aussi que le bit m_{w-1} fait office de bit de signe puisqu'il vaut 0 pour les nombres positifs et 1 pour les nombres négatifs (il est cependant considéré comme faisant partie de la mantisse, et d'ailleurs les autres bits ne représentent pas du tout la valeur absolue de m).

Format, *lsb*, *msb*. Un format de virgule fixe en complément à deux est donc caractérisé par à la fois l'exposant implicite ℓ et la largeur w de la mantisse. Je note $\mathbb{F}_{\ell,w}$ l'ensemble des nombres représentables dans un tel format, c'est-à-dire :

$$\mathbb{F}_{\ell,w} \triangleq \{m \times 2^\ell \mid m \in \mathbb{Z} \wedge -2^{w-1} \leq m \leq 2^{w-1} - 1\} \tag{1.8}$$

Par extension, j'utilise parfois aussi $\mathbb{F}_{\ell,w}$ pour désigner le format lui-même.

La figure 1.4 illustre la représentation d'un nombre dans le format correspondant à $\ell = -3$ et $w = 8$. Cette représentation consiste simplement en 8 bits notés $m_0, \ldots, m_7$. On rappelle que le bit initial m_7 indique le signe mais fait aussi partie de la mantisse. Le point montre la position de la virgule à titre indicatif, mais elle ne fait pas partie de la représentation puisqu'elle est implicite. Cette position est constante pour un format considéré, d'où le nom d'arithmétique en *virgule fixe*.

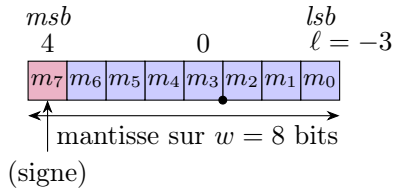


FIGURE 1.4 – Répartition des bits d'un nombre dans le format $\mathbb{F}_{-3,8}$

Au dessus de certains bits de la figure 1.4 est indiqué l'exposant correspondant au bit : par exemple, l'exposant 0 pour le bit juste avant la virgule. Le dernier bit est appelé le *lsb* (*Least Significant Bit*) et par extension, l'exposant

correspondant est aussi appelé le *lsb*. On remarque que ce *lsb* est en fait l'exposant implicite du format : $lsb = \ell = -3$. Le bit le plus à gauche, et par extension l'exposant associé, sont appelés le *msb* (*Most Significant Bit*). Ici, le *msb* vaut 4.

Un format de virgule fixe en complément à deux est complètement caractérisé par son *msb* et son *lsb*. D'ailleurs, un format est parfois défini comme la donnée d'un *msb* et un *lsb*. Notons que la convention que j'ai choisie (définir un format par un *lsb* ℓ et une largeur w) est équivalente vu qu'on a toujours la relation :

$$w = msb - lsb + 1 \tag{1.9}$$

En combinant les équations (1.6) et (1.7), on peut écrire la valeur du nombre représenté par les bits $m_0, \ldots, m_{w-1}$ en fonction du *lsb* et du *msb* du format :

$$-m_{w-1}2^{msb} + \sum_{i=0}^{w-2} m_i 2^{i+lsb} \tag{1.10}$$

Encore une autre convention usuelle (mais que je n'utilise pas dans ce manuscrit) consiste à représenter un format de virgule fixe sous la forme $\langle e, f \rangle$ où e est le nombre de bits avant la virgule implicite (partie entière) et f le nombre de bits après la virgule (partie fractionnaire). Le format illustré sur la figure 1.4 serait ainsi noté $\langle 5, 3 \rangle$. On retrouve les autres caractéristiques du format grâce aux relations $\ell = -f$, $w = e + f$ et $msb = e - 1$.

Exemple. Approximons π sur $w = 8$ bits avec pour exposant implicite $\ell = -3$, c'est-à-dire dans $\mathbb{F}_{-3,8}$. À partir de l'écriture binaire de π donnée dans l'équation (1.4), on introduit le facteur 2^{-3} et on arrondit au plus proche pour obtenir une mantisse entière :

$$\pi \approx 11001_2 \times 2^{-3} \tag{1.11}$$

On complète à gauche avec des 0 jusqu'à avoir 8 bits et on obtient la représentation en virgule fixe de la figure 1.5. Le bit initial faisant office de bit de signe est bien un 0 puisque le nombre qu'on représente est positif.

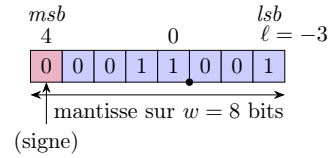


FIGURE 1.5 – Exemple : approximation de π dans $\mathbb{F}_{-3,8}$

On remarque qu'on a beaucoup de 0 initiaux qui ne sont pas nécessaires pour représenter l'approximation de π considérée. Quitte à disposer de 8 bits, si on diminue un peu le *lsb* ℓ, on peut avoir plus de bits sur la droite donc plus de précision, sans pour autant perdre d'information. Un meilleur *lsb* pour approcher π sur 8 bits est $\ell = -5$; on obtient alors la représentation de la figure 1.6.

Il reste un 0 initial qui n'est pas du tout inutile : il indique que le nombre est positif. Si on décalait encore le *lsb* en prenant $\ell = -6$, le bit d'exposant 1, valant actuellement 1, serait le nouveau bit initial donc multiplié par -2^1 au lieu de 2^1. Le laisser à 1 produirait un nombre négatif, mais le passer à 0 modifierait aussi

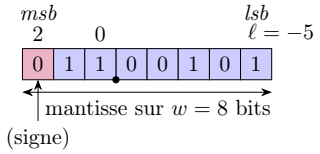


FIGURE 1.6 – Exemple : approximation de π dans $\mathbb{F}_{-5,8}$

grandement la valeur finale... Ainsi, si on essaie de représenter π dans $\mathbb{F}_{-6,8}$, on perd beaucoup d'information et on obtient une valeur complètement différente. Ceci se produit parce que π est en dehors des valeurs extrêmes représentables de $\mathbb{F}_{-6,8}$: c'est un *overflow*.

Arrondis. Les modes d'arrondi définis en section 1.3.1 s'appliquent aussi à la virgule fixe. Cependant, il faut noter que l'arrondi vers le bas est particulièrement fréquent en virgule fixe en complément à deux, car dans cette représentation il s'agit d'une simple troncature : on se contente d'effacer les bits au-delà de celui auquel on veut arrondir. En virgule flottante, c'est l'arrondi vers 0 qui s'obtient par troncature ; cet arrondi n'est généralement pas utilisé en virgule fixe, d'autant plus qu'il n'est pas compatible avec l'*overflow* modulaire comme on verra en section 5.4.1.

Overflows. Les *overflows* présentés en section 1.3.1 s'appliquent aussi à la virgule fixe, pour laquelle ils sont même une considération plus importante qu'en virgule flottante. En effet, les valeurs représentables extrêmes d'un format de virgule fixe sont souvent du même ordre que les valeurs qu'on aura effectivement besoin de représenter, donc on ne peut pas ignorer la possibilité d'un *overflow*. À titre indicatif, les valeurs extrêmes du fomat $\mathbb{F}_{\ell,w}$ sont $-2^{w+\ell-1}$ et $2^{w+\ell-1}-2^{\ell}$ (c'est-à-dire -2^{msb} et $2^{msb}-2^{lsb}$).

On rencontre plusieurs possibilités de gestion des *overflows* en virgule fixe. Les deux modes d'*overflow* les plus usuels sont l'*overflow saturé* et l'*overflow modulaire*. Le premier consiste à renvoyer la valeurs extrême qui a été excédée. Le second applique une opération de modulo mathématique afin de retomber sur une valeur représentable.

L'*overflow* modulaire sera détaillé en section 5.4.1. On verra qu'il est facile à implémenter pour la représentation en complément à deux (il revient à simplement supprimer les bits à gauche du *msb*) et présente des propriétés arithmétiques intéressantes. Par exemple, on peut (sous certaines conditions) garantir que le résultat final est correct même quand des *overflows* ont eu lieu dans les calculs intermédiaires (ceci est connu en traitement du signal sous le nom de *règle de Jackson* [68]).

Une autre approche usuelle consiste à choisir les formats de virgule fixe d'un programme donné de manière à éviter complètement les *overflows* dans ce programme. En effet, si on connaît des bornes sur les valeurs que peut prendre une variable donnée, on peut choisir un *msb* suffisamment grand pour que ces bornes soient comprises entre les valeurs extrêmes représentables. Notons que la correction fonctionnelle du programme repose alors souvent sur cette absence d'*overflows* : il faut donc bien s'assurer que cette absence est effectivement garantie.

Comparaison avec la virgule flottante. On relève trois différences principales dans la représentation d'un nombre en virgule fixe par rapport à la virgule flottante : la nature implicite de l'exposant, l'encodage de la mantisse (comme un entier en complément à deux, au lieu d'un bit de signe et des bits de fraction) et l'absence de valeurs particulières (infinis, zéros, NaN, dénormalisés).

L'intérêt principal de la virgule fixe est que les calculs ne font intervenir que des nombres entiers, pour lesquels on dispose d'unités de calcul (ALU) rapides et économes en termes à la fois de taille de circuit, d'énergie requise et de coût de production. En effet, les exposants implicites sont gérés par le développeur lors de la conception du programme, dans lequel n'apparaissent ensuite que les mantisses entières. Au contraire, la virgule flottante requiert un unité de calcul dédiée, qui doit gérer explicitement les exposants et les valeurs particulières. Ce matériel est présent dans tous les ordinateurs modernes, mais pas toujours dans les systèmes embarqués. De plus, la virgule fixe est plus flexible : n'importe quel format est possible tant que le développeur fait les ajustements appropriés, tandis que la virgule flottante est assujettie aux formats disponibles dans l'unité de calcul utilisée (généralement les formats exigés ou recommandés par la norme IEEE-754 [65]). La virgule fixe permet donc d'adapter les formats utilisés au programme considéré afin d'encore gagner en vitesse de calcul et en économie d'énergie. Dans le cadre des systèmes embarqués, on utilise donc majoritairement la virgule fixe.

En contrepartie, la virgule fixe demande un important travail de la part du développeur, travail laissé au matériel dans le cas de la virgule flottante : choix des formats (exposants et largeurs), alignement pour chaque opération (décalage des opérandes pour mettre la virgule à la même position) et parfois gestion de comportements exceptionnels (comme un *overflow* ou une division par zéro). En particulier, le choix des formats à utiliser pour les différentes variables d'un programme en virgule fixe est un problème important et délicat. En effet, on veut réduire autant que possible les largeurs des formats afin que les calculs impliquent moins de bits, devenant ainsi plus rapides et moins coûteux en énergie. Mais cela signifie augmenter le *lsb* ou diminuer le *msb* (équation (1.9)). La première possibilité entraîne une perte de précision, tandis que la seconde augmente la fréquence des *overflows*.

1.3.3 Arithmétique en précision finie générique et définitions communes aux différentes arithmétiques

Certains programmes sont conçus exclusivement dans l'optique du format *binary64* par exemple, ou encore sont seulement destinés à utiliser la virgule fixe avec des formats bien choisis pour chaque variable. Mais parfois, on veut étudier le comportement en précision finie d'un même programme pour différents choix d'arithmétiques et de formats possibles. On utilise alors une *arithmétique en précision finie générique* avec aussi peu d'hypothèses que possible.

Un *format générique*, noté $\mathbb{F}$, est simplement un sous-ensemble des nombres réels : $\mathbb{F} \subset \mathbb{R}$. Les éléments de $\mathbb{F}$ sont les nombres *représentables* dans ce format. Une fonction d'arrondi (ou juste un arrondi) vers ce format, notée $\bigcirc$, est simplement une fonction $\mathbb{R} \to \mathbb{F}$. On lui demande généralement d'être croissante et de laisser inchangés les nombres du format ($\forall r \in \mathbb{F}$, $\bigcirc(r) = r$).

Arrondi correct. L'implémentation en précision finie d'une fonction mathématique sur les réels vérifie la propriété d'*arrondi correct* si le résultat renvoyé est l'arrondi du résultat exact en précision infinie. Autrement dit, pour un arrondi considéré $\bigcirc : \mathbb{R} \to \mathbb{F}$ et un entier $n > 0$, l'implémentation $\widetilde{f} : \mathbb{F}^n \to \mathbb{F}$ d'une fonction $f : \mathbb{R}^n \to \mathbb{R}$ est *correctement arrondie* si :

$$\forall r_1, \ldots, r_n \in \mathbb{F}, \quad \widetilde{f}(r_1, \ldots, r_n) = \bigcirc(f(r_1, \ldots, r_n)) \tag{1.12}$$

La norme IEEE-754 [65] exige des implémentations correctement arrondies (pour chaque format flottant et mode d'arrondi défini par cette norme) de six opérations considérées comme élémentaires : l'addition, la soustration, la multiplication et la division de deux nombres, ainsi que la racine carrée et le *FMA* (*Fused Multiply-Add*, qui prend trois arguments x, y, z et renvoie $x \times y + z$).

La propriété d'arrondi correct permet de caractériser de manière unique le résultat que doit renvoyer une implémentation. De plus, pour un format flottant ou fixe, on sait donner une borne (relative ou absolue, ou une combinaison des deux selon les cas) sur l'erreur due à un seul arrondi. On peut alors étendre ces bornes à l'erreur sur une opération correctement arrondie.

Erreur locale. Même si on ne dispose pas d'une implémentation correctement arrondie d'une fonction, on peut généralement s'attendre à ce que l'erreur sur une seule application ne soit pas trop importante. Pour un format donné de virgule flottante ou fixe, on peut souvent déterminer une borne explicite sur cette erreur. On appelle ainsi *erreur locale* l'erreur due à un seul arrondi ou une seule application d'une fonction implémentée $\widetilde{f} : \mathbb{F}^n \to \mathbb{F}$ (par exemple une somme de produits ci-dessous). Comme mentionné en section 1.1, le problème est souvent la propagation d'erreurs d'arrondi au fil de nombreux calculs (quand les termes de chaque opérations sont des valeurs calculées déjà sujettes à des erreurs d'arrondi). Une grande partie du chapitre 4 consistera à exprimer l'erreur en sortie d'un filtre numérique, affectée par de nombreuses propagations et accumulations, en fonction seulement d'erreurs locales qui sont beaucoup plus facile à contrôler.

Somme de produits. Beaucoup d'implémentations de filtres numériques reposent sur l'opération de somme de produits ($f(x_1, \ldots, x_n, y_1, \ldots, y_n) = \sum_{i=1}^{n} x_i y_i$). Le chapitre 5 s'intéressera à des implémentations en virgule fixe de cette opération, certaines correctement arrondies, d'autres garantissant au moins une borne assez étroite sur l'erreur locale associée. Je noterai $\odot$ un produit scalaire ou matriciel en précision finie, calculé à l'aide d'une ou plusieurs implémentation(s) de somme de produits (par exemple $\boldsymbol{x} \odot \boldsymbol{y}$ ou $\boldsymbol{A} \odot \boldsymbol{B}$). D'ailleurs, je noterai aussi $\bigcirc(\boldsymbol{A})$ l'arrondi d'une matrice $\boldsymbol{A}$ coefficient par coefficient, c'est-à-dire dont le coefficient en position (i,j) est $\bigcirc(\boldsymbol{A}_{ij})$.

1.4 Traitement du signal et filtre numérique

Les nombreuses applications des filtres numériques, notamment dans des contextes critiques, ont été évoquées dans la section 1.1. La section courante introduit les notions principales sur les filtres numériques qui seront utilisées tout au long de ce manuscrit.

1.4.1 Signal

Les signaux sont utilisés pour décrire l'évolution au cours du temps d'une grandeur physique : son, vitesse, position, tension électrique, température, etc. Ainsi, un signal x associe une valeur $x(t)$ à chaque instant t considéré. On parle de *signal analogique* quand les valeurs prises par le signal sont directement les valeurs du phénomène physique mesuré. Par exemple, la tension électrique observée à l'écran d'un oscilloscope est un signal analogique.

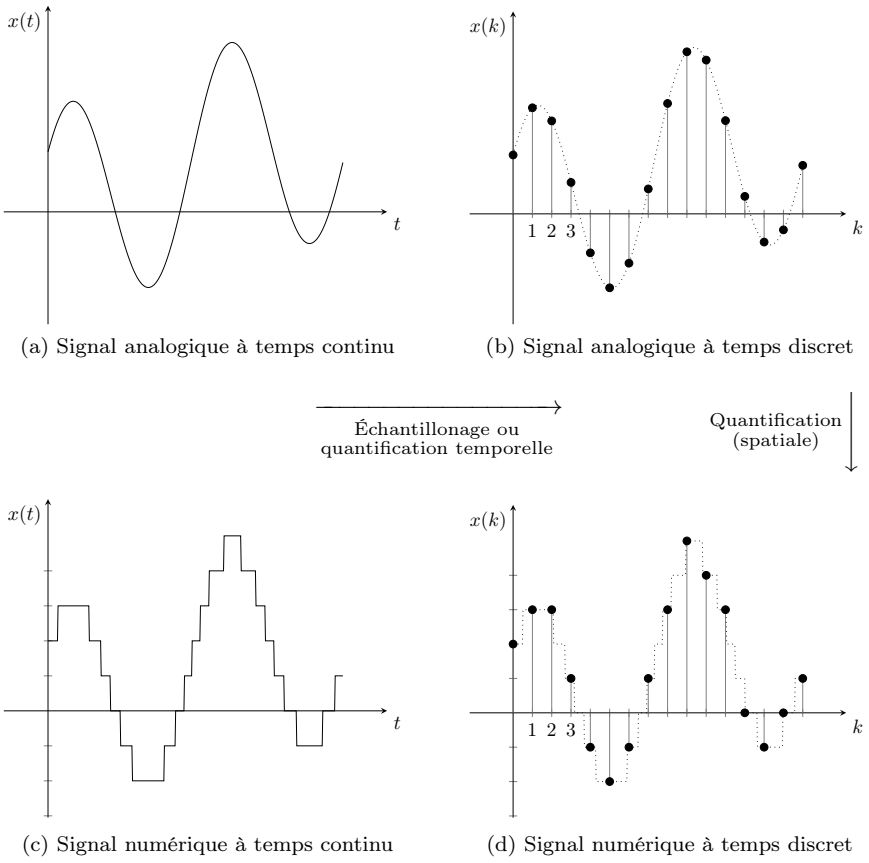
Un signal analogique x est généralement *à temps continu*, c'est-à-dire que le temps t parcourt un domaine continu (souvent un intervalle de l'ensemble $\mathbb{R}$ des nombres réels) comme illustré par la figure 1.7a. Au contraire, un signal *à temps discret* est une suite de valeurs $x(k)$ où le temps est représenté par un indice k entier comme sur la figure 1.7b (l'ensemble d'indices est souvent les entiers relatifs $\mathbb{Z}$ ou les entiers naturels $\mathbb{N}$; on rencontre aussi des ensembles finis comme $\{0, 1, \ldots, N\}$ pour un $N \in \mathbb{N}$ donné). Un signal à temps continu x_c est souvent échantillonné en un signal à temps discret x_d. On parle alors de *quantification temporelle*. La plupart du temps, on utilise un échantillonnage à temps constant : $x_d(k) = x_c(kT_e)$ pour une période d'échantillonnage T_e donnée.

On parle de *signal numérique* quand les valeurs prises par le signal sont représentées de manière codée, par exemple avec des nombres à virgule flottante ou fixe dans un ordinateur. Les signaux numériques sont plus faciles à reproduire et peuvent être traités par des programmes informatiques. Les figures 1.7c et 1.7d donnent des exemples de signaux numériques à temps continu ou discret [3]. On parle de *quantification spatiale*, ou simplement *quantification*, quand un signal analogique est transformé en signal numérique par approximation de ses valeurs.

Convention. Comme leur nom l'indique, les filtres numériques traitent des signaux qui sont numériques. De plus, ces signaux sont à temps discret. Dans la suite, le mot *signal* signifiera donc toujours *signal numérique à temps discret*.

Comme un format de précision finie peut être vu comme un sous-ensemble de $\mathbb{R}$, on utilise généralement les nombres réels pour représenter les valeurs des signaux numériques, en gardant à l'esprit que ces valeurs peuvent être sujettes à des erreurs d'arrondi. Cependant, on s'intéresse souvent aussi aux signaux idéaux qui sont définis et calculés dans $\mathbb{R}$ avec une précision infinie. Le comportement de ces signaux est plus facile à caractériser mathématiquement, notamment parce qu'on dispose alors de propriétés comme l'associativité ou la commutativité de certaines opérations, auxquelles on n'a pas accès en précision finie. Pour certains domaines non critiques, on peut s'en tenir là car l'erreur entre ces signaux théoriques et la réalité n'est pas très grave. Mais même pour les domaines critiques où les erreurs d'arrondis doivent soigneusement être prises en compte, il est utile de bien connaître les signaux idéaux : ils servent de modèles auxquels on pourra ensuite comparer les signaux calculés en pratique en précision finie.

3. Dans la majorité des cas, un signal analogique est à temps continu et un signal numérique est à temps discret. À tel point qu'on rencontre dans la littérature des définitions de signal analogique requérant un temps continu, et des définitions de signal numérique requérant un temps discret. Néanmoins, il existe des exceptions : par exemple, un signal analogique à temps discret formé par un circuit électrique avec une horloge.



(a) Signal analogique à temps continu

(b) Signal analogique à temps discret

(c) Signal numérique à temps continu

(d) Signal numérique à temps discret

FIGURE 1.7 – Quantification des signaux

Je m'intéresse à deux familles de signaux théoriques en précision infinie. Un *signal scalaire* x, aussi appelé *signal réel*, prend directement des valeurs réelles : $x(k) \in \mathbb{R}$. Un *signal vectoriel* $\boldsymbol{x}$ (noté en gras par convention) a pour valeurs des vecteurs de nombres réels (par exemple pour décrire une position) : $\boldsymbol{x}(k) \in \mathbb{R}^n$ pour un certain n. On dit que $\boldsymbol{x}$ est de taille n. Par extension de la notation $\boldsymbol{v}_i$ représentant la composante d'indice i d'un vecteur $\boldsymbol{v}$, on notera $\boldsymbol{x}_i$ le signal tel que :

$$\forall k, \quad \boldsymbol{x}_i(k) \triangleq \boldsymbol{x}(k)_i \tag{1.13}$$

où le symbole $\triangleq$ signifie « égal par définition ». On remarque que $\boldsymbol{x}_i$ est un signal scalaire puisque $\boldsymbol{x}_i(k) \in \mathbb{R}$. Ainsi, un signal vectoriel de taille n peut aussi être vu comme un vecteur de n signaux scalaires.

1.4.2 Filtre numérique

Un filtre numérique $\mathscr{H}$ est une fonction mathématique qui transforme un signal d'entrée u en un signal de sortie y, comme représenté par la figure 1.8. On écrit alors $y = \mathscr{H}\{u\}$. Il s'agit d'une transformation du signal entier, pas

seulement point à point : la sortie $y(k)$ à l'instant k peut dépendre de toutes les valeurs du signal d'entrée u, au lieu de seulement la valeur $u(k)$ à cet instant. Bien sûr, en pratique, un filtre calculé en temps réel ne peut pas dépendre des entrées futures : $y(k)$ est donc seulement autorisé à dépendre des entrées $u(l)$ pour $l \leq k$. On dit alors que le filtre est *causal* (voir la section 2.2.2).

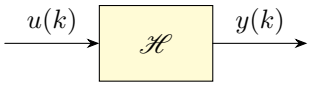


FIGURE 1.8 – Filtre numérique

On parle de filtre *SISO* (*Single-Input Single-Output*) quand l'entrée u et la sortie y sont des signaux scalaires, et de filtre *MIMO* (*Multiple-Input Multiple-Output*) quand il s'agit de signaux vectoriels, qui sont alors notés $\boldsymbol{u}$ et $\boldsymbol{y}$. Je noterai toujours n_u la taille du signal $\boldsymbol{u}$ (c'est-à-dire la taille des vecteurs $\boldsymbol{u}(k)$) et n_y celle de $\boldsymbol{y}$. J'écrirai parfois que le filtre a n_u entrées et n_y sorties, ce qui est synonyme d'une entrée et une sortie vectorielles de tailles respectives n_u et n_y.

En contrôle-commande, on modélise un système à contrôler (*plant* en anglais) par un filtre. Un second filtre, le *contrôleur*, reçoit en entrées à la fois des valeurs de référence $r(k)$ et les sorties $y(k)$ du système à contrôler (on peut grouper les deux en un seul signal d'entrée vectoriel), et en déduit les entrées $u(k)$ à lui fournir pour rapprocher les sorties des valeurs de référence. Ceci est illustré par la figure 1.9. Le tout est appelé un *système en boucle fermée*. Je n'ai pas étudié directement ces systèmes, mais beaucoup d'éléments de ma formalisation en Coq peuvent leur être indirectement appliqués. Des travaux s'intéressant spécifiquement à ces systèmes seront présentés en section 1.5. Étendre ma formalisation pour prouver des propriétés propres à ces systèmes sera laissé en perspective en section 6.3.2.

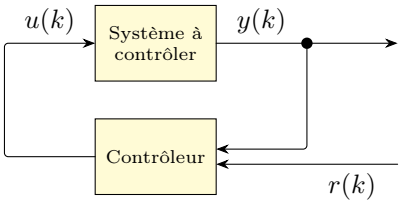


FIGURE 1.9 – Contrôle-commande : système en boucle fermée

On dit qu'un filtre est *linéaire invariant dans le temps*, abrégé en *LTI* (*Linear Time-Invariant*), s'il est linéaire :

$$\forall u_1, u_2, \forall a, b \in \mathbb{R}, \quad \mathscr{H}\{a\,u_1 + b\,u_2\} = a\,\mathscr{H}\{u_1\}(k) + b\,\mathscr{H}\{u_2\} \tag{1.14}$$

et que lorsqu'on retarde une entrée d'un certain nombre d'unités de temps, la sortie est retardée d'autant. Une définition plus détaillée est donnée en section 2.2.1 ; et les opérations sur les signaux utilisées (multiplication par a du signal u_1, addition de deux signaux...) sont définies en section 2.1.3.

Cette thèse étudie exclusivement les filtres LTI, qui sont très courants en traitement du signal (filtres à réponse impulsionnelle finie (FIR) ou infinie (IIR) définis en section 2.2.3) et en contrôle-commande (on parle de commande linéaire quand le contrôleur est LTI ; on modélise généralement aussi le système à contrôler comme un filtre LTI).

La présentation des filtres donnée ici est dite dans le *domaine temporel*. Une approche complémentaire courante en traitement du signal consiste à étudier ces mêmes filtres dans le *domaine fréquentiel*, avec notamment la notion de fonction de transfert. Le domaine fréquentiel n'est pas du tout étudié dans cette thèse (qui se concentre sur les algorithmes calculés en précision finie dans le domaine temporel). La section 2.2.6 en proposera cependant un bref aperçu. De plus, des formalisations en HOL s'intéressant au domaine fréquentiel [104, 105] seront présentées en section 1.5.

1.4.3 Exemples de filtres LTI

Les exemples suivants de filtres sont tous LTI.

Filtre lisseur. Un exemple basique de filtre SISO est le *filtre lisseur* suivant. Chaque sortie est la moyenne entre l'entrée courante et l'entrée précédente. Autrement dit, le signal de sortie y est défini à partir du signal d'entrée u par la relation suivante :

$$\forall k, \quad y(k) = \frac{u(k) + u(k-1)}{2} \tag{1.15}$$

Leaky integrator. Ce filtre SISO calcule rapidement une approximation de la moyenne des $\frac{1}{\alpha}$ dernières entrées, pour un α donné tel que $0 < \alpha < 1$. Il est défini par la relation de récurrence suivante :

$$\forall k, \quad y(k) = \alpha\, u(k) + (1-\alpha)\, y(k-1) \tag{1.16}$$

Remarque : initialisation. Une définition récursive comme celle du *leaky integrator* requiert une initialisation. On verra en section 2.1.1 qu'on suppose que les signaux considérés sont toujours nuls pour les indices strictement négatifs. En particulier, toutes les relations récursives définissant des filtres peuvent être initialisées par :

$$\forall k < 0, \quad y(k) = 0 \tag{1.17}$$

Moteur à induction. Voici un autre filtre SISO qui est utilisé dans un modèle de moteur à induction (il apparaît dans [67, section 2.3], qui l'a lui-même extrait d'un exemple de [94]) :

$$y(k) = 2{,}813u(k) - 0{,}0163u(k-1) - 1{,}872u(k-2) - 1{,}068y(k-1) - 0{,}1239y(k-2) \tag{1.18}$$

Exemple jouet récurrent. Voici un exemple de filtre MIMO à deux entrées et une sortie (c'est-à-dire une entrée vectorielle $\boldsymbol{u}$ de taille 2 et une sortie vectorielle

$\boldsymbol{y}$ de taille 1). Le signal de sortie a donc une seule composante $\boldsymbol{y}_1$, qui est définie récursivement par :

$$\boldsymbol{y}_1(k) = 2\,(\,3\,\boldsymbol{u}_1(k) + \boldsymbol{u}_2(k-2)\,) - \frac{1}{2}\,\boldsymbol{y}_1(k-1) \tag{1.19}$$

On étend maintenant cet exemple à un filtre MIMO à deux entrées et deux sorties, en définissant une seconde composante de sortie $\boldsymbol{y}_2$:

$$\begin{cases} \boldsymbol{y}_1(k) = 2\,(\,3\,\boldsymbol{u}_1(k) + \boldsymbol{u}_2(k-2)\,) - \frac{1}{2}\,\boldsymbol{y}_1(k-1) \\ \boldsymbol{y}_2(k) = 3\,(\,-4\,(\,7\,\boldsymbol{u}_2(k) + \boldsymbol{u}_2(k-1)\,) - \frac{1}{2}\,\boldsymbol{y}_1(k-1) + \frac{1}{5}\,\boldsymbol{y}_2(k-1)\,) \end{cases} \tag{1.20}$$

Cet exemple jouet servira d'exemple récurrent dans les chapitres 2 et 3.

1.4.4 Filtre modèle et filtre implémenté

On parle de *filtre modèle* quand un filtre est défini par une relation mathématique en précision infinie. Les exemples de filtres ci-dessus sont des filtres modèles. Notons que les filtres modèles que je considère traitent déjà des signaux à temps discret. D'autres approches existantes partent d'un filtre modèle dont les signaux d'entrée et sortie sont à temps continu, et étudient alors leur échantillonnage en signaux à temps discret (comme on a vu en figure 1.7).

Lorsqu'on s'intéresse au filtre calculé en pratique par un ordinateur, à l'aide d'une arithmétique en précision finie (section 1.3), on dit que c'est un *filtre implémenté*. On note $\mathscr{H}^*$ un tel filtre et y^* (ou $\boldsymbol{y}^*$ si le filtre est MIMO) sa sortie pour signaler que des arrondis peuvent avoir lieu [4], comme illustré par la figure 1.10.

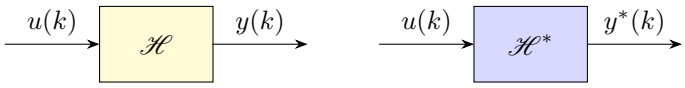


FIGURE 1.10 – Filtre modèle (à gauche) et filtre implémenté (à droite)

Exemple de filtre implémenté. On peut s'intéresser au filtre implémenté obtenu en calculant l'équation (1.16) du *leaky integrator* avec des nombres flottants *binary64* et un arrondi au plus proche pair (présentés en sections 1.3.1 et 1.3.3 mais les détails n'ont pas d'importance : ce qui compte ici est d'avoir choisi un format et un arrondi particuliers). On note $\oplus$, $\ominus$ et $\otimes$ les opérations basiques correctement arrondies correspondantes. Le filtre implémenté considéré est alors défini par la relation :

$$\forall k, \quad y^*(k) = \alpha^* \otimes u(k) \oplus (1 \ominus \alpha^*) \otimes y^*(k-1) \tag{1.21}$$

où α^* est l'arrondi de α vers le format choisi. En effet, ce nombre doit lui aussi être manipulé par le programme calculant le filtre. On parle de *quantification* du coefficient α (*quantization* en anglais). Notons que si α était déjà dans le format alors on a simplement $\alpha^* = \alpha$.

4. L'entrée d'un filtre implémenté est encore notée u (ou $\boldsymbol{u}$ dans le cas MIMO), bien qu'elle soit aussi composée de nombres dans un format de précision finie. En effet, comme les valeurs $u(k)$ ont été fournies directement dans un tel format, le filtre $\mathscr{H}^*$ n'a pas besoin de les arrondir lui-même donc ne les marque pas comme des valeurs approchées.

Pour simplifier l'exemple, on a choisi le même format et le même arrondi pour toutes les opérations. Notons que quand on utilise de la virgule fixe, on choisit souvent des formats différents pour chaque opération.

Propagation des erreurs. Le problème de propagation et d'accumulation des erreurs d'arrondi dans les filtres, mentionné en section 1.1, est bien visible dans l'équation (1.21). La nouvelle sortie $y^*(k)$ est non seulement calculée par des opérations sujettes à des erreurs d'arrondi, mais dépend aussi de la sortie précédente $y^*(k-1)$, qui est elle-même calculée approximativement en utilisant la valeur approchée $y^*(k-2)$, et ainsi de suite.

Erreur finale. Pour un filtre modèle $\mathscr{H}$ et un filtre implémenté $\mathscr{H}^*$ considérés, on note $\Delta y(k)$ l'*erreur finale* au temps k entre leurs sorties respectives y et y^* pour une même entrée u :

$$\forall k, \quad \Delta y(k) \triangleq y^*(k) - y(k) \tag{1.22}$$

On pourra ainsi étudier le *signal d'erreur finale* Δy (ou $\boldsymbol{\Delta y}$ pour un filtre MIMO) dans le chapitre 4. Un objectif sera notamment de trouver une borne M la plus petite possible de ce signal d'erreur ($\forall k$, $|\Delta y(k)| \leq M$) afin d'assurer que la sortie implémentée $\boldsymbol{y}$ reste proche de la sortie modèle y.

Diverses implémentations d'un filtre modèle. À partir d'un même filtre modèle, on peut définir de nombreux filtres implémentés, par exemple en choisissant différents formats de précision finie. Mais même si on se restreint au même format, on peut obtenir des filtres implémentés distincts si on change la structure des opérations. Par exemple, le *leaky integrator* modèle peut aussi être décrit par la relation :

$$\forall k, \quad y(k) = \alpha\,(u(k) - y(k-1)) + y(k-1) \tag{1.23}$$

En effet, les règles sur l'addition, la soustraction et la multiplication dans $\mathbb{R}$ assurent que la sortie $y(k)$ obtenue est la même que pour l'équation (1.16). En revanche, le filtre implémenté défini par :

$$\forall k, \quad y^*(k) = \alpha^* \otimes (u(k) \ominus y^*(k-1)) \oplus y^*(k-1) \tag{1.24}$$

n'est pas le même que celui de l'équation (1.21), bien que les opérations soient toujours en *binary64*. En effet, les calculs en précision finie ne partagent généralement pas les règles d'associativité ou distributivité qu'on connaît dans $\mathbb{R}$. Lorsque les calculs intermédiaires sont différents, les arrondis qui peuvent avoir lieu à chaque opération risquent de mener à des résultats différents pour $y^*(k)$.

Inversement, un même filtre implémenté peut techniquement correspondre à plusieurs filtres modèles. Par exemple, le filtre implémenté de l'équation (1.21) peut être associé aux filtres modèles définis par l'équation (1.16) pour différentes valeurs initiales de α s'arrondissant toutes vers le même α^*.

Mais la plupart du temps, on considère d'abord un filtre modèle désiré, et on s'intéresse aux différents filtres implémentés correspondants. Pour explorer ceux-ci, on va utiliser la notion de *réalisation* en section 1.4.5.

La plupart des propriétés que peut avoir un filtre, par exemple le fait d'être LTI, concernent les filtres modèles. En effet, les calculs en précision finie empêchent généralement d'avoir des propriétés comme la linéarité. À la place, on considère parfois qu'un filtre implémenté est LTI si le filtre modèle pour lequel il a été construit est LTI.

Convention. Lorsque j'écris simplement *filtre*, cela sous-entend un *filtre modèle*. Si je parle d'un *filtre implémenté*, c'est explicitement mentionné.

1.4.5 Réalisation d'un filtre

Pour un filtre modèle considéré, on a vu qu'on peut obtenir divers filtres implémentés en faisant varier les formats de précision finie utilisés ou la structure des calculs. Une *réalisation* de filtre est à mi-chemin entre un filtre modèle et un filtre implémenté : il s'agit d'une relation entrée-sortie définissant un filtre et imposant la structure des calculs, c'est-à-dire la succession de calculs intermédiaires à effectuer, mais n'indiquant pas les formats à utiliser. En terme informatique, une réalisation n'est ni plus ni moins qu'un algorithme où les formats de précision finie ne sont pas mentionnés.

Exemples. Les relations de la section 1.4.3 définissant des filtres modèles suggèrent naturellement des réalisations correspondantes. Par exemple, à partir de l'équation (1.16) on obtient la réalisation définie par l'algorithme 1.

Algorithme 1 : Réalisation du *leaky integrator*

Entrée : signal u
Sortie : signal y (par convention $y(k) = 0$ pour $k < 0$)
foreach $k \geq 0$ **do**
  $y(k) \leftarrow \alpha\, u(k) + (1 - \alpha)\, y(k-1)$
end

Mais en partant de l'équation (1.23), on peut obtenir une réalisation différente du *leaky integrator*, donnée par l'algorithme 2. Encore une fois, les équations (1.16) et (1.23) sont équivalentes mathématiquement et définissent le même filtre modèle, mais les réalisations associées sont distinctes : ce n'est pas le même algorithme.

Algorithme 2 : Une autre réalisation du *leaky integrator*

Entrée : Signal u
Sortie : Signal y (par convention $y(k) = 0$ pour $k < 0$)
foreach $k \geq 0$ **do**
  $y(k) \leftarrow \alpha\, (u(k) - y(k-1)) + y(k-1)$
end

Filtre réalisable. Un filtre modèle est *réalisable* s'il existe au moins une réalisation de ce filtre. Les filtres de la section 1.4.3 sont tous réalisables. Un exemple de filtre non réalisable est le filtre défini par la relation entrée-sortie : $\forall k,\ y(k) = u(k+1)$. Mathématiquement, c'est bien un filtre, c'est-à-dire une fonction qui transforme des signaux en signaux. Mais on ne peut pas construire

ce filtre en pratique puisqu'au temps k, on ne connaît pas encore $u(k+1)$. Remarquons que ce filtre ne vérifie pas la propriété de *causalité* mentionnée en section 1.4.2 et détaillée en section 2.2.2.

Multiples réalisations puis filtres implémentés. À un même filtre modèle réalisable peuvent être associées une infinité de réalisations (même si souvent, le filtre modèle est défini par une relation entrée-sortie qui suggère une réalisation en particulier). Il suffit en effet de modifier la structure des calculs d'une réalisation pour obtenir une réalisation différente du même filtre modèle, comme dans l'exemple ci-dessus. Ensuite, une réalisation donnée peut produire une infinité de filtres implémentés selon les choix de format et d'arrondi pour chaque calcul apparaissant dans la réalisation (voire des choix parmi différents algorithmes possibles pour un calcul plus complexe comme un somme de produits). Ceci est illustré par la figure 1.11.

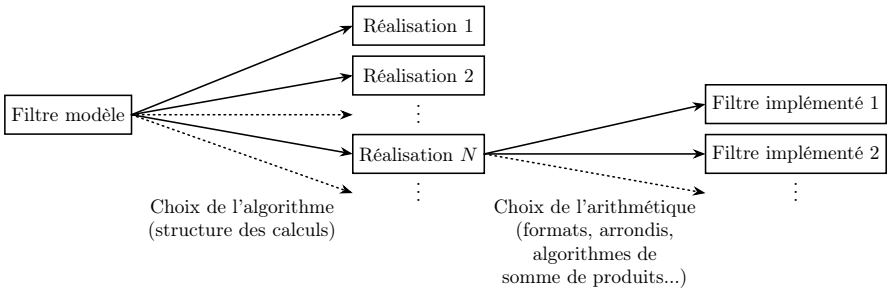


FIGURE 1.11 – Choix d'une réalisation puis d'un filtre implémenté

Les divers filtres implémentés possibles pour un même filtre modèle se distinguent par de nombreuses caractéristiques comme les complexités en temps et en mémoire, le coût matériel, la taille du circuit dans un système embarqué, la consommation d'énergie, le comportement numérique (à quel point la précision finie impacte la précision des résultats), le temps de développement nécessaire, etc. Selon le contexte dans lequel le filtre implémenté sera utilisé, on accorde plus ou moins d'importance à chacun de ces critères. L'enjeu est donc de trouver un bon compromis pour le contexte considéré. Or plusieurs des caractéristiques ci-dessus dépendent partiellement ou totalement de la réalisation utilisée. Le choix de la réalisation est donc une étape très importante lorsqu'on souhaite implémenter un filtre modèle donné.

Il existe de nombreuses familles de réalisations bien documentées, parmi lesquelles on peut chercher la meilleure réalisation selon le contexte. Quelques unes des familles les plus connues de réalisations de filtres LTI sont présentées en section 2.3.

1.5 État de l'art

Cette section présente un état de l'art des applications des différentes familles de méthodes formelles aux filtres numériques LTI calculés en précision finie.

Interprétation abstraite [33]. En 2004, Feret [38] propose une méthode pour prouver l'absence d'erreur d'exécution dans un programme en C implémentant un filtre numérique en virgule flottante. Il se base sur l'interpréteur abstrait développé par Blanchet *et al.* [16] pour vérifier de nombreux logiciels critiques en virgule flottante dans des systèmes embarqués, qui deviendra plus tard l'interpréteur Astrée [34]. Feret observe que la plupart des *warnings* signalés par cet analyseur sont des risques d'*overflow* dans des programmes de filtre numérique. En effet, les domaines utilisés ne sont pas assez précis pour cette catégorie de programme. Il résout ce problème en décrivant comment construire, pour un filtre LTI donné, un domaine abstrait plus précis spécifiquement pour ce filtre. Il illustre cette méthode sur un filtre passe-haut et un filtre du second ordre. Grâce à ce domaine, presque tous les *warnings* de l'interpréteur abstrait utilisé [16] ont disparu. Cette approche ne garantit cependant rien sur la valeur de la sortie, et le cas de la virgule fixe n'est pas étudié.

Preuve interactive avec HOL [93]. Akbarpour *et al.* [3] proposent en 2005 la première formalisation de l'arithmétique en virgule fixe en HOL, inspirée de plusieurs formalisations préexistantes de la virgule flottante [50, 25]. Ils définissent différents formats de virgule fixe et modes d'arrondi. Ils formalisent aussi la gestion d'exceptions comme les *overflows* ou les opérations invalides. Ils analysent les erreurs d'arrondi dans les opérations basiques (addition, soustraction, multiplication et division). Puis ils utilisent cette formalisation pour comparer des implémentations en virgule flottante et en virgule fixe. En exemple d'application, ils considèrent un intégrateur numérique, c'est-à-dire un filtre LTI défini par la relation suivante où a est un coefficient constant (similaire au *leaky integrator* de la section 1.4.3) :

$$y(k) = u(k-1) + ay(k-1) \tag{1.25}$$

Ils bornent l'erreur sur sa sortie résultant du passage d'une implémentation en flottant à une implémentation en fixe.

En 2007, Akbarpour et Tahar [2] formalisent en HOL des analyses d'erreurs plus approfondies de filtres LTI. Ils définissent le filtre modèle en précision infinie et les implémentations en virgule flottante et en virgule fixe d'un même filtre. Ils s'intéressent à l'erreur finale entre les sorties de chaque implémentation et la sortie idéale. Ils expriment cette erreur pour trois réalisations de filtre LTI : la forme directe I (section 2.3.1), ainsi que les décompositions parallèle et en cascade (section 2.3.3) en sous-filtres d'ordre au plus 2. Cette expression prend en compte à la fois la quantification des coefficients, les erreurs dues aux opérations et la propagation des erreurs. Ces travaux sont donc assez similaires aux filtres d'erreurs présentés en sections 4.1 et 4.2, même si les réalisations considérées et la présentation sont différentes. En revanche, ils ne bornent pas l'expression qu'ils ont obtenue pour l'erreur finale. Notons que l'utilisation du théorème du *Worst-Case Peak Gain* (section 4.3) pour borner cette erreur date seulement de 2013 [58].

Des travaux plus récents s'intéressent à la formalisation des filtres dans le domaine fréquentiel (présenté rapidement en section 2.2.6). Ainsi, Siddique *et al.* [104] formalisent la transformée en Z en HOL Light en 2014, et l'utilisent pour obtenir la fonction de transfert et la réponse fréquentielle d'un filtre IIR donné sous forme d'équation aux différences. Ils étendent cette formalisation en 2018, avec notamment une preuve d'unicité [105].

***Model checking* [7].** DSVerifier [5] (Digital-Systems Verifier, 2015) [67] est un outil vérifiant et certifiant automatiquement des propriétés d'implémentations de filtres numériques en précision finie, à l'aide de *model checking* borné [30] reposant sur la résolution SMT (*Satisfiability Modulo Theories*). DSVerifier se présente sous la forme d'un module greffé à un *model checker* efficace préexistant [32] : il reçoit une spécification de système numérique et génère un fichier de code C vérifiable par un tel *model checker*. Les propriétés qui peuvent être vérifiées sont l'absence d'*overflows*, l'absence d'oscillations de cycles limites pour des entrées nulles, la stabilité numérique des pôles ou des zéros de la fonction de transfert (pour un filtre à minimum de phase), et enfin le respect de contraintes temporelles. Pour utiliser l'outil, il faut lui fournir un filtre (sous la forme des coefficients de sa fonction de transfert), un format de virgule fixe (sous la forme $\langle e, f \rangle$ du nombre de bits avant et après la virgule), des bornes sur les entrées, la réalisation choisie parmi celles supportées (formes directes I et II et II transposée présentées en section 2.3, ainsi que plusieurs formes directes delta ou cascade non abordées dans ce manuscrit) et enfin la propriété à vérifier parmi celles listées ci-dessus. DSVerifier certifie alors la propriété, ou bien produit un contre-exemple. Cet outil s'appuie entre autres sur des études en 2013 et 2014 d'applications de *model checking* borné avec résolution SMT aux filtres numériques pour une partie des propriétés et réalisations listées ci-dessus [1, 13, 14]. En 2019, DSVerifier $v2.0$ [28] ajoute notamment la vérification de la robustesse d'un système LTI en boucle fermée (c'est-à-dire à quel point le système reste stable en dépit d'erreurs de quantification ou d'arrondi ou d'autres perturbations), ainsi que la quantification de l'erreur sur la sortie d'un tel système. Des spécifications de performance comme le temps de montée (*settling-time*) et le dépassement (*overshoot*) de la réponse impulsionnelle sont ajoutées en 2020 [26]. Par ailleurs, DSVerifier est appliqué au cas spécifique des véhicules aériens sans pilote en 2018 [27]. Ainsi, il s'agit d'un outil efficace pour vérifier des propriétés numériques d'une implémentation de filtre donnée dont les valeurs des coefficients sont explicitement fournies et utilisant un seul format de virgule fixe donné pour tous les calculs, plutôt que des propriétés générales communes à toute une famille de filtres.

Autres preuves automatiques. En 2016, Wang *et al.* [112] vérifient la robustesse de systèmes de contrôle LTI en boucle fermée, à la fois au niveau du modèle et de l'implémentation. Ils utilisent un solveur d'optimisation SDP (Semi-Définie Positive ; on parle aussi de *semidefinite programming* en anglais) pour obtenir des marges vectorielles (des indicateurs de stabilité similaires aux marges de phase et de gain) et des invariants sur les variables d'état. Ils implémentent un outil qui détermine automatiquement la robustesse d'un système SISO en produisant des surapproximations des marges de phase et de gain. Ils prennent en compte les effets de l'arithmétique en virgule flottante, à la fois dans le solveur d'optimisation SDP et dans le système numérique. La solution choisie pour le solveur est une vérification *a posteriori* de sa sortie à l'aide d'un algorithme prouvé en Coq. Pour le système lui-même, des bornes sont calculées à partir des erreurs relatives sur les opérations ; l'absence d'*overflows* est aussi prouvée, tout comme le fait que les *underflows* sont négligeables pour les nombres d'itérations

5. www.dsverifier.org

qu'on peut rencontrer en pratique. En revanche, le cas de la virgule fixe n'est pas considéré.

En 2016 également, Park *et al.* [98] utilisent une combinaison d'exécution symbolique et d'optimisation convexe pour vérifier qu'un filtre implémenté en C correspond bien à une spécification donnée. Les filtres LTI considérés sont sous forme de *State-Space* (section 3.1). Le principe est de prouver que le filtre calculé par le code est bien le filtre spécifié initialement, même lorsqu'un générateur de code a optimisé la succession de calculs à effectuer, ce qui peut complètement changer les matrices de coefficients et les variables d'état du *State-Space*. En revanche, les erreurs d'arrondi sont ignorées pour le moment : les calculs sont supposés exacts. Dans un premier temps, le *State-Space* correspondant au code C étudié est déterminé par exécution symbolique (outil PathCrawler). Ensuite, on s'efforce de montrer que ce *State-Space* décrit le même filtre que le *State-Space* initial : on cherche une matrice $\boldsymbol{T}$ permettant de passer d'un *State-Space* à l'autre par changement de base (section 3.4.2). La recherche d'une telle matrice est alors formulée d'une part comme un problème de satisfiabilité, et d'autre part comme un problème d'optimisation convexe. Les performances sont évaluées en utilisant le solveur SMT CVC4 ou le solveur d'optimisation convexe CVX. L'optimisation convexe est la seule méthode pour laquelle le temps d'exécution reste raisonnable pour des *State-Space* dont le nombre de variables d'état va jusqu'à $n_x = 14$. Puis en 2017, les mêmes auteurs étendent ces travaux pour prendre en compte les erreurs d'arrondi en virgule flottante [99]. Les relations du *State-Space* issu du code contiennent désormais des termes d'erreurs. Une analyse d'erreurs sur les opérations flottantes fournit des bornes relatives ou absolues sur ces termes. On cherche alors une relation de changement de base approximative avec une certaine tolérance spécifiée. Les termes d'erreurs supplémentaires rendent la résolution trop difficile pour les solveurs SMT. En revanche, l'approche par optimisation convexe fonctionne toujours. Finalement, ce travail prend bien en compte les erreurs d'arrondi dans une itération du filtre (calcul du nouveau vecteur d'état et de la sortie à un temps k). Cependant, il ne tient pas compte de l'accumulations possibles des erreurs au fil des itérations, et n'étudie pas non plus le cas de la virgule fixe.

J'ai restreint cette section aux travaux les plus similaires à ma thèse : ceux qui font intervenir à la fois méthodes formelles, filtres numériques LTI et arithmétique en précision finie. Des références de traitement du signal, incluant des analyses d'erreurs d'arrondi mais sans méthodes formelles, seront données au cours des chapitres 2 à 4 pour accompagner les notions qui y sont introduites. Un état de l'art rapide des algorithmes de somme de produits en précision finie sera proposé au début du chapitre 5. Par ailleurs, [101] offre un état de l'art très détaillé sur les applications de preuve formelle automatique ou interactive aux systèmes cyber-physiques, une famille qui inclut les filtres numériques mais est beaucoup plus vaste.

1.6 Plan et organisation de la formalisation Coq

Ma thèse consiste majoritairement en une formalisation en Coq des filtres numériques LTI, des algorithmes permettant de les calculer, et de théorèmes participant à l'analyse de leur comportement en précision finie. Le plan du manuscrit,

présenté en section 1.6.2, correspond aux différents éléments de cette formalisation. Pour mieux situer ces éléments, je détaille d'abord en section 1.6.1 les objectifs de l'analyse d'erreurs à laquelle ils contribuent. Enfin, la section 1.6.3 donne le lien vers les fichiers Coq et décrit leur organisation.

1.6.1 Objectifs de l'analyse des erreurs d'arrondi

Comme expliqué en section 1.1, mon objectif est d'utiliser les méthodes formelles, en particulier l'assistant de preuve Coq, pour certifier que les implémentations de filtres numériques ont le comportement attendu en dépit des erreurs d'arrondi dues à la précision finie. Mais « comportement attendu » est vague et peut signifier beaucoup de choses différentes. Maintenant que j'ai défini les filtres et la virgule fixe, je peux détailler les deux objectifs principaux vers lesquels tendent les éléments d'analyse d'erreurs que je formalise.

Le premier objectif consiste à borner l'erreur finale entre les sorties d'un filtre modèle et d'un filtre implémenté correspondant : $\boldsymbol{\Delta y}(k) \triangleq \boldsymbol{y}^*(k) - \boldsymbol{y}(k)$. On souhaite obtenir une borne la plus étroite possible, c'est-à-dire la plus proche possible de l'erreur dans le pire cas. Je me sers de cet objectif comme fil directeur car il s'agit d'un but facile à exprimer, et dont on comprend l'intérêt : si $\boldsymbol{\Delta y}(k)$ est petite pour n'importe quel k, cela signifie que l'implémentation reste toujours proche du modèle. Une borne certifiée (et suffisamment étroite) sur cette erreur finale peut être prise en compte lors de la conception d'un système critique utilisant le filtre considéré.

Le second objectif, propre à la virgule fixe, est de vérifier que les formats choisis pour chaque variable dans une implémentation sont appropriés. En particulier, on veut assurer qu'il n'y a pas d'*overflow* imprévu. On verra en section 5.4 qu'il est parfois normal d'avoir des *overflows* modulaires dans certains calculs intermédiaires. Mais souvent, les formats des variables sont choisis expressément pour pouvoir représenter sans *overflow* les valeurs à calculer, et la correction du programme entier repose sur cette absence d'*overflow*. Notons qu'il n'est pas nécessaire de vérifier formellement tout le procédé par lequel ces formats sont choisis. Il suffit de certifier que les formats issus de ce procédé (qu'on peut voir comme une boîte noire ou un oracle) remplissent bien les propriétés que leur construction était censée garantir, par exemple l'absence d'*overflow* évoquée précédemment. Je ne parle pas beaucoup de cet objectif dans ce manuscrit. Mais comme j'expliquerai en section section 4.3.4, les techniques utilisées pour borner l'erreur finale $\boldsymbol{\Delta y}$ permettent aussi d'obtenir des bornes sur les variables intermédiaires du programme (car on peut les voir comme la sortie d'un autre filtre bien choisi). On peut alors utiliser de telles bornes pour garantir l'absence d'*overflow*.

1.6.2 Plan

L'organisation de ce manuscrit est liée aux étapes principales de l'analyse d'erreur que je souhaite formaliser en Coq, visibles sur la figure 1.12 et détaillées ci-dessous. On a vu en section 1.4.5 que l'implémentation d'un filtre modèle passe souvent par le choix d'un algorithme appelé réalisation, puis le choix de formats en précision finie et d'arrondis pour les différentes variables et opérations. On s'intéresse ici à l'objectif fil directeur décrit ci-dessus : borner l'erreur finale $\boldsymbol{\Delta y}$.

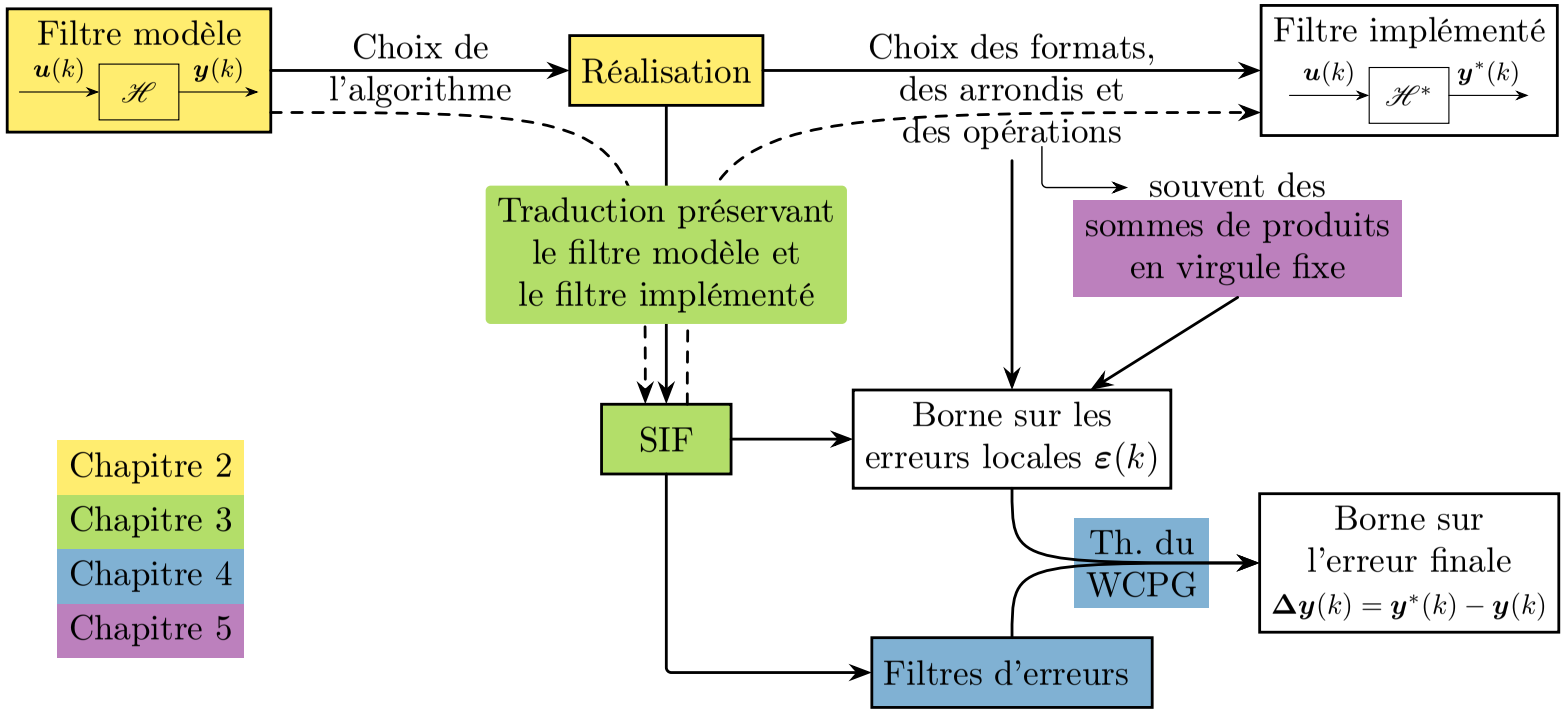


FIGURE 1.12 – Étapes principales de l'analyse d'erreurs à formaliser

Le chapitre 2 établit les bases de ma formalisation : les définitions et propriétés les plus classiques des signaux et des filtres numériques LTI. Il présente aussi des familles de réalisations de filtres usuelles. La précision finie n'intervient pas encore : les filtres de ce chapitre sont des filtres modèles.

J'ai mentionné précédemment que de nombreuses familles de réalisations sont utilisées en pratique. Mais on ne souhaite pas effectuer une nouvelle analyse d'erreurs pour chacune d'entre elles. C'est pourquoi le chapitre 3 s'intéresse à une famille de réalisations appelée la *SIF* (*Specialized Implicit Form*). C'est une représentation matricielle *universelle*, au sens où n'importe quelle réalisation de filtre LTI peut être traduite vers une SIF en préservant la succession de calculs à effectuer. Cela signifie notamment que la réalisation initiale et la SIF obtenue décrivent le même filtre modèle, et aussi le même filtre implémenté pour un même choix de formats et d'arrondis. Autrement dit, les flèches en pointillé de la figure 1.12 complètent des diagrammes commutatifs. L'intérêt est qu'il suffit désormais d'effectuer une analyse d'erreurs sur la SIF, puisque les résultats de cette analyse sont transférables à n'importe quelle réalisation (la traduction préserve la succession de calculs, donc l'intégralité du comportement en précision finie). Le chapitre 3 formalise la SIF et les traductions vers celle-ci depuis les familles de réalisations présentées dans le chapitre 2. Il présente et formalise également le *State-Space*, une famille de réalisations couramment utilisée par la communauté de l'automatique. Ici, le *State-Space* est surtout introduit dans un but didactique car la SIF en est une extension, construite sur les mêmes principes mais beaucoup plus complexe. Enfin, je formalise des transformations d'une SIF à une autre conservant le filtre modèle mais modifiant volontairement le filtre implémenté, dans le but de chercher un algorithme ayant de meilleures propriétés (temps de calcul, énergie, erreur d'arrondie finale, etc.).

Le chapitre 4 présente deux théorèmes au centre de l'analyse d'erreurs de la SIF : le *théorème des filtres d'erreurs* et le *théorème du* Worst-Case Peak Gain (WCPG), tous deux prouvés en Coq. Le premier permet de caractériser la propagation et l'accumulation potentielle des erreurs d'arrondi au cours des nombreuses itérations d'un filtre. Il s'applique à un filtre donné sous forme de *State-Space* ou de SIF. Je présente d'abord le cas plus simple du *State-Space*, puis celui de la SIF (particulièrement pertinent grâce à l'universalité expliquée dans le chapitre précédent). Le principe est que l'erreur finale $\boldsymbol{\Delta y}$, affectée par la propagation des erreurs, est exprimée en fonction d'erreurs locales c'est-à-dire des erreurs individuelles sur une seule opération (éventuellement complexe : ici une somme de produits) ou un seul arrondi, qui ne dépendent donc pas d'autres erreurs. Ces erreurs locales sont rassemblées dans un signal $\boldsymbol{\varepsilon}$, tel que le $\boldsymbol{\varepsilon}(k)$ est composé des erreurs respectives de chaque opération individuelle de somme de produits durant l'itération k. Ensuite, le théorème du *Worst-Case Peak Gain* établit une borne sur la sortie d'un filtre en fonction d'une borne sur l'entrée fournie et du *Worst-Case Peak Gain* du filtre, une quantité qui dépend uniquement de ses coefficients. Comme le nom l'indique, cela correspond au gain d'amplitude maximal pour toutes les entrées possibles. Je prouve en outre que le *Worst-Case Peak Gain* d'un filtre est optimal (c'est la plus petite valeur qui satisfait le théorème), et même atteignable (il existe une entrée dont le gain en amplitude est égal à cette valeur). Combiner les théorèmes des filtres d'erreurs et du *Worst-Case Peak Gain* permet d'obtenir une borne sur le signal d'erreur

finale $\boldsymbol{\Delta y}$, à condition de connaître une borne sur le signal d'erreurs locales $\boldsymbol{\varepsilon}$. Il reste donc à borner les erreurs locales $\boldsymbol{\varepsilon}_i(k)$ dans les sommes de produits utilisées pour calculer le filtre. Ces erreurs locales dépendent de l'arithmétique et des formats en précision finie considérés, ainsi que de l'algorithme de somme de produits utilisé, qui fait l'objet du chapitre suivant.

Le chapitre 5 s'intéresse aux algorithmes de somme de produits. Ce chapitre se place dans le cadre de l'arithmétique en virgule fixe, majoritairement utilisée pour implémenter des filtres dans des systèmes embarqués. Dans un premier temps, j'ignore le problème des *overflows* en considérant des formats théoriques de virgule fixe n'imposant aucune borne sur la mantisse, dans lesquels il n'y a jamais d'*overflow*. Cette approche provient de la bibliothèque Flocq [20], à laquelle j'ai ajouté une légère surcouche pour traiter plus simplement le cas de la virgule fixe en base 2. Dans ce contexte, je formalise trois algorithmes de somme de produits et je prouve une borne sur l'erreur sur la sortie de chacun (c'est la borne dont on avait besoin au chapitre précédent). Les deux premiers algorithmes sont issus de la littérature : l'*accumulateur de Kulisch* et un algorithme usuel en traitement du signal, basé sur l'utilisation de bits de garde. Le troisième algorithme est original. Il est correctement arrondi pour un arrondi au plus proche suivant n'importe quelle règle de *tie-breaking* désirée. Il est plus rapide que l'accumulateur de Kulisch et plus précis que l'algorithme à bits de garde (au sens où la borne sur l'erreur en sortie est plus étroite). Enfin, je m'intéresse à la prise en compte des *overflows* : je formalise l'*overflow* modulaire en virgule fixe en complément à deux. Je prouve alors que la borne sur l'erreur de l'algorithme à bits de garde reste valable même si des *overflows* modulaires sont possibles.

Enfin, le chapitre 6 conclut le manuscrit en résumant les contributions de cette thèse. Il propose aussi des commentaires, détails techniques et retours d'expérience sur le développement en Coq de ma formalisation. Il présente enfin des perspectives de ce travail.

Notons que les chapitres 3 et 4 dépendent chacun des chapitres précédents, puisque tous deux utilisent les bases de ma formalisation des filtres établies par le chapitre 2, et que le chapitre 4 fait intervenir le *State-Space* et la SIF du chapitre 3. Le chapitre 5 est quant à lui indépendant des chapitres 2 à 4, bien que motivé par le chapitre 4.

1.6.3 Organisation et lien vers la formalisation en Coq

La formalisation en Coq est disponible à l'adresse :

```
https://gitlab.com/dianegalloiswong/coq-digitalfilters
```

La figure 1.13 illustre les dépendances entre les fichiers qui composent cette formalisation. Comme mentionné précédemment, on remarque que les chapitres 2 à 4 s'enchaînent tandis que le chapitre 5 est indépendant. L'application des théorèmes prouvés dans le chapitre 4 en utilisant un des algorithmes de somme de produits formalisés dans le chapitre 5 n'a pas été effectuée par manque de temps.

Ceci est donc laissé en perspective en section 6.3 et produirait des fichiers dépendant à la fois de ceux des chapitres 4 et 5.

J'utilise deux fichiers externes, c'est-à-dire récupérés d'autres formalisations (avec quelques modifications listées au début de ces fichiers) :

— `lia_tactics` [6] définit des tactiques pour résoudre automatiquement des buts d'arithmétiques linéaires impliquant les opérations de MathComp sur `nat` ou `int`. Ces tactiques effectuent un pré-traitement des buts et hypothèses afin de les convertir vers les opérations de la bibliothèque standard sur `nat` ou `Z`, puis appliquent les tactiques standard `lia` ou `omega`.

— `Rstruct` de CoqApprox [7] définit les structures canoniques de MathComp sur le type `R` des réels : il construit toute la chaîne de structures jusqu'à montrer que `R` forme un corps. En particulier, j'utilise dans ma formalisation que `R` est un anneau commutatif unitaire.

Les fichiers principaux de la formalisation, ainsi que quelques uns des fichiers auxiliaires, sont présentés tout au long de ce manuscrit avec la correspondance suivante :

Fichier(s)	**Section(s)**
`signal`	Section 2.1
`Epsilon_instances`, `ChoiceType_from_Epsilon`	Section 2.1.2
`filter`	Section 2.2
`TF_coefficients`	Section 2.3.1
`State_Space`	Section 3.1
`SIF`	Sections 3.2 à 3.4
`error_filters_StSp`	Section 4.1
`error_filters_SIF`	Section 4.2
`wcpg`	Section 4.3
`FIX`	Section 5.1.3
`SOP_model`	Section 5.2.1
`SOP_Kulisch`	Section 5.2.2
`summation_decreasinglsb`	Section 5.2.3
`SOP_guard`	Sections 5.2.4 et 5.4.3
`summation_nearest`	Sections 5.3.1 et 5.3.2
`odd_rounding`, `summation_odd`	Section 5.3.3
`overf`, `FIXoverf`	Section 5.4.2

6. `https://github.com/amahboubi/lia4mathcomp`
7. `http://tamadi.gforge.inria.fr/CoqApprox`

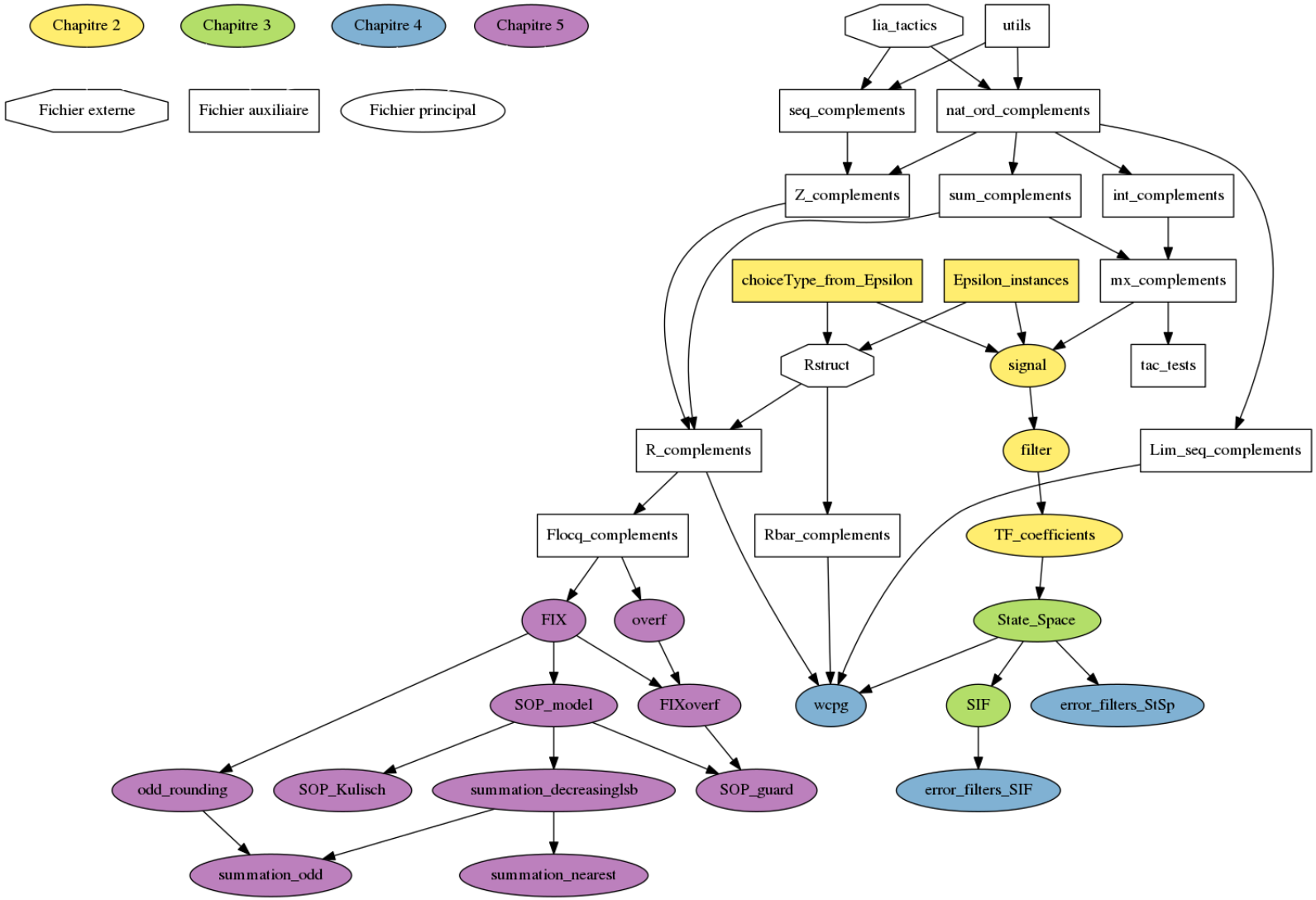


FIGURE 1.13 – Graphe de dépendances des fichiers de la formalisation en Coq

1.7 Notations et conventions

Cette section liste les notations et conventions utilisées dans ce manuscrit. Afin de les rendre facilement consultables, cela inclut des notations déjà introduites précédemment, ou représentant des notions qui ne seront expliquées que dans des chapitres futurs. J'indique alors les sections à consulter pour plus d'informations.

Convention scalaire / vecteur / matrice. Un nombre scalaire est noté en minuscule simple (par exemple x), un vecteur en minuscule en gras (par exemple $\boldsymbol{x}$) et une matrice en majuscule en gras (par exemple $\boldsymbol{X}$). Cette convention s'étend à un signal scalaire x ou vectoriel $\boldsymbol{x}$ (section 1.4.1).

Ensembles mathématiques usuels.

Ensemble	**Notation**	**Type en Coq** (section 1.2.3)
Nombres réels	$\mathbb{R}$	`R` (bibliothèque standard)
Entiers naturels	$\mathbb{N}$	`nat` (b. standard + opérations MathComp)
Entiers relatifs	$\mathbb{Z}$	`int` (MathComp) dans les chapitres 2 à 4 `Z` (bibliothèque standard) dans le chapitre 5

Notations mathématiques usuelles.

Notation	**Signification**
$\triangleq$	égal par définition
$\lvert x \rvert$	valeur absolue
$\lfloor x \rfloor$	partie entière inférieure
$\lceil x \rceil$	partie entière supérieure
$x \equiv y\ [p]$	congruence modulo p c'est-à-dire $\exists k \in \mathbb{Z}, x = y + kp$ (notons que x, y, p peuvent être des réels)
$[a,b]$ resp. $]a,b[$	intervalle fermé respectivement ouvert
$[a,b[$ ou $]a,b]$	intervalle fermé d'un côté et ouvert de l'autre
$\boldsymbol{A}^T$	transposée de la matrice $\boldsymbol{A}$
$\boldsymbol{A}_{ij}$ ou $\boldsymbol{A}_{i,j}$	coefficient en position (i, j) de la matrice $\boldsymbol{A}$
$\boldsymbol{x}_i$	composante d'indice i du vecteur $\boldsymbol{x}$ (ou bien, si $\boldsymbol{x}$ est un signal vectoriel, le signal scalaire tel que $\boldsymbol{x}_i(k) = \boldsymbol{x}(k)_i$)

Quelques notations souvent utilisées dans Coq. (section 1.2.5)

Notation	Signification
`RT`	un anneau (souvent commutatif et/ou unitaire)
`VT`	un espace vectoriel sur `RT` (techniquement un module à gauche de dimension finie sur cet anneau)
`'M[T]_(h, w)`	type des matrices de taille `h` $\times$ `w` à coefficients dans `T`
`'cV[T]_n`	type des vecteurs colonne de hauteur `n` à coefficients dans `T`
`*mx`	produit matriciel
`s ' i`	composant d'indice `(i:nat)` de la liste `(s:seq T)` (ch. 5) où le type `T` doit avoir un zéro, renvoyé par défaut lorsque `i` $\geq$ `size s` (par exemple `T : ringType`)

Arithmétique en précision finie générique. (section 1.3.3)

Notation	Signification
$\mathbb{F}$	un format c'est-à-dire un sous-ensemble de $\mathbb{R}$
$\bigcirc$	une fonction d'arrondi $\mathbb{R} \to \mathbb{F}$
$\oplus$ ou $\ominus$ ou $\otimes$	opération $+$ ou $-$ ou $\times$ correctement arrondie selon $\bigcirc$
$\bigcirc(\boldsymbol{A})$	arrondi coefficient par coefficient d'une matrice $\boldsymbol{A}$ c'est-à-dire $\bigcirc(\boldsymbol{A})_{ij} \triangleq \bigcirc(\boldsymbol{A}_{ij})$
$\odot$	un produit scalaire ou matriciel en précision finie

Arithmétique en virgule fixe. (sections 5.1.1 et 5.4.1)

Notation	Signification
$\mathbb{F}_\ell$	format de *lsb* ℓ sans *overflow* : $\mathbb{F}_\ell = \{m \times 2^\ell \mid m \in \mathbb{Z}\}$
$\bigtriangledown_\ell$ ou $\bigtriangleup_\ell$	arrondi vers le bas (resp. vers le haut) vers $\mathbb{F}_\ell$
$\bigcirc_\ell^\tau$	arrondi au plus proche vers $\mathbb{F}_\ell$ pour la règle de *tie-breaking* donnée τ
$\bigcirc_\ell$	un arrondi donné vers $\mathbb{F}_\ell$ (souvent un arrondi au plus proche)
$\overset{+}{\bigtriangledown}_\ell$ ou $\overset{\times}{\bigtriangledown}_\ell$	addition (resp. multiplication) correctement arrondie pour l'arrondi vers le bas (par exemple $x \overset{+}{\bigtriangledown}_\ell y \triangleq \bigtriangledown_\ell(x+y)$)
$\mathbb{F}_{\ell,w}$	format de lsb ℓ et de largeur w : $\mathbb{F}_{\ell,w} \triangleq \{m \times 2^\ell \mid m \in \mathbb{Z} \wedge -2^{w-1} \leq m \leq 2^{w-1}-1\}$
(similairement $\bigtriangledown_{\ell,w}$ $\bigcirc_{\ell,w}$ $\overset{+}{\bigtriangledown}_{\ell,w}$ etc. pour les arrondis vers $\mathbb{F}_{\ell,w}$)	

Conventions : signal et filtre. Sauf indication contraire, *signal* signifiera *signal numérique à temps discret* (section 1.4.1) *causal* (section 2.1.1). Et *filtre* signifiera *filtre numérique LTI* (section 1.4.2). En l'absence de mention explicite, il s'agira d'un filtre modèle plutôt que d'un filtre implémenté (section 1.4.4). Les réalisations (section 1.4.5) seront aussi implicitement des réalisations de filtre LTI.

Filtres numériques. (section 1.4)

Notation	Signification
$\mathscr{H}$	un filtre modèle (fonction mathématique en précision infinie)
u	un signal d'entrée scalaire fourni à un filtre SISO
y	le signal de sortie scalaire obtenu pour l'entrée u
$\boldsymbol{u}$ et $\boldsymbol{y}$	signaux d'entrée et de sortie vectoriels pour un filtre MIMO
n_u et n_y	tailles des signaux d'entrée et de sortie d'un filtre MIMO
$\mathscr{H}^*$	un filtre implémenté (calculé en précision finie)
y^* ou $\boldsymbol{y}^*$	sortie d'un filtre implémenté pour l'entrée u ou $\boldsymbol{u}$
h	réponse impulsionnelle d'un filtre SISO (section 2.2.3)
$\boldsymbol{h}$	réponse impulsionnelle d'un filtre MIMO (exception à la convention scalaire/vecteur/matrice : voir la section 2.2.3)
$\boldsymbol{x}$	signal d'état du filtre modèle correspondant à un *State-Space* (section 3.1) ou une SIF pour l'entrée $\boldsymbol{u}$
$\boldsymbol{t}$	signal auxiliaire du filtre modèle correspondant à une SIF (section 3.2) pour l'entrée $\boldsymbol{u}$
n_x et n_t	tailles de $\boldsymbol{x}$ et $\boldsymbol{t}$
$\boldsymbol{x}^*$ et $\boldsymbol{t}^*$	signaux d'état et auxiliaire du filtre implémenté correspondant à un *State-Space* ou une SIF
$\langle\!\langle \mathscr{H} \rangle\!\rangle$	*Worst-Case Peak Gain* de $\mathscr{H}$ (section 4.3)

Chapitre 2

Formalisation des filtres numériques linéaires

Les notions principales de traitement du signal ont été présentées dans la section 1.4. Ce chapitre reprend ces notions en donnant plus de détails : des définitions plus précises, les propriétés principales, et surtout leur formalisation en Coq. La section 2.1 est consacrée aux signaux, et la section 2.2 aux filtres. La section 2.3 présente quelques unes des familles les plus usuelles de réalisations de filtre LTI. Les définitions et propriétés de ce chapitre sont toutes classiques en traitement du signal [96].

Ce chapitre se place intégralement dans le cadre de la précision infinie. Les filtres étudiés ici sont des filtres modèles. Le chapitre 3 donnera un moyen d'exprimer un filtre implémenté dans la formalisation. Le comportement des filtres en précision finie sera étudié dans le chapitre 4.

2.1 Signal

Les signaux étudiés par le traitement du signal ont été présentés en section 1.4.1. La section 2.1.1 détaille la définition du type `signal` en Coq, qui repose sur plusieurs choix techniques. La section 2.1.2 explique le lien entre les axiomes utilisés dans la formalisation et l'égalité sur le type `signal`. La section 2.1.3 introduit les opérations de base sur les signaux, et les structures algébriques formées grâce à ces opérations. La section 2.1.4 définit l'impulsion de Dirac et le produit de convolution, qui donnent aux signaux une structure d'anneau commutatif. La section 2.1.5 explique comment la récurrence de Peano habituelle et la récurrence forte sont adaptées pour les entiers relatifs puis utilisées pour construire des signaux. Enfin, la section 2.1.6 rentre dans les détails d'un choix technique sur les indices des signaux mentionné en section 2.1.1.

2.1.1 Définition Coq

Comme expliqué en section 1.4, on s'intéresse aux signaux numériques discrets. Un signal est donc une fonction d'un ensemble d'indices entiers, qui représentent le temps, vers un ensemble de valeurs réelles ou vectorielles. Pour

formaliser cette définition en Coq, il faut choisir quels types représentent ces ensembles.

Type des valeurs. On factorise les deux ensembles usuels pour les valeurs, $\mathbb{R}$ et $\mathbb{R}^n$ pour une taille n donnée, en un espace vectoriel [1] `VT` de dimension finie sur un anneau commutatif `RT`. On utilise pour cela les types respectifs `vectType` et `comRingType` de la bibliothèque MathComp [86], introduits en section 1.2.5. Les propriétés qu'on prouve sur les signaux sont donc très générales. En pratique, l'anneau `RT` sera plus tard instancié à $\mathbb{R}$, mais pourrait par exemple être $\mathbb{C}$.

Type des indices. La difficulté principale est le choix du type des indices. On a vu en section 1.2.3 qu'on a plusieurs types d'entiers différents en Coq, en particulier `nat` pour les entiers naturels et `Z` (bibliothèque standard) ou `int` (MathComp) pour les entiers relatifs. Des ordinaux `'I_n` de MathComp seraient aussi envisageables pour un ensemble d'indices de la forme $\{0,\dots,n-1\}$. *J'ai choisi d'utiliser des indices dans* $\mathbb{Z}$ (type `int` de MathComp), mais de *ne considérer que les signaux qui sont causals* :

Définition 1 (Signal causal). *Un signal x est* causal *si :*

$$\forall k < 0,\ x(k) = 0. \tag{2.1}$$

C'est une hypothèse usuelle en traitement du signal : les applications pratiques ont un début donc on peut assurer cette hypothèse en choisissant bien l'origine temporelle $k = 0$. Cette approche est mathématiquement équivalente à indexer les signaux sur $\mathbb{N}$, mais cela change le comportement des énoncés et des preuves. Les avantages et inconvénients de ce choix, ainsi que d'autres options envisagées, seront discutés dans la section 2.1.6.

Définition d'un signal. Finalement, un signal x est une fonction `int → VT` accompagnée d'une preuve qu'elle est causale, c'est-à-dire nulle pour $k < 0$. Pour grouper la fonction et la preuve, on utilise un type enregistrement (`Record`).

```
Context {RT : comRingType} {VT : vectType RT}.
Definition causal (x : int → VT) :=
  forall k : int, k < 0 → x k = 0.
Record signal :=
  { signal_fun :> int → VT ; signal_causal : causal signal_fun }.
```

La syntaxe `:>` ajoute une coercion (section 1.2.4) du type enregistrement vers le champ de type `int → VT`. Ainsi, un élément `x : signal` peut être utilisé directement comme une fonction `int → VT`, par exemple en écrivant `x k` pour un `k` donné dans `int`.

Exemple de construction d'un signal. Pour définir un signal particulier, il suffit de prouver qu'une fonction `int → VT` est causale. En effet, lorsqu'on a défini le type `signal` comme un `Record`, Coq a automatiquement créé `Build_signal`

1. Comme expliqué en section 1.2.5, j'appelle `VT` un espace vectoriel par association avec le type `vectType` de MathComp bien que ce soit techniquement un module sur l'anneau `RT` (pour pouvoir parler d'espace vectoriel, il faudrait que `RT` soit un corps). De toute façon, en pratique `RT` sera instancié à un corps comme $\mathbb{R}$ donc `VT` sera vraiment un espace vectoriel. Je définis ici `RT` comme seulement un anneau commutatif car c'est suffisant pour ma formalisation.

qui prend en arguments une fonction `int → VT` et une preuve qu'elle est causale, et construit le signal correspondant. Par exemple, définissons l'opposé d'un signal `x` (dont les valeurs sont les opposés respectifs de celles de `x`). Comme `x` est causal, on prouve trivialement que la fonction `(fun k ⇒ - x k)` l'est aussi. Il suffit alors d'utiliser `Build_signal`.

```
Fact opps_causal (x : signal) : causal (fun k ⇒ - x k).
Definition opps (x : signal) := Build_signal (opps_causal x).
```

Notons qu'on a seulement fourni à `Build_signal` le second de ses deux arguments : la preuve de causalité. C'est parce que le premier (la fonction) est un argument implicite, que Coq infère à partir du second.

Signal scalaire, signal vectoriel. Rappelons que les filtres numériques font intervenir deux catégories de signaux : les signaux scalaires à valeurs dans $\mathbb{R}$ (ici `RT`) et les signaux vectoriels à valeur dans $\mathbb{R}^n$. Ces catégories sont facilement définies par instanciation de l'espace vectoriel `VT` dans la définition de `signal`. Pour les signaux scalaires `scsignal`, c'est l'espace vectoriel correspondant à l'anneau `RT` lui-même, donné par `regular_vectType` de MathComp. Pour les signaux vectoriels `vsignal`, il s'agit des vecteurs colonnes sur `RT` d'une taille n donnée, c'est-à-dire les matrices de taille $n \times 1$. On définit aussi les signaux matriciels de taille quelconque $h \times w$, qui ne sont pas directement manipulés par les filtres mais serviront à exprimer la réponse impulsionnelle d'un filtre MIMO en section 2.2.3.

```
Context {RT : comRingType}.
Definition scsignal := @signal RT (regular_vectType RT).
Definition vsignal (n : nat) :=
  @signal RT (matrix_vectType RT n 1).
Definition msignal (h w : nat) :=
  @signal RT (matrix_vectType RT h w).
```

(La notation `@signal` sert seulement à fournir explicitement à `signal` des arguments qui sont normalement implicites. C'est un détail technique de Coq qui n'a pas d'importance dans ce manuscrit.)

Convention : contexte sous-entendu. Afin d'alléger les extraits de Coq, on omettra désormais les déclarations `Context {RT : comRingType}` et `Context {VT : vectType RT}`. On écrira aussi `signal`, `scsignal`, ou `vsignal n` même quand il faudrait techniquement préciser `@signal RT VT`, `@scsignal RT` ou `@vsignal RT n`. Bien sûr, les déclarations seront à nouveau explicites si elles diffèrent de celles données ci-dessus, par exemple `{RT : comUnitRingType}` quand l'anneau aura besoin d'être unitaire.

2.1.2 Égalité de deux signaux et axiomes utilisés

Cette section explique le lien entre les axiomes que j'utilise dans ma formalisation et le type `signal` que je viens de définir.

Rappel de la section 1.2.3 : axiomes de ma formalisation. Les axiomes que j'utilise sont :

— les axiomes de la bibliothèque standard qui définissent les réels [87].

— `FunctionalExtensionality` : deux fonctions identiques point à point sont égales.
— `ProofIrrelevance` : deux preuves d'une même propriété sont égales.
— `R_epsilon_statement` et `fnv_epsilon_statement` : des instanciations de l'axiome *epsilon* de Hilbert aux types `R` (nombres réels) et `nat → VT`, dont l'intérêt sera expliqué à la fin de cette section.

Extensionnalité des signaux. Les axiomes `FunctionalExtensionality` et `ProofIrrelevance` sont utilisés dans de nombreux développements mathématiques, et sont généralement acceptés comme sûrs par la communauté [2]. Je les utilise pour prouver que deux signaux qui prennent la même valeur sur tous les entiers relatifs sont égaux. Ceci est exprimé par le lemme `signalP` (qui est une équivalence, la réciproque étant évidente). La notation `x1 =1 x2` signifie l'égalité point à point : `forall k : int, x1 k = x2 k`.

```
Lemma signalP (x1 x2 : signal) : x1 =1 x2 ⟷ x1 = x2.
```

Comme on a défini les signaux comme des fonctions causales, il suffit d'ailleurs que deux signaux soient égaux sur tous les entiers positifs.

```
Lemma signalP_ge0 (x1 x2 : signal) :
  (forall k : int, k ≥ 0 → x1 k = x2 k) ⟷ x1 = x2.
```

Décidabilité de l'égalité sur les signaux. Pour équiper le type `signal` des structures algébriques de MathComp dans la prochaine section, on doit d'abord montrer que c'est un `eqType`, c'est-à-dire que l'égalité de deux signaux est décidable. Mais on ne peut pas le prouver sans aucun axiome, car les signaux sont des fonctions des entiers vers `RT` qui est juste un `comRingType` sans hypothèse supplémentaire. Cependant, la bibliothèque Coquelicot [18] prouve `LPO` (*Limited Principled of Omniscience*, énoncé ci-dessous) à partir des axiomes définissant les réels de la bibliothèque standard. Or on utilisera de toute façon ces réels dans la suite de la formalisation (dans les chapitres 4 et 5, entre autres pour borner des erreurs d'arrondi en précision finie). On ne perd donc rien à utiliser `LPO`, qui rend la décidabilité de l'égalité sur les signaux facile à prouver.

```
Check LPO.   (* LPO : forall P : nat → Prop,
                      (forall n : nat, P n \/ ~ P n) →
                      {n : nat | P n} + {forall n : nat, ~ P n} *)
Lemma signal_comparable : comparable signal.
                      (* forall x y : signal, {x = y} + {x ≠ y} *)
Definition signal_eqMixin := comparableMixin signal_comparable.
Canonical signal_eqType := EqType signal signal_eqMixin.
```

La notation `{ _ } + { _ }` représente une disjonction décidable, et la notation `{ x : T | e }` l'existence constructive d'un élément `x` de type `T` rendant l'expression `e` vraie.

Structure de `choiceType`. Le second prérequis pour utiliser les structures algébriques de MathComp dans la prochaine section est prouver que les signaux forment un `choiceType`, c'est-à-dire un type avec un opérateur de choix (similaire à l'axiome du choix). Cela nécessite encore une fois des axiomes. À

2. https://github.com/coq/coq/wiki/The-Logic-of-Coq

partir de l'axiome *epsilon* de Hilbert (`epsilon_statement` du fichier `Epsilon` de la bibliothèque standard), on peut prouver que tout `eqType` non vide est un `choiceType`. Comme j'ai seulement besoin de cet axiome pour montrer que les réels et les signaux sont des `choiceType`, j'ai préféré admettre spécifiquement les instanciations d'*epsilon* à ces deux types. Elles sont déclarées comme axiomes dans mon fichier `Epsilon_instances` (qui contient aussi en commentaire leur preuve si on admet `Epsilon` de la bibliothèque standard). Pour les signaux, comme le type `signal` n'est pas encore défini, on utilise `nat → VT` avec lequel il est en bijection.

```
Axiom R_epsilon_statement :
  forall P : R → Prop, {x : R | (exists x0 : R, P x0) → P x}.
Axiom fnv_epsilon_statement :
  forall (RT : ringType) (VT : vectType RT),
  forall P : (nat → VT) → Prop,
  {x : nat → VT | (exists x0, P x0) → P x}.
```

Ensuite, dans le fichier `signal`, on prouve que cette propriété sur `nat → VT` se transmet au type `signal` en explicitant la bijection. On obtient alors la structure de `choiceType` grâce à `EpsilonChoiceMixin` de mon fichier `choiceType_from_Epsilon`.

```
Lemma signal_epsilon_statement :
  forall P : signal → Prop,
  {x : signal | (exists x0, P x0) → P x}.
Definition signal_choiceMixin :=
  EpsilonChoiceMixin signal_epsilon_statement.
Canonical signal_choiceType :=
  ChoiceType signal signal_choiceMixin.
```

2.1.3 Opérations basiques, $\mathbb{Z}$-module, module sur l'anneau `RT`

Cette section présente les trois opérations basiques sur les signaux et les structures algébriques sur les signaux qu'elles permettent d'obtenir.

Opérations basiques. Le traitement du signal s'appuie sur trois opérations essentielles sur les signaux :

— L'addition (terme à terme) : $y = x_1 + x_2$ signifie $\forall k, y(k) = x_1(k) + x_2(k)$. Pour la définition Coq, il suffit de prouver trivialement que l'addition des signaux causals x_1 et x_2 est encore causale : $\forall k < 0, y(k) = 0$.

```
Fact adds_causal (x1 x2 : signal) :
  causal (fun k ⇒ x1 k + x2 k).
Definition adds (x1 x2 : signal) :=
  Build_signal (adds_causal x1 x2).
```

— La multiplication par un scalaire $c \in$ `RT` : $y = c\,x$ signifie $\forall k, y(k) = c\,x(k)$. En Coq, la multiplication du scalaire `c` par `v` $\in$ `VT` est notée `c *: v`.

```
Fact scales_causal (c : RT) (x : signal) :
  causal (fun k ⇒ c *: x k).
Definition scales (c : RT) (x : signal) :=
  Build_signal (scales_causal c x).
```

— Le *retard* (ou *décalage temporel*) : « y est le signal x retardé de K unités de temps » signifie $\forall k, y(k) = x(k - K)$.
Comme x est causal, on a toujours : $\forall k < K, y(k) = 0$.
Notons qu'avec la définition des signaux que j'utilise, K doit être positif ou nul : sinon, y risque de ne pas être causal. On veut de toute façon supposer $K \geq 0$ en pratique : on ne peut pas décaler de $K < 0$ c'est-à-dire avancer un signal dans le temps, puisqu'on ne peut pas voir dans le futur.

```
Fact delay_causal (K : int) (x : signal) :
  causal (fun k ⇒ if K ≥ 0 then x (k-K) else 0).
Definition delay (K : int) (x : signal) :=
  Build_signal (delay_causal K x).
```

Dans la définition Coq, j'ai choisi de renvoyer arbitrairement le signal identiquement nul lorsque $K < 0$, car c'est plus facile de travailler avec des fonctions totales. Cependant, le retard reste généralement utilisé sous l'hypothèse que $K \geq 0$, comme dans le lemme d'élimination `delayE`. (On rappelle que la notation `=1` représente l'égalité des images de deux fonctions pour chaque entrée.)

```
Lemma delayE (K : int) (K_ge0 : K ≥ 0) (x : signal) :
  delay K x =1 (fun k ⇒ x (k - K)).
```

Structures algébriques. Ces opérations font apparaître des structures algébriques sur l'ensemble des signaux. Pour les formaliser, on s'appuie sur la bibliothèque MathComp comme expliqué en section 1.2.5. Pour accompagner l'addition, on définit très facilement le signal identiquement nul `signal0`, et l'opposé d'un signal `opps` apparaît comme exemple en section 2.1.1. Les signaux forment alors un groupe abélien c'est-à-dire un $\mathbb{Z}$-module (`zmodType`). Les preuves sont triviales grâce aux lemmes correspondants sur `VT`.

```
Definition signal0 := Build_signal (fun _ _ ⇒ erefl 0).

(* preuves triviales de addsA (associativité), addsC
  (commutativité), add0s (signal0 + x = x) et addNs (-x+x=0) *)

Definition signal_zmodMixin := ZmodMixin addsA addsC add0s addNs.
Canonical signal_zmodType := ZmodType signal signal_zmodMixin.
```

Avec la multiplication par une constante, on obtient ensuite un `RT`-module à gauche (`lmodType`).

```
(* preuves triviales de scalesA (a *: (b *: x) = (a * b) *: x),
   scale1s (1 *: x = x), scalesDr et scalesDl (distributivités) *)

Definition signal_lmodMixin :=
  LmodMixin scalesA scale1s scalesDr scalesDl.
Canonical signal_lmodType := LmodType R signal signal_lmodMixin.
```

2.1.4 Impulsion de Dirac, produit de convolution et anneau commutatif

On définit ici l'impulsion de Dirac et le produit de convolution de deux signaux. On obtient alors une structure d'anneau commutatif sur les signaux.

Définition 2 (Impulsion de Dirac). *L'*impulsion de Dirac δ *(ou juste le* dirac *par abus de langage) est le signal scalaire défini par :*

$$\delta(k) = \begin{cases} 1 & si\ k = 0 \\ 0 & sinon \end{cases} \tag{2.2}$$

```
Fact dirac_causal :
  causal (fun k ⇒ if k is 0 then (1 : regular_vectType R) else 0).
Definition dirac : scsignal := Build_signal dirac_causal.
```

Le dirac est omniprésent en traitement du signal. C'est notamment l'élément neutre du produit de convolution défini ci-dessous. On verra aussi plus bas qu'il permet de définir une base infinie canonique pour les signaux scalaires. De plus, l'image du dirac à travers un filtre, appelée réponse impulsionnelle, est elle aussi une notion essentielle du traitement du signal (qui sera présentée en section 2.2.3).

Définition 3 (Produit de convolution). *Le* produit de convolution *de deux signaux scalaires x et y, noté $x * y$, est défini par :*

$$\forall k \in \mathbb{Z}, \quad (x * y)(k) = \sum_{i=0}^{k} x(i)\, y(k-i) \tag{2.3}$$

```
Fact convols_causal (x y : scsignal) :
  causal (fun k : int ⇒
          \sum_(0 ≤ i < (k + 1)%int) x i * y (k-(i:int))).
Definition convols (x y : scsignal) : scsignal :=
  Build_signal (convols_causal x y).
```

Remarquons que comme on ne considère que des signaux causals, on aurait aussi bien pu écrire :

$$\forall k \in \mathbb{Z}, \quad (x * y)(k) = \sum_{i \in \mathbb{Z}} x(i)\, y(k-i) \tag{2.4}$$

En effet, $x(i)$ est nul pour $i < 0$ et $y(k-i)$ est nul pour $i > k$.

Anneau commutatif. On prouve facilement que le produit de convolution est associatif, commutatif, distributif sur l'addition, et a un élément neutre : le dirac. Les signaux forment donc un anneau commutatif.

```
Definition signal_ringMixin :=
  ComRingMixin convolsA convolsC convol1s convolsDl nonzero1s.
Canonical signal_ringType := RingType signal signal_ringMixin.
Canonical signal_comRingType := ComRingType signal convolsC.
```

Décomposition d'un signal en somme de diracs retardés. Une autre propriété du dirac est qu'on peut écrire, pour tout signal scalaire x :

$$\forall k \in \mathbb{Z}, \quad x(k) = \sum_{i=0}^{+\infty} x(i)\delta(k-i) \tag{2.5}$$

vu que le seul terme non nul de la somme est lorsque $i = k$. L'intérêt est qu'en notant δ_i le dirac retardé de i unités de temps, on obtient l'égalité suivante (entre des signaux et non plus entre des valeurs scalaires) :

$$x = \sum_{i=0}^{+\infty} x(i)\delta_i \tag{2.6}$$

Les diracs retardés $(\delta_i)_{i\in\mathbb{N}}$ forment donc une base infinie de l'ensemble des signaux scalaires, considérée comme la base canonique. Les signaux scalaires forment alors un espace vectoriel de dimension infinie sur `RT` (toujours par léger abus de langage vu que `RT` est ici défini comme seulement un anneau plutôt qu'un corps). Cependant, on ne peut pas utiliser le type `vectType` de MathComp qui est seulement pour les espaces vectoriels de dimension finie.

En Coq, on évite ici de manipuler des sommes infinies dont il faudrait prouver la convergence. À la place, on montre que si on considère la somme des $x(i)\delta_i$ seulement jusqu'à un certain $K \geq 0$, on obtient un signal qui coïncide avec x sur $\{0, 1, \ldots, K\}$.

```
Lemma signal_sum_dirac (x : scsignal) (K k : int) :
  0 ≤ k ≤ K →
  x k = (\sum_(i < (K + 1)%int) (x i : RT) *: delay i dirac) k.
```

Ce lemme servira à prouver la caractérisation d'un filtre LTI par sa réponse impulsionnelle en section 2.2.3.

2.1.5 Construction récursive d'un signal

En traitement du signal, les signaux sont très souvent construits par récurrence. On a donc besoin de pouvoir prouver des propriétés et construire des fonctions récursivement sur $\mathbb{Z}$. Comme les signaux sont nuls pour $k < 0$, on décide de faire l'initialisation sur tous les $k < 0$ simultanément. Cela ne correspond pas aux récurrences sur $\mathbb{Z}$ présentes dans la bibliothèque standard (qui initialisent pour $k = 0$ puis montrent deux hérédités : de k vers $k + 1$ pour $k \geq 0$, et de k vers $k - 1$ pour $k \leq 0$). On définit donc de nouvelles preuves par induction et constructions récursives dans le fichier `int_complements`.

On utilise deux variantes de récurrence : la récurrence la plus usuelle, dite *de Peano*, dont l'hérédité passe de k à $k + 1$, et la *récurrence forte* où le terme en k peut dépendre de tous les termes antérieurs.

Preuve par récurrence usuelle de Peano sur `int`. On veut prouver une propriété $P(k)$ pour tout $k \in \mathbb{Z}$. Comme expliqué ci-dessus, on traite tous les $k < 0$ pendant l'initialisation. Pour les $k \geq 0$, on utilise la récurrence usuelle sur $\mathbb{N}$. Le cas $k = 0$ est donc lui aussi inclus dans l'initialisation, qui est finalement faite pour tout $k \leq 0$. L'hérédité est de la forme $P(k) \Rightarrow P(k + 1)$, avec l'hypothèse

supplémentaire que $k \geq 0$. Ce lemme est facile à prouver à partir de l'induction sur les entiers naturels fournie par Coq (tactique `elim`).

```
Lemma peanoindz (P : int → Prop) :
  (forall k, k ≤ 0 → P k) →
  (forall k, 0 ≤ k → P k → P (k + 1)) →
  forall k, P k.
```

Preuve par récurrence forte sur `int`. La récurrence forte sur $\mathbb{N}$ sur laquelle on se base est :

$$(\forall n \in \mathbb{N},\ (\forall i < n, P(i)) \Rightarrow P(n)) \implies \forall n \in \mathbb{N}, P(n) \tag{2.7}$$

Cette variante ne nécessite pas d'initialisation puisque $\forall i < 0, P(i)$ est toujours vrai sur $\mathbb{N}$. Pour l'étendre à $\mathbb{Z}$, on se contente d'initialiser pour $k < 0$, puis de garder la même hérédité pour $k \geq 0$. On prouve le lemme correspondant en utilisant `peanoindz` pour montrer que $\forall k \in \mathbb{Z}, (\forall i < k, i < k \Rightarrow P(i))$.

```
Lemma strongindz (P : int → Prop) :
  (forall k, k < 0 → P k) →
  (forall k, k ≥ 0 → (forall i, i < k → P i) → P k) →
  forall k, P k.
```

Constructions récursives sur `int`. On veut construire récursivement une fonction f vers un type T quelconque, pas forcément `Prop` comme précédemment. Pour la récurrence de Peano, on se donne une fonction d'initialisation $f_{\text{BC}} : \mathbb{Z} \to T$ (le nom vient de *Base Case*) et une fonction d'hérédité $f_{\text{IS}} : \mathbb{Z} \to T \to T$ (comme *Induction Step*), et on définit f par :

$$\begin{cases} f(k) = f_{\text{BC}}(k) & \text{si } k \leq 0 \\ f(k+1) = f_{\text{IS}}(k)(f(k)) & \text{si } k \geq 0 \end{cases} \tag{2.8}$$

Pour la récurrence forte, le type de f_{BC} est le même, mais celui de f_{IS} devient $\mathbb{Z} \to (\mathbb{Z} \to T) \to T$, et f est obtenue comme :

$$\begin{cases} f(k) = f_{\text{BC}}(k) & \text{si } k < 0 \\ f(k+1) = f_{\text{IS}}(k)(f') & \text{si } k \geq 0, \text{ où } f' \text{ est la restriction de } f \text{ à }]-\infty, k[\end{cases} \tag{2.9}$$

Comme les fonctions partielles ne sont pas très pratiques à utiliser en Coq, on considère f' comme une fonction totale qui est égale à f sur $]-\infty, k[$, d'où le type de f_{IS}. Bien sûr, f_{IS} ne doit surtout pas utiliser les valeurs de f' en $[k, +\infty[$. Cela est exprimé par la propriété `strongrecz_safe` plus bas.

Voici les types complets pour les deux constructions récursives, avec la définition de la première. Je n'ai pas inclus la définition de `strongrecz` car elle est un peu longue et pas très lisible, mais elle se trouve dans le fichier `int_complements`.

```
Definition peanorecz
           (T : Type) (f_BC : int → T) (f_IS : int → T → T) k :=
  let f_nat :=
    nat_rect (fun _ ⇒ T) (f_BC 0) (fun m ⇒ f_IS (Posz m)) in
  match k with
    | Negz _ ⇒ f_BC k
    | Posz 0 ⇒ f_BC 0
    | Posz n ⇒ f_nat n
  end.
Check peanorecz.
  (* forall T : Type, (int → T) → (int → T → T) → int → T *)
Check strongrecz.
  (* forall T : Type,
       (int → T) → (int → (int → T) → T) → int → T *)
```

Ces définitions ne devraient pas avoir besoin d'être dépliées. À la place, les lemmes suivants servent d'interface. Pour le lemme sur l'hérédité de la récurrence forte, on a besoin de l'hypothèse `strongrecz_safe` signifiant que $f_{\mathrm{IS}}(k)(f')$ ne dépend pas des valeurs de f' sur $[k, +\infty[$, comme expliqué précédemment.

```
Lemma peanoreczBC
        (T : Type) (f_BC : int → T) (f_IS : int → T → T) k :
  k ≤ 0 → peanorecz f_BC f_IS k = f_BC k.

Lemma peanoreczIS
        (T : Type) (f_BC : int → T) (f_IS : int → T → T) k :
  0 ≤ k →
  peanorecz f_BC f_IS (k + 1) = f_IS k (peanorecz f_BC f_IS k).

Lemma strongreczBC (T : Type)
        (f_BC : int → T) (f_IS : int → (int → T) → T) k :
  k < 0 → strongrecz f_BC f_IS k = f_BC k.

Definition strongrecz_safe
             (T : Type) (f_IS : int → (int → T) → T) : Prop :=
  forall (k : int) (f'1 f'2 : int → T),
  0 ≤ k → (forall i : int, i < k → f'1 i = f'2 i) →
  f_IS k f'1 = f_IS k f'2.

Lemma strongreczIS (T : Type)
        (f_BC : int → T) (f_IS : int → (int → T) → T) :
  strongrecz_safe f_IS →
  forall k : int, 0 ≤ k →
  strongrecz f_BC f_IS k = f_IS k (strongrecz f_BC f_IS).
```

Construction récursive d'un signal. Pour construire un signal à partir des fonctions précédentes, il suffit d'initialiser à 0 sur les indices strictement négatifs. Notons que dans le cas de la récurrence de Peano qui demande des valeurs initiales pour tout $k \leq 0$, on pourrait demander en argument une valeur pour $k = 0$, qui n'est pas obligée d'être nulle. Mais dans la suite, à chaque fois que

j'ai besoin de construire un signal x par récurrence de Peano, il se trouve que je veux $x(0) = 0$. J'ai donc décidé de simplifier un peu la définition en initialisant toujours cette valeur aussi à 0.

```
Fact signal_peanorec_causal (f_IS : int → VT → VT) :
  causal (peanorecz (fun _ ⇒ (0:VT)) f_IS).
Definition signal_peanorec (f_IS : int → VT → VT) :=
  Build_signal (signal_peanorec_causal f_IS).
Fact signal_strongrec_causal (f_IS : int → (int → VT) → VT) :
  causal (strongrecz (fun _ ⇒ 0) f_IS).
Definition signal_strongrec
             (f_IS : int → (int → VT) → VT) : signal :=
  Build_signal (signal_strongrec_causal f_IS).
```

2.1.6 Choix technique : type pour les indices des signaux

Cette section discute le choix du type utilisé pour les indices des signaux, mentionné en section 2.1.1.

Définition mathématique d'un signal : fonction causale sur $\mathbb{Z}$. Du point de vue du traitement du signal, même s'il existe des signaux définis seulement sur un intervalle borné d'entiers, considérer des indices dans $\mathbb{Z}$ est le cas le plus général. Mais en pratique, le passé n'est pas infini, donc on peut supposer qu'un signal ne commence qu'à partir d'un certain instant k_0 et est nul avant. L'intérêt est qu'un tel signal peut être facilement défini par récurrence, puisqu'on peut le voir comme une fonction sur $\mathbb{N}$ décalée. Or on étudie un nombre fini de signaux à la fois : quitte à translater le temps, on peut donc s'arranger pour qu'ils soient tous nuls avant $k = 0$, ce qui facilite encore les choses. C'est pour cela que j'ai choisi d'utiliser la définition d'un signal comme une fonction sur $\mathbb{Z}$ qui est causale (c'est-à-dire qui vaut 0 sur les indices strictement négatifs).

Pourquoi pas $\mathbb{N}$? Les signaux causals sur $\mathbb{Z}$ sont en bijection avec les fonctions sur $\mathbb{N}$, donc pourquoi ne pas considérer directement des signaux sur $\mathbb{N}$? L'intérêt de travailler dans $\mathbb{Z}$ est de ne pas avoir de cas particulier pour les premières valeurs. Par exemple, on veut définir un signal x à partir de la relation de récurrence :

$$\forall k, x(k) = x(k-1) - 3x(k-2) + 5 \tag{2.10}$$

Si les indices sont dans $\mathbb{N}$, on doit donner des valeurs initiales pour $x(0)$ et $x(1)$: en effet, la relation de récurrence ne peut pas être utilisée car $x(-2)$ et $x(-1)$ ne sont pas définis. Avec un signal causal sur $\mathbb{Z}$, on sait que ces valeurs sont nulles donc la relation de récurrence suffit.

Type des indices en Coq : options envisagées. On a choisi quelle définition mathématique utiliser pour un signal, mais il faut encore décider comment la représenter en Coq. Voici les trois solutions que j'ai envisagées. Finalement, je n'ai pas l'impression qu'il y en ait une bien meilleure que les autres. Ce choix est donc surtout une préférence personnelle parmi les divers avantages et inconvénients.

— **Indices dans `nat`.** J'ai expliqué pourquoi je préfère considérer $\mathbb{Z}$ plutôt que $\mathbb{N}$ dans la définition mathématique, mais cela n'empêche a priori pas d'utiliser `nat` dans Coq. On doit juste garder à l'esprit que le signal `x : nat → VT` représente le signal mathématique x qui prend ces valeurs pour $k \geq 0$ et est nul pour $k < 0$. Dans l'exemple précédent, on sait que $x(-1)$ et $x(-2)$ valent 0 même si on ne peut pas les exprimer ainsi en Coq. C'est la définition formelle la plus facile à exprimer et comprendre : simplement `signal := nat → VT`. Travailler avec `nat` est un autre avantage : c'est un type très utilisé, qui dispose de nombreux lemmes et tactiques. Cependant, cette solution nécessite de faire attention quand on utilise une valeur pour un indice contenant une soustraction, ce qui est fréquent en traitement du signal. En effet, la soustraction de Coq sur `nat` est saturée : `n - i` vaut `0` lorsque `n < i`, donc on ne peut pas juste écrire `x (n - 2)`. On peut passer par une définition comme :

```
Definition signal_val_sub n i :=
  if i ≤ n then x (n - i) else 0.
```

Cela fonctionne, mais complique un peu les définitions de traitement du signal et nécessite plus de cas particuliers dans les preuves.

— **Indices dans `nat` en imposant `x(0) = 0`.** Une idée naturelle suite au problème de la solution précédente est de décaler les signaux d'une unité de temps : les valeurs pertinentes sont pour `n ≥ 1`, et on impose que `x(0) = 0`. On tire ainsi parti de la soustraction saturée de Coq pour toujours avoir `x(n-i) = 0` lorsque `n < i`. On n'a plus besoin de `signal_val_sub`, et les cas particuliers fonctionnent tout seuls. De plus, on conserve l'avantage de travailler sur le type `nat`. C'est probablement l'approche qui conduit aux preuves les plus simples. L'inconvénient majeur est que cela produit des définitions très étranges pour quelqu'un de familier avec le traitement du signal. Par exemple, le dirac vaut alors `1` en `n=1` et vaut `0` partout ailleurs, en particulier δ`(0) = 0` ! C'est rédhibitoire pour moi, car des énoncés Coq trop différents de la littérature compliqueraient la propagation des méthodes formelles dans la communauté du traitement du signal.

— **Indices dans `int` en imposant `x(k) = 0` pour `k < 0`.** La solution retenue est la plus proche de la définition mathématique. La définition du type `signal` lui-même (section 2.1.1) est un peu complexe, mais les indices sont bien compréhensibles dans les définitions ultérieures. Les cas particuliers sont faciles à traiter, comme pour la solution précédente. L'inconvénient majeur est que le type `int` n'a pas autant de lemmes et tactiques que `nat`. De plus, on doit parfois travailler avec un mélange de `nat` et `int` avec des coercions (expliquées en section 1.2.4). J'ai donc écrit des tactiques pour prouver des propriétés faisant intervenir des entiers naturels vus comme des entiers relatifs à partir d'hypothèses sur seulement les entiers naturels, et inversement. Ces tactiques seront discutées en section 6.2.3.

2.2 Filtre numérique

Cette section présente la formalisation en Coq des filtres LTI et de leurs principales propriétés, ainsi que quelques notions un peu plus approfondies qu'en section 1.4.2. La section 2.2.1 définit les filtres LTI en Coq. La section 2.2.2 présente deux propriétés qui sont généralement exigées des filtres étudiés en pratique : la causalité et la BIBO stabilité. La section 2.2.3 définit la réponse impulsionnelle d'un filtre et montre son lien avec le produit de convolution. La section 2.2.4 présente l'équation aux différences, une manière très usuelle de définir un filtre en particulier. La section 2.2.5 introduit les graphes de flot de données pour représenter facilement des algorithmes définissant des filtres. Enfin, la section 2.2.6 donne un bref aperçu des filtres du point de vue du domaine fréquentiel, qui n'est pas étudié dans cette thèse.

2.2.1 Définition d'un filtre LTI

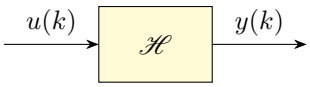


FIGURE 2.1 – Filtre numérique (filtre modèle en précision infinie)

Rappels de la section 1.4.2. Un *filtre numérique* $\mathscr{H}$ est une fonction mathématique qui transforme un signal d'entrée u en un signal de sortie y, comme illustré par la figure 2.1. On écrit alors $y = \mathscr{H}\{u\}$. Cette définition autorise théoriquement la sortie $y(k)$ au temps k à dépendre de toutes les valeurs du signal d'entrée u, pas seulement de l'entrée courante $u(k)$. Cependant, dans les filtres utilisés en pratique, cette sortie $y(k)$ ne peut dépendre que des entrées antérieures $u(l)$ pour $l \leq k$, et pas des entrées futures qui ne sont pas encore connues. On dit alors que le filtre est *causal* : voir la section 2.2.2 pour plus de détails. Un filtre est dit *SISO* (*Single-Input Single-Output*) si les signaux d'entrée et sortie sont scalaires, ou *MIMO* (*Multiple-Input Multiple-Output*) pour des signaux vectoriels.

Définition Coq. La définition Coq est la même que la définition mathématique : une fonction d'un ensemble de signaux vers un autre. L'anneau commutatif `RT` est le même pour les signaux d'entrée et sortie. En revanche, les espaces vectoriels pour les valeurs peuvent être différents : par exemple, pour un filtre MIMO, les signaux d'entrée et sortie sont à valeurs dans $\texttt{RT}^{n_u}$ et $\texttt{RT}^{n_y}$ respectivement. On prend donc en paramètres deux espaces vectoriels `VTin` et `VTout`.

```
Context {RT : comRingType} {VTin VTout : vectType RT}.
Definition filter := @signal RT VTin → @signal RT VTout.
```

Comme pour les signaux scalaires et vectoriels, ce même type `filter` peut être instancié à des filtres explicitement SISO ou MIMO.

```
Definition SISO_filter :=
  @filter RT (regular_vectType RT) (regular_vectType RT).
Definition MIMO_filter (nu ny : nat) :=
  @filter RT (matrix_vectType RT nu 1) (matrix_vectType RT ny 1).
```

Définition 4 (LTI). *Un filtre numérique est* linéaire invariant dans le temps *(*LTI *pour* Linear Time Invariant*) s'il est compatible avec les trois opérations basiques sur les signaux définies en section 2.1.3. Les propriétés à vérifier sont donc (en quantifiant universellement les signaux, la constante* $c \in \mathtt{RT}$*, et l'entier* $K \geq 0$*) :*

- *Compatibilité avec l'addition :* $\mathscr{H}\{u_1 + u_2\} = \mathscr{H}\{u_1\} + \mathscr{H}\{u_2\}$
- *Compatibilité avec la multiplication par une constante :* $\mathscr{H}\{c\,u\} = c\,\mathscr{H}\{u\}$
- *Compatibilité avec le retard, aussi appelée* invariance dans le temps *:*

$$(\forall k, u'(k) = u(k-K)) \implies (\forall k, \mathscr{H}\{u'\}(k) = \mathscr{H}\{u\}(k-K))$$

```
Definition filter_compat_add (H : filter) :=
  forall u1 u2 : signal, H (u1 + u2) = H u1 + H u2.
Definition filter_compat_scale (H : filter) :=
  forall (c : RT) (u : signal), H (c *: u) = c *: H u.
Definition filter_compat_delay (H : filter) :=
  forall (K : int) (u : signal), K ≥ 0 →
  H (delay K u) = delay K (H u).
Definition LTI_filter (H : filter) :=
  filter_compat_add H ∧ filter_compat_scale H
                      ∧ filter_compat_delay H.
```

Dans cette thèse, je m'intéresse seulement aux filtres LTI, qui sont des briques de base très courantes en traitement du signal et en contrôle-commande [96].

2.2.2 Causalité et BIBO stabilité

La causalité et la BIBO stabilité définies ci-dessous sont deux autres propriétés partagées par la plupart des filtres utilisés en pratique.

Définition 5 (Causalité). *Un filtre est* causal *si la sortie courante ne dépend pas des entrées futures :* $y(k)$ *n'a le droit de dépendre que des* $u(m)$ *pour* $m \leq k$*.*

Bien sûr, tous les filtres utilisés en pratique sont causals puisque les entrées futures sont inconnues. La notion de causalité d'un filtre est étroitement liée à la causalité d'un signal, notamment à travers la réponse impulsionnelle qui sera définie en section 2.2.3. Mais même sans passer par la réponse impulsionnelle, comme on a supposé en section 2.1.1 que tous les signaux sont causals, on peut prouver qu'un filtre LTI est toujours causal.

```
Definition causal_filter (H : filter) : Prop :=
  forall (u1 u2 : signal) (k : int),
  (forall m : int, m ≤ k → u1 m = u2 m) → (H u1) k = (H u2) k.
Theorem causal_LTI_filter (H : filter) :
  LTI_filter H → causal_filter H.
```

La définition suivante requiert une notion de norme : l'anneau `RT` est ici instancié au type $\mathbb{R}$ des réels, et les valeurs des signaux sont soit des réels, soit des vecteurs de nombres réels.

Définition 6 (BIBO stabilité). *Un filtre est* BIBO stable (Bounded-Input Bounded-Output) *si pour toute entrée bornée, la sortie est aussi bornée* [70]. *Pour un filtre SISO, cela s'écrit :*

$$\forall u,\ (\exists M_u \in \mathbb{R}, \forall k,\ |u(k)| \leq M_u) \implies (\exists M_y \in \mathbb{R}, \forall k,\ |\mathscr{H}\{u\}(k)| \leq M_y)$$

Pour un filtre MIMO, on peut par exemple utiliser la norme infinie pour les vecteurs :

$$\forall \boldsymbol{u},\ (\exists M_u \in \mathbb{R}, \forall k,\ \|\boldsymbol{u}(k)\|_\infty \leq M_u) \implies (\exists M_y \in \mathbb{R}, \forall k,\ \|\mathscr{H}\{\boldsymbol{u}\}(k)\|_\infty \leq M_y)$$

On peut supposer que les filtres étudiés sont BIBO stables car c'est une propriété qu'on exige souvent des filtres utilisés en pratique. Cependant, on n'aura pas besoin de cette propriété avant le chapitre 4.

La définition Coq se trouve dans le fichier `wcpg` qui fait l'objet de la section 4.3, car c'est seulement là que j'instancie `RT` à `R`. Elle ne couvre que le cas SISO car je n'utilise pas le cas MIMO.

```
Definition bounded_scsignal (x : @scsignal R) :=
  exists M : R, forall k, (Rabs (x k) ≤ M)%R.
Definition BIBO_SISO_filter (H : SISO_filter) :=
  forall u : scsignal, bounded_scsignal u → bounded_scsignal (H u).
```

2.2.3 Réponse impulsionnelle

Une autre notion essentielle en traitement du signal est la réponse impulsionnelle d'un filtre. On s'intéresse d'abord au cas simple d'un filtre SISO, puis au cas un peu plus complexe d'un filtre MIMO.

Définition 7 (Réponse impulsionnelle, cas SISO). *La* réponse impulsionnelle *d'un filtre SISO $\mathscr{H}$, notée h, est le signal de sortie produit par $\mathscr{H}$ lorsqu'on lui donne en entrée le dirac δ (section 2.1.4) :*

$$h \triangleq \mathscr{H}\{\delta\} \tag{2.11}$$

```
Definition impulse_response (H : SISO_filter) : scsignal :=
  H (@dirac RT).
```

Caractérisation d'un filtre. La réponse impulsionnelle a de nombreuses propriétés intéressantes. Notamment, la sortie d'un filtre LTI est toujours égale au produit de convolution (également défini en section 2.1.4) de la réponse impulsionnelle h par l'entrée u :

$$\forall u, \quad \mathscr{H}\{u\} = h * u \tag{2.12}$$

```
Theorem convolution_impulse_response (H : SISO_filter) :
  LTI_filter H →
  forall u : @scsignal RT, H u = convols (impulse_response H) u.
```

La preuve s'appuie sur la décomposition d'un signal en somme de diracs retardés (`signal_sum_dirac` en section 2.1.4).

La réponse impulsionnelle suffit donc à caractériser un filtre. On peut alors définir des familles de filtres, ou déterminer les propriétés d'un filtre, à partir de la réponse impulsionnelle.

Définition 8 (FIR et IIR). *Un filtre LTI est dit* FIR *(*Finite Impulse Response*) si sa réponse impulsionnelle est nulle à partir d'un certain indice :* $\exists K, \forall k \geq K, h(k) = 0$*. Il est* IIR *(*Infinite Impulse Response*) si au contraire, sa réponse impulsionnelle a un support infini [96].*

Ces deux familles seraient faciles à définir en Coq, mais je n'ai pas eu besoin de les formaliser car toute la formalisation s'applique aux deux cas de toute façon.

Lien avec la causalité. Un filtre LTI est causal si et seulement si sa réponse impulsionnelle est causale. Comme on a vu en section 2.2.2, avec notre définition des signaux, tous les filtres LTI sont donc causals.

Lien avec la BIBO stabilité. Un filtre LTI est BIBO stable si et seulement si la série correspondant à sa réponse impulsionnelle converge absolument :

$$\sum_{k \in \mathbb{Z}} |h(k)| < +\infty \tag{2.13}$$

Cette propriété sera prouvée et utilisée dans la section 4.3.

Cas MIMO. Le cas MIMO est plus compliqué, puisqu'il n'y a pas d'équivalent simple du dirac. À la place, on regarde l'effet de chaque composante de l'entrée sur chaque composante de la sortie.

On note $\mathscr{H}_{ij}$ le filtre SISO correspondant à la variation de la sortie $\boldsymbol{y}_i$ en fonction de l'entrée $\boldsymbol{u}_j$ (le choix de noter l'indice de la sortie avant celui de l'entrée vient de la définition de la réponse impulsionnelle plus bas) :

$$\forall u \text{ (signal scalaire)}, \quad \mathscr{H}_{ij}\{u\} \triangleq (\mathscr{H}\{k \mapsto u(k)\boldsymbol{e_j}\})_i \tag{2.14}$$

où $\boldsymbol{e_j}$ est le vecteur unitaire avec un 1 en position j et des zéros ailleurs :

$$\begin{cases} (\boldsymbol{e_j})_j \triangleq 1 \\ (\boldsymbol{e_j})_p \triangleq 0 \text{ si } p \neq j \end{cases} \tag{2.15}$$

Définition 9 (Réponse impulsionnelle, cas MIMO). *La réponse impulsionnelle d'un filtre MIMO* $\mathscr{H}$*, notée*[3] $\boldsymbol{h}$*, est la matrice de taille* $n_y \times n_u$ *telle que le coefficient* $\boldsymbol{h}_{ij}$ *est la réponse impulsionnelle du filtre SISO* $\mathscr{H}_{ij}$*.*

La réponse impulsionnelle d'un filtre MIMO est donc une matrice de signaux scalaires. On peut aussi la voir comme un signal dont les valeurs $\boldsymbol{h}(k)$ sont des matrices. En Coq, on choisit la seconde option, représentée par le type `msignal` : cela s'intègre plus facilement au reste de la formalisation.

3. Exceptionnellement, on ne note pas cette matrice en majuscule, pour ne pas confondre avec le filtre $\mathscr{H}$ qui devient `H` en Coq, ni avec la fonction de transfert H (qui heureusement n'apparaît pas en Coq : voir la section 2.2.6).

```
Context {nu ny : nat}.
Definition to_SISO_yi_uj (H : MIMO_filter nu ny)
             (i : 'I_ny) (j : 'I_nu) : SISO_filter :=
  fun (u : scsignal) ⇒
    ith_scsignal (H (vsignal_of_sc_at_pos u j)) i.

Lemma MIMO_impulse_response_causal (H : MIMO_filter nu ny) :
 causal (fun k : int ⇒
   \matrix_(i, j) (impulse_response (to_SISO_yi_uj H i j) k)
                                              : 'M[RT]_(ny, nu)).
Definition MIMO_impulse_response
             (H : MIMO_filter nu ny) : msignal ny nu :=
  Build_signal MIMO_impulse_response_causal.
```

Caractérisation d'un filtre MIMO. Comme dans le cas SISO, un filtre MIMO est complètement caractérisé par sa réponse impulsionnelle. La sortie s'exprime encore comme le produit de convolution de la réponse impulsionnelle et de l'entrée :

$$\forall \boldsymbol{u}, \quad \mathscr{H}\{\boldsymbol{u}\} = \boldsymbol{h} * \boldsymbol{u} \tag{2.16}$$

Le produit de convolution entre un signal matriciel et un signal vectoriel est défini comme pour des signaux scalaires, sauf que les termes de la somme sont des produits matriciels :

$$(\boldsymbol{h} * \boldsymbol{u})(k) \triangleq \sum_{i \in \mathbb{Z}} \boldsymbol{h}(i)\boldsymbol{u}(k-i) \tag{2.17}$$

Rappelons que $\boldsymbol{h}(i)$ est une matrice de taille $n_y \times n_u$, que l'on peut donc bien multiplier par le vecteur $\boldsymbol{u}(k-i)$ de hauteur n_u.

La preuve de l'équation (2.16) se déduit facilement de son équivalent pour le cas SISO.

```
Theorem convolution_MIMO_impulse_response (H : MIMO_filter nu ny):
  LTI_filter H → forall u : vsignal nu,
  H u = convolution_mx_ve (MIMO_impulse_response H) u.
```

2.2.4 Équation aux différences et filtres d'ordre fini

Une autre manière très usuelle de caractériser un filtre SISO est de donner sa relation entrée-sortie sous la forme suivante, appelée *équation aux différences* :

$$\forall k \in \mathbb{Z}, \quad y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=1}^{n} a_i y(k-i) \tag{2.18}$$

Le filtre est alors caractérisé par l'entier n, appelé l'*ordre du filtre*, et les coefficients $(a_i)_{1 \leq i \leq n}$ et $(b_i)_{0 \leq i \leq n}$, appelés *coefficients de la fonction de transfert* (ce nom sera expliqué dans la section 2.2.6).

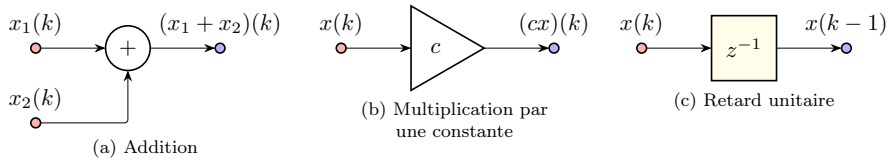
Une équation aux différences est une réalisation (section 1.4.5) qui est aussi connue sous le nom de *forme directe I*. Sa formalisation sera présentée dans la section 2.3.1.

On dit qu'un filtre est d'*ordre fini* s'il peut s'écrire sous la forme d'une équation aux différences. Ce n'est pas le cas de tous les filtres LTI définissables mathématiquement. En revanche, tous les filtres définissables algorithmiquement, c'est-à-dire tous les filtres qui ont une réalisation (section 1.4.5), sont d'ordre fini. Dans la suite, je m'intéresse donc seulement aux filtres d'ordre fini.

Remarque. Il ne faut pas confondre la notion d'ordre fini avec la distinction entre FIR et IIR introduite en section 2.2.3. Un filtre FIR est toujours d'ordre fini, mais un filtre IIR peut être d'ordre fini ou infini.

2.2.5 Graphe de flot de données

Un filtre LTI est souvent représenté par un *graphe de flot de données* : la sortie $y(k)$ est construite à partir de l'entrée $u(k)$ en utilisant les trois opérations élémentaires de la figure 2.2 [4]. On reconnaît les trois opérations basiques sur les signaux de la section 2.1.3. La seule différence est que le retard est restreint au retard unitaire, mais il suffit évidemment d'en enchaîner K pour représenter un retard de $K \geq 0$ unités de temps. Dans le cas d'un filtre MIMO, les entrées $\boldsymbol{u}_j(k)$ et sorties $\boldsymbol{y}_i(k)$ peuvent figurer séparément sur le graphe.



(a) Addition

(b) Multiplication par une constante

(c) Retard unitaire

FIGURE 2.2 – Composants élémentaires d'un graphe de flot de données

Un filtre représenté sous forme de graphe de flot de données est LTI par construction. Inversement, tout filtre LTI peut être représenté sous forme d'un tel graphe, à condition cependant d'autoriser les graphes infinis. Mais heureusement, les filtres auxquels on s'intéresse, à savoir les filtres LTI d'ordre fini définis dans la section 2.2.4, sont exactement les filtres représentables par un graphe de flot de données fini.

Exemple. Reprenons l'exemple jouet à deux entrées et deux sorties de la section 1.4.3 :

$$\begin{cases} \boldsymbol{y}_1(k) = 2\,(\,3\,\boldsymbol{u}_1(k) + \boldsymbol{u}_2(k-2)\,) - \frac{1}{2}\,\boldsymbol{y}_1(k-1) \\ \boldsymbol{y}_2(k) = 3\,(\,-4\,(\,7\,\boldsymbol{u}_2(k) + \boldsymbol{u}_2(k-1)\,) - \frac{1}{2}\,\boldsymbol{y}_1(k-1) + \frac{1}{5}\,\boldsymbol{y}_2(k-1)\,) \end{cases} \quad \text{(rappel de (1.20))}$$

Le graphe de flot de données correspondant est donné par la figure 2.3.

4. Le retard unitaire est noté z^{-1} car il prend cette forme dans le cadre de la transformée en z (voir section 2.2.6). On peut aussi rencontrer la notation q^{-1}.

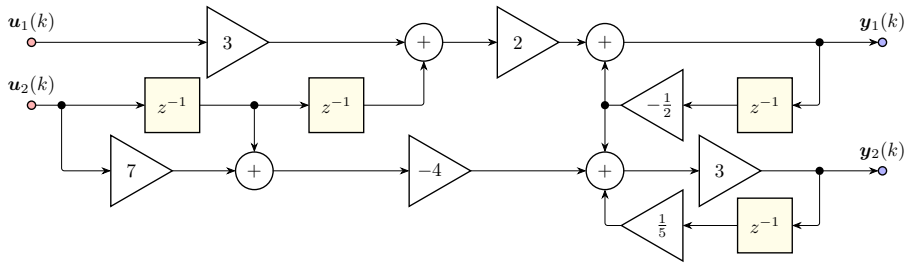


FIGURE 2.3 – Graphe de flot de données de l'exemple jouet

2.2.6 Domaine fréquentiel et fonction de transfert

Dans cette thèse, j'étudie les filtres dans le domaine temporel exclusivement : les signaux prennent des valeurs en fonction du temps k. Mais en traitement du signal, les filtres sont souvent considérés dans le domaine fréquentiel, notamment grâce à la fonction de transfert. Voici un bref aperçu de cette approche : l'objectif est de présenter un contexte un peu plus large et d'expliquer d'où vient l'appellation *coefficients de la fonction de transfert*. Cette section n'est pas formalisée en Coq.

Transformée en z. La *transformée en z*, notée $\mathcal{Z}\{.\}$, est une généralisation de la transformée de Fourier. Elle associée au signal x la fonction complexe $\mathcal{Z}\{x\} : \mathbb{C} \to \mathbb{C}$ telle que :

$$\forall z \in \mathbb{C}, \quad \mathcal{Z}\{x\}(z) = \sum_{k \in \mathbb{Z}} x(k) z^{-k} \tag{2.19}$$

Cela permet d'obtenir un décomposition fréquentielle du signal, qui a un sens physique fort. Par ailleurs, la transformée en z a des propriétés, notamment de linéarité et d'invariance dans le temps, qui sont très utiles dans le cadre des filtres LTI.

Fonction de transfert. On définit ensuite la *fonction de transfert* d'un filtre SISO $\mathscr{H}$, notée H, à partir des transformées en z d'une entrée u et la sortie correspondante $\mathscr{H}\{u\}$:

$$\forall z \in \mathbb{C}, \quad H(z) = \frac{\mathcal{Z}\{\mathscr{H}\{u\}\}(z)}{\mathcal{Z}\{u\}(z)} \tag{2.20}$$

Grâce aux propriétés de la transformée en z, la fonction de transfert H est indépendante de l'entrée u choisie et caractérise complètement le filtre. En choisissant comme entrée le dirac δ, on remarque que la fonction de transfert est la transformée en z de la réponse impulsionnelle : $H = \mathcal{Z}\{h\}$.

Coefficients de la fonction de transfert. Un résultat important est que la fonction de transfert du filtre décrit par l'équation aux différences (équa-

tion (2.18)) vérifie :

$$\forall z \in \mathbb{C}, \quad H(z) = \frac{\sum_{i=0}^{n} b_i z^{-i}}{1 + \sum_{i=1}^{n} a_i z^{-i}} \tag{2.21}$$

Un filtre est souvent choisi pour vérifier certaines propriétés fréquentielles. Une stratégie possible consiste alors à déterminer sa fonction de transfert, puis passer dans le domaine temporel grâce à l'équation aux différences ou une autre réalisation qui fait intervenir les mêmes coefficients a_i et b_i que la fonction de transfert. La section 2.3 donne d'autres exemples de telles réalisations. C'est pour cela que les a_i et b_i sont particulièrement connus pour être les *coefficients de la fonction de transfert*.

Cas MIMO. La fonction de transfert se généralise aux filtres MIMO de la même manière que la réponse impulsionnelle dans la section 2.2.3. La fonction de transfert $\boldsymbol{H}$ d'un filtre MIMO est la matrice de taille $n_y \times n_u$ telle que $\boldsymbol{H}_{ij}$ est la fonction de transfert du filtre SISO $\mathscr{H}_{ij}$. On peut donc la voir comme une matrice de fonctions $\mathbb{C} \to \mathbb{C}$, ou bien comme une seule fonction $\mathbb{C} \to \mathbb{C}^{n_y \times n_u}$.

Rien de cela ne fait partie de ma formalisation des filtres en Coq (sauf les coefficients de la fonction de transfert qui seront définis en Coq en section 2.3.1). En effet, ma formalisation se concentre sur les algorithmes de filtre dans le domaine temporel et leurs erreurs d'arrondi. Cependant, des formalisations du domaine fréquentiel en HOL ont été présentées en section 1.5.

2.3 Quelques familles usuelles de réalisations

On a vu en section 1.4.5 qu'un même filtre peut être décrit par une infinité de réalisations, et l'objectif est de trouver la meilleure possible selon un ou plusieurs critères : coût matériel, consommation d'énergie, complexités temps et mémoire, compatibilité avec l'architecture matérielle, comportement numérique en précision finie... Lorsqu'on cherche la bonne réalisation pour un filtre donné, on cherche en général parmi des familles connues de réalisations. Par famille de réalisations, j'entends une structure générale qu'on peut instancier à des coefficients particuliers afin d'obtenir une réalisation. Par exemple, l'équation aux différences (équation (2.18)) est une famille de réalisations : pour des coefficients (a_i) et (b_i) donnés, elle devient une réalisation du filtre dont la fonction de transfert a ces coefficients. Par abus de langage, les familles de réalisations sont souvent appelées simplement réalisations.

Dans cette section, je présente quelques familles usuelles de réalisations de filtre LTI : les formes directes I et II (section 2.3.1), leurs transposées (section 2.3.2), et enfin la décomposition parallèle et celle en cascade (section 2.3.3). Deux autres réalisations classiques, le *State-Space* et la SIF, feront l'objet du chapitre 3 à cause de leur importance particulière dans ma formalisation des erreurs d'arrondi dans les filtres en précision finie. Il existe beaucoup d'autres familles de réalisations LTI, par exemple les structures LGS [82] et LCW [81] et la forme directe II transposée avec opérateur ρ (ρDFIIt) [83].

Les formes directes I et II sont accompagnées de leur formalisation en Coq. Les autres réalisations présentées ici ne sont pas directement définies en Coq. Cependant, la construction d'une SIF décrivant le même filtre que ces réalisations sera formalisée dans la section 3.3.

2.3.1 Formes directes et coefficients de la fonction de transfert

Les *formes directes* sont des familles de réalisations qui dépendent directement des coefficients de la fonction de transfert (section 2.2.6) d'un filtre, d'où leur nom.

Forme directe I. La *forme directe I* (ou *DFI* pour l'anglais *Direct Form I*) est juste un autre nom de l'équation aux différences. C'est donc une famille de réalisations qui associe aux coefficients $(a_i)_{1\leq i\leq n}$ et $(b_i)_{0\leq i\leq n}$ la relation entrée-sortie :

$$\forall k,\quad y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=1}^{n} a_i y(k-i) \qquad \text{(rappel de (2.18))}$$

L'algorithme 3 définit le filtre implémenté associé, et le graphe de flot de données correspondant se trouve en figure 2.4a. On remarque qu'on peut calculer $y(k)$ avec une seule grande somme de produits, où les coefficients sont les b_i et les $(-a_i)$.

Algorithme 3 : Forme directe I

foreach k **do**
  $y(k) \leftarrow \sum_{i=0}^{n} b_i u(k-i) + \sum_{i=1}^{n} (-a_i) y(k-i)$
end

En observant l'algorithme, on constate qu'on a besoin de garder en mémoire les n valeurs précédentes de u et de y, ce qui fait $2n$ valeurs au total (sauf cas particulier si a_n ou b_n est nul). On remarque que c'est justement le nombre de retards unitaires dans le graphe de flot de données.

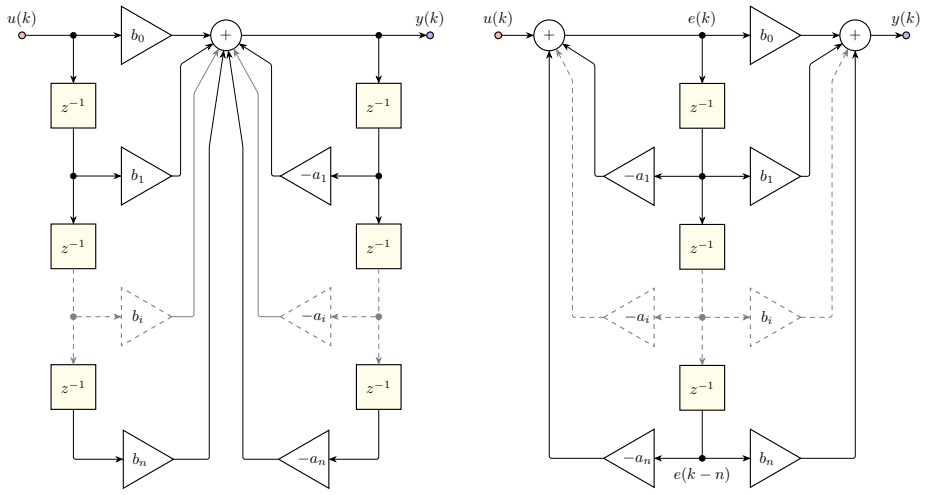
Forme directe II. La *forme directe II* (ou *DFII*) rassemble les informations à mémoriser dans un signal auxiliaire e. Aux coefficients (a_i) et (b_i) est associée la réalisation :

$$\forall k,\quad \begin{cases} e(k) = u(k) - \displaystyle\sum_{i=1}^{n} a_i e(k-i) \\ y(k) = \displaystyle\sum_{i=0}^{n} b_i e(k-i) \end{cases} \tag{2.22}$$

L'algorithme 4 définit le filtre implémenté associé, et le graphe de flot de données correspondant se trouve en figure 2.4b.

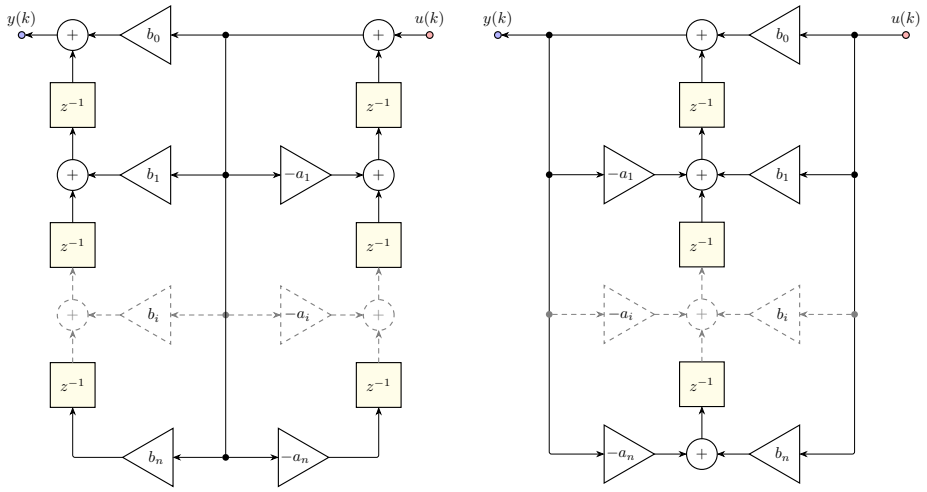
À chaque temps k, on calcule $e(k)$ et $y(k)$ en utilisant les n dernières valeurs de e. Il n'y a donc plus que n valeurs à mémoriser, ce qui est encore une fois le nombre de retards unitaires dans le graphe.

Comme les formes directes I et II utilisent toutes les deux les coefficients de la fonction de transfert, pour des coefficients donnés on doit retomber sur le même filtre qu'on passe par l'une ou par l'autre. Ceci est prouvé en Coq plus loin.



(a) Forme directe I (b) Forme directe II

FIGURE 2.4 – Graphes de flot de données des formes directes



(a) Forme directe I transposée (b) Forme directe II transposée

FIGURE 2.5 – Graphes de flot de données des formes directes transposées (l'entrée est à droite et la sortie à gauche pour mieux voir la transposition)

Algorithme 4 : Forme directe II

foreach k **do**

$$e(k) \leftarrow u(k) + \sum_{i=1}^{n} (-a_i) e(k-i)$$

$$y(k) \leftarrow \sum_{i=0}^{n} b_i e(k-i)$$

end

Formalisation. Afin de définir les formes directes I et II en Coq, on commence par construire un type pour les coefficients de la fonction de transfert : ils sont décrits par un ordre n et deux suites de coefficients a et b.

```
Record TFCoeffs :=
  { TFC_n : nat ; TFC_a : nat → RT ; TFC_b : nat → RT }.
```

En pratique, on utilisera seulement les valeurs de a sur les indices $1 \leq i \leq n$ et celles de b sur $0 \leq i \leq n$, mais encore une fois, c'est plus facile de travailler avec des fonctions totales. On aurait pu définir a et b comme des vecteurs mais cela n'a pas vraiment d'intérêt : soit on décale les indices de a ce qui est désastreux dans les définitions des formes directes, soit on a encore une valeur a_0 jamais utilisée qui empêche de gagner des propriétés comme l'unicité de la représentation d'un jeu de coefficients.

Pour construire le filtre obtenu à partir d'un jeu de coefficients en passant par la forme directe I, on utilise la construction d'un signal par récurrence forte présentée en section 2.1.5, qui requiert une fonction `f_IS : int → (int → VT) → VT` (où `IS` signifie *Induction Step*). Comme c'est la construction canonique d'un filtre à partir des coefficients de sa fonction de transfert, on l'appelle `filter_from_TFC`. On prouve que ce filtre est LTI.

```
Variable tfc : TFCoeffs.
Notation n := (TFC_n tfc).
Notation a := (TFC_a tfc).
Notation b := (TFC_b tfc).

Definition DFI_fIS (u : signal) : int → (int → VT) → VT :=
  fun (k : int) (y_before_k : int → VT) ⇒
      \sum_(0 ≤ i < n.+1) b i *: u (k - (i : int))
    - \sum_(1 ≤ i < n.+1) a i *: y_before_k (k - (i : int)).
Definition filter_from_TFC : filter :=
  fun (u : signal) ⇒ signal_strongrec (DFI_fIS u).

Theorem LTI_filter_from_TFC : LTI_filter filter_from_TFC.
```

Pour la forme directe II, on utilise la même construction par récurrence forte pour construire le signal auxiliaire e à partir de l'entrée u. Puis, y est facile à définir à partir de e : on montre simplement que le signal obtenu est bien causal.

```
Definition DFII_u2e_fIS (u : signal) :=
  fun (k : int) (e_before_k : int → VT) ⇒
    u k - \sum_(1 ≤ i < n.+1) a i *: e_before_k (k-(i:int)).
Fact DFII_e2y_causal (e : signal) :
  causal (fun k ⇒
            \sum_(0 ≤ i < n.+1) (TFC_b tfc i) *: e (k-(i:int))).
Definition DFII_e2y (e : signal) :=
  Build_signal (DFII_e2y_causal e).

Definition filter_by_DFII : filter :=
  fun (u : signal)⇒ DFII_e2y (signal_strongrec (DFII_u2e_fIS u)).
```

Enfin, on prouve qu'on obtient bien le même filtre en passant par une forme directe ou l'autre. La preuve est facile en utilisant le principe de récurrence forte `strongindz` sur `int`, également défini en section 2.1.5. Un corollaire est que le filtre construit par la forme directe II est bien LTI aussi.

```
Theorem DFI_DFII_same_filter : filter_from_TFC = filter_by_DFII.
```

2.3.2 Formes directes transposées

Le *théorème de transposition* [96] des graphes de flot de données énonce que le filtre décrit par un graphe SISO est conservé si on « retourne » le graphe : on inverse toutes les flèches ; on échange l'entrée et la sortie ; les embranchements (quand plusieurs flèches sont issues d'un même noeud) deviennent des additions, et réciproquement les additions deviennent des embranchements ; les multiplications par une constante et les retards ne sont pas modifiés à part que l'inversion des flèches change leur sens.

Les *formes directes I et II transposées* sont les réalisations correspondant aux graphes de la figure 2.5, obtenus en appliquant le théorème de transposition aux graphes des formes directes I et II respectivement (cette figure est placée avec la figure 2.4 pour pouvoir facilement comparer les graphes). Grâce au théorème en question, ces réalisations décrivent encore une fois le filtre dont la fonction de transfert a pour coefficients les (a_i) et (b_i) donnés.

On remarque que le graphe de la forme directe I transposée ressemble à celui de la forme directe II, et de même pour la forme directe II transposée avec la forme directe I : à chaque fois, seuls les retards sont positionnés différemment. Cependant, cela suffit à rendre le comportement en précision finie différent : les retards additionnels forcent l'utilisation de plusieurs petites additions dans les formes directes transposées, au lieu d'une seule grande somme de produits pour la forme directe I, ou deux sommes de produits pour la forme directe II.

Les formes directes transposées ne sont pas formalisées séparément en Coq. Néanmoins, la section 3.3.3 formalisera comment le filtre décrit par des coefficients de fonction de transfert (a_i) et (b_i) donnés peut être exprimé sous la forme d'une autre réalisation appelée la SIF, de telle sorte que les calculs effectués par cette nouvelle réalisation soient en fait les mêmes que ceux de la forme directe II transposée.

2.3.3 Décomposition parallèle ou en cascade

Étant données deux réalisations définissant des filtres $\mathscr{H}_1$ et $\mathscr{H}_2$, on peut les assembler de diverses manières pour obtenir une nouvelle réalisation. On notera $\mathscr{H}$ le filtre résultant.

On peut sommer les filtres, en définissant la sortie de $\mathscr{H}$ comme la somme des sorties de $\mathscr{H}_1$ et $\mathscr{H}_2$:

$$\forall u, \quad \mathscr{H}\{u\} = \mathscr{H}_1\{u\} + \mathscr{H}_2\{u\} \tag{2.23}$$

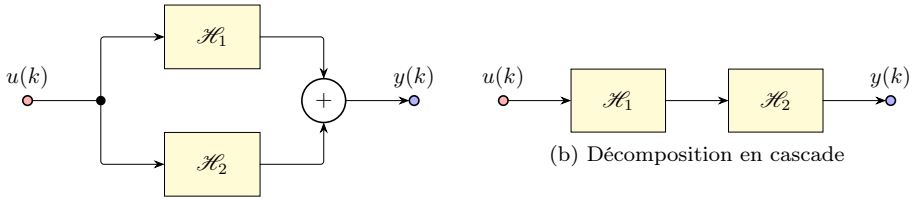
On dit alors que la réalisation finale est une *décomposition parallèle* en deux sous-réalisations, illustrée par la figure 2.6a.

On peut aussi brancher $\mathscr{H}_1$ et $\mathscr{H}_2$ à la suite l'un de l'autre, c'est-à-dire composer les fonctions mathématiques (rappelons que les filtres sont par définition

des fonctions sur les signaux) :

$$\forall u, \quad \mathscr{H}\{u\} = \mathscr{H}_2\{\mathscr{H}_1\{u\}\} \tag{2.24}$$

La réalisation obtenue est une *décomposition en cascade*, représentée sur la figure 2.6b.



(a) Décomposition parallèle

(b) Décomposition en cascade

FIGURE 2.6 – Décompositions en sous-réalisations

Remarque. Ces définitions s'appliquent techniquement à des filtres SISO comme MIMO. Cependant, elles sont utilisées presque exclusivement pour des filtres SISO. En effet, ces derniers se décomposent facilement en sous-filtres en considérant leur fonction de transfert, qui est une fraction rationnelle (section 2.2.6). La décomposition parallèle revient alors à décomposer celle-ci en une somme de fractions rationnelles plus petites, généralement du second ordre, tandis que la décomposition en cascade revient à factoriser la fonction de transfert.

Réponses impulsionnelles. Pour ces deux décompositions, la réponse impulsionnelle h du filtre final $\mathscr{H}$ s'exprime très facilement à partir des réponses impulsionnelles h_1 et h_2 des composants respectifs $\mathscr{H}_1$ et $\mathscr{H}_2$ (ici supposés SISO). Dans le cas de la décomposition parallèle, h est simplement leur somme : $h = h_1 + h_2$. Ce résultat s'obtient immédiatement à partir de la définition de la réponse impulsionnelle comme l'image du dirac :

$$h = \mathscr{H}\{\delta\} = \mathscr{H}_1\{\delta\} + \mathscr{H}_2\{\delta\} = h_1 + h_2 \tag{2.25}$$

Dans le cas de la décomposition en cascade, h est le produit de convolution des deux autres réponses impulsionnelles : $h = h_2 * h_1$. Cela provient de la caractérisation d'un filtre LTI par sa réponse impulsionnelle prouvée dans la section 2.2.3 (on rappelle que les filtres sont implicitement supposés LTI dans ce manuscrit) :

$$h = \mathscr{H}\{\delta\} = \mathscr{H}_2\{\mathscr{H}_1\{\delta\}\} = \mathscr{H}_2\{h_1\} = h_2 * h_1 \tag{2.26}$$

Formalisation. Je n'ai pas écrit de définition explicite de ces deux décompositions en Coq car on dispose déjà d'outils permettant des les exprimer facilement. Pour la décomposition parallèle, `GRing.add_fun_head` de MathComp renvoie la somme de deux fonctions. La notation infixe correspondante est `\+`. Pour la décomposition en cascade, `Basics.compose` de la bibliothèque standard définit la composition de deux fonctions.

Par exemple, on prouve très facilement les expressions des réponses impulsionnelles données ci-dessus.

```
Remark impulse_response_parallel (H1 H2 : SISO_filter) :
  impulse_response (H1 \+ H2) =
  impulse_response H1 + impulse_response H2.
Remark impulse_response_cascade (H1 H2 : SISO_filter) :
  LTI_filter H2 →
  impulse_response (Basics.compose H2 H1) =
  convols (impulse_response H2) (impulse_response H1).
```

Rappelons que `Remark` est pour Coq un synonyme de `Lemma`, `Theorem`, `Fact`, etc. Je l'utilise seulement pour signaler au lecteur que ces propriétés ne seront pas utilisées dans la suite de la formalisation (mais elles sont censées être très rapides à prouver, donc autant s'assurer qu'elles le sont vraiment).

Les sections 3.3.4 et 3.3.5 formaliseront les traductions de ces deux décompositions vers la SIF, une réalisation universelle qui fera l'objet du chapitre 3.

Chapitre 3

La SIF : réalisation universelle des filtres LTI implémentés

Comme expliqué dans les sections 1.4.5 et 2.3, l'implémentation d'un filtre repose sur un choix parmi une grande variété de réalisations. Ce choix, qui constitue à lui seul un domaine de recherche [49], dépend fortement du contexte. Beaucoup de familles de réalisations différentes sont donc utilisées en pratique : mon analyse formelle des erreurs d'arrondi doit être applicable à chacune d'entre elles. Cependant, écrire une formalisation distincte pour chacune n'est bien sûr pas souhaitable. Ce problème est résolu grâce à une réalisation appelée la *SIF* (*Specialized Implicit Form*) [54, 55, 56]. Pour n'importe quelle réalisation de filtre LTI, on peut construire une SIF qui décrit le même algorithme : à la fois le filtre modèle $\mathscr{H}$ et le filtre implémenté $\mathscr{H}^*$ sont conservés. La SIF est donc une *réalisation universelle des filtres LTI modèles et implémentés*. En particulier, il suffit d'effectuer l'analyse des erreurs d'arrondi de la SIF : cette analyse (qui fait l'objet du chapitre 4) s'appliquera alors à n'importe quelle réalisation de filtre LTI.

Dans un premier temps, la section 3.1 présente le grand frère de la SIF : le *State-Space* [70]. Il s'agit d'une réalisation qui est elle-même couramment utilisée, notamment par la communauté de l'automatique. Mais ici, son intérêt est surtout didactique : le *State-Space* est beaucoup plus simple à expliquer que la SIF, et les idées derrière sa construction sont reprises et étendues par la SIF. Plus précisément, le *State-Space* est déjà une *réalisation universelle pour les filtres modèles* : pour n'importe quelle réalisation de filtre LTI, on peut construire un *State-Space* qui décrit le même filtre modèle $\mathscr{H}$ (c'est-à-dire la même relation mathématique entrée-sortie, définie avec des additions et multiplications exactes). En revanche, il n'existe *pas forcément un State-Space qui décrit le même filtre implémenté* $\mathscr{H}^*$ que la réalisation initiale : la structure des calculs est souvent modifiée, si bien que le comportement numérique en précision finie risque d'être différent (voir la section 3.1.7). La SIF se base sur le *State-Space* et hérite son universalité pour les filtres modèles, mais l'étend de manière à gagner l'universalité pour les filtres implémentés.

La section 3.2 définit la SIF elle-même avec ses principales propriétés. La section 3.3 présente des traductions des réalisations introduites en section 2.3 vers la SIF préservant le filtre implémenté. Enfin, la section 3.4 décrit des transforma-

tions d'une SIF vers une autre SIF préservant le filtre modèle mais modifiant le filtre implémenté afin d'essayer de lui donner de meilleures propriétés pratiques (compatibilité avec l'architecture matérielle, coût, comportement en précision finie, etc.).

3.1 Le *State-Space* : réalisation universelle des filtres modèles

Comme expliqué ci-dessus, le *State-Space*, couramment utilisé en automatique, est une réalisation universelle des filtres modèles, et aussi un bon moyen pour moi d'introduire des principes qui seront réutilisés par la SIF. Le *State-Space* exprime n'importe quel filtre modèle LTI d'ordre fini sous une forme matricielle concise. Il utilise pour cela un *vecteur d'état* $\boldsymbol{x}(k)$ qui décrit l'état interne du filtre, c'est-à-dire toutes les informations du passé dont on a encore besoin pour le présent ou le futur. Cela est détaillé dans la section 3.1.1. C'est de là que vient le nom : *représentation d'état*, même si je préfère utiliser le nom anglais *State-Space*. Les sections 3.1.2 et 3.1.3 définissent le *State-Space*, d'abord mathématiquement puis en Coq. La section 3.1.4 présente ses propriétés. La section 3.1.5 explique comment mettre n'importe quel filtre modèle sous forme de *State-Space*, et illustre la méthode proposée sur un exemple. La section 3.1.6 construit un *State-Space* correspondant à un filtre donné sous la forme des coefficients de sa fonction de transfert, en passant par la forme directe II. Enfin, la section 3.1.7 explique pourquoi beaucoup de filtres implémentés ne peuvent pas être représentés sous forme de *State-Space*. Le problème de l'universalité pour les filtres implémentés sera résolu par la SIF dans la section 3.2.

3.1.1 Principe : vecteur d'état $\boldsymbol{x}(k)$

Le *State-Space* est basé sur l'utilisation d'un *signal d'état*, noté $\boldsymbol{x}$, comme mémoire du filtre : le *vecteur d'état* $\boldsymbol{x}(k)$ à l'instant k stocke toutes les informations du passé dont on a encore besoin pour calculer la sortie $\boldsymbol{y}(k)$ ou les sorties futures. Ses composantes $\boldsymbol{x}_i(k)$ sont appelées les *variables d'état*.

Considérons notre exemple jouet à deux entrées et une sortie :

$$\boldsymbol{y}_1(k) = 2\,(\,3\,\boldsymbol{u}_1(k) + \boldsymbol{u}_2(k-2)\,) - \frac{1}{2}\,\boldsymbol{y}_1(k-1) \qquad \text{(rappel de (1.19))}$$

Au temps k, pour calculer $\boldsymbol{y}_1(k)$, on remarque qu'on a besoin de deux informations du passé : $\boldsymbol{y}_1(k-1)$ et $\boldsymbol{u}_2(k-2)$, qui doivent donc être disponibles dans $\boldsymbol{x}(k)$. Mais on voit aussi qu'on aura besoin de $\boldsymbol{u}_2(k-1)$ pour calculer la sortie suivante $\boldsymbol{y}_1(k+1)$, donc cette valeur doit aussi être stockée dans $\boldsymbol{x}(k)$. Ainsi, on pose par exemple :

$$\boldsymbol{x}(k) = \begin{pmatrix} \boldsymbol{u}_2(k-1) \\ \boldsymbol{u}_2(k-2) \\ \boldsymbol{y}_1(k-1) \end{pmatrix} \tag{3.1}$$

L'ordre des composantes de $\boldsymbol{x}(k)$ n'a pas d'importance ; ce qui compte est que toutes les informations nécessaires y figurent.

De manière générale, il suffit d'assigner une composante de $\boldsymbol{x}$ à chaque retard unitaire présent dans le graphe de flot de données du filtre. Le graphe de

l'exemple précédent est donné par la figure 3.1. Ce graphe a bien trois retards unitaires qui correspondent aux $\boldsymbol{x}_i$ définis ci-dessus. La flèche qui sort du nœud correspond à $\boldsymbol{x}_i(k)$ et indique comment sa valeur est utilisée. Celle qui arrive sur le nœud montre comment l'état suivant $\boldsymbol{x}_i(k+1)$ est calculé.

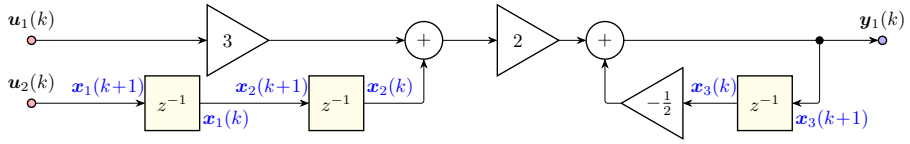


FIGURE 3.1 – Graphe de flot de données du premier exemple jouet avec les $\boldsymbol{x}_i$ pour le *State-Space*

Les sections 3.1.5 et 3.1.6 contiennent d'autres exemples de construction d'un vecteur d'état $\boldsymbol{x}(k)$ adapté à un filtre donné sous forme d'un graphe de flot de données.

La sortie $\boldsymbol{y}(k)$ peut alors être calculée à partir de seulement l'entrée courante $\boldsymbol{u}(k)$ et le vecteur d'état $\boldsymbol{x}(k)$. Il faut aussi mettre à jour la mémoire, c'est-à-dire calculer le vecteur d'état suivant $\boldsymbol{x}(k+1)$, lui aussi à partir de $\boldsymbol{u}(k)$ et $\boldsymbol{x}(k)$ uniquement. Sur l'exemple, en regardant la figure 3.1 :

$$\begin{cases} \boldsymbol{x}_1(k+1) = \boldsymbol{u}_2(k) \\ \boldsymbol{x}_2(k+1) = \boldsymbol{x}_1(k) \\ \boldsymbol{x}_3(k+1) = 2\,(\,3\,\boldsymbol{u}_1(k) + \boldsymbol{x}_2(k)\,) - \dfrac{1}{2}\,\boldsymbol{x}_3(k) \\ \quad \boldsymbol{y}_1(k) = 2\,(\,3\,\boldsymbol{u}_1(k) + \boldsymbol{x}_2(k)\,) - \dfrac{1}{2}\,\boldsymbol{x}_3(k) \end{cases} \tag{3.2}$$

De manière générale, un filtre LTI est construit à partir des trois opérations de base et le signal d'état gère les retards, donc il ne reste plus que des additions et multiplications par une constante. Chaque composante de $\boldsymbol{y}(k)$ et $\boldsymbol{x}(k+1)$ peut donc être exprimée comme une combinaison linéaire des composantes de $\boldsymbol{u}(k)$ et $\boldsymbol{x}(k)$, en développant au besoin : par exemple $2\,(\,3\,\boldsymbol{u}_1(k) + \boldsymbol{u}_2(k-2)\,) = 6\,\boldsymbol{u}_1(k) + 2\,\boldsymbol{u}_2(k-2)$. C'est pour cela que la définition du *State-Space* ci-dessous sera bien universelle pour les filtres modèles.

3.1.2 Définition mathématique

Définition 10 (*State-Space*). *Un* State-Space *[70] est la donnée de* quatre matrices de coefficients $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$*, de tailles compatibles comme indiqué ci-dessous. Le filtre modèle MIMO $\mathscr{H}$ correspondant à ce State-Space est défini par la relation entrée-sortie* [1] *:*

$$\mathscr{H} \quad \begin{cases} \boldsymbol{x}(k+1) &= \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}\boldsymbol{u}(k) \\ \boldsymbol{y}(k) &= \boldsymbol{C}\boldsymbol{x}(k) + \boldsymbol{D}\boldsymbol{u}(k) \end{cases} \tag{3.3}$$

1. Comme d'habitude, les signaux sont initialisés à 0 pour $k < 0$. En appliquant la relation de récurrence, on a donc $\boldsymbol{x}(0) = \boldsymbol{0}$. D'autres définitions autorisent une initialisation arbitraire pour $\boldsymbol{x}(0)$, mais si $\boldsymbol{x}(0) \neq \boldsymbol{0}$ alors le filtre n'est pas LTI (à moins que $\boldsymbol{C} = \boldsymbol{0}$, auquel cas le filtre est juste une multiplication par une constante et $\boldsymbol{x}$ ne sert à rien).

En notant n_x la taille de $\boldsymbol{x}$, les tailles des matrices sont donc :

$$\begin{array}{ll} \boldsymbol{A} \in \mathtt{RT}^{n_x \times n_x} & \boldsymbol{B} \in \mathtt{RT}^{n_x \times n_u} \\ \boldsymbol{C} \in \mathtt{RT}^{n_y \times n_x} & \boldsymbol{D} \in \mathtt{RT}^{n_y \times n_u} \end{array} \tag{3.4}$$

Graphe de flot de données. La figure 3.2 représente le *State-Space* sous forme de graphe de flot de données. Les nœuds du graphe représentent des opérations matricielles, ce qui est assez inhabituel : normalement, il faudrait faire apparaître chaque composante de vecteur séparément, mais le graphe deviendrait vite surchargé. On a vu en section 3.1.1 que la taille de $\boldsymbol{x}(k)$ correspond au nombre de retards unitaires dans le graphe, mais c'était en considérant des flots scalaires. Ici, les fils représentent des signaux vectoriels donc il suffit d'un retard unitaire vectoriel pour décrire entièrement $\boldsymbol{x}$.

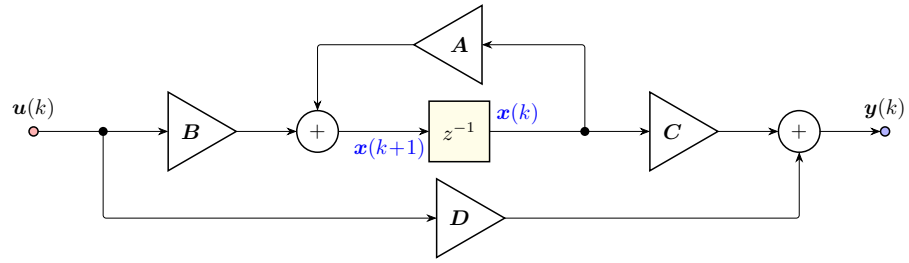


FIGURE 3.2 – *State-Space* : graphe de flot de données

Écriture implicite et matrice $\boldsymbol{Z}$. L'équation (3.3) peut s'écrire en une seule égalité faisant intervenir des matrices par blocs, appelée *écriture implicite* :

$$\left(\begin{array}{c} \boldsymbol{x}(k+1) \\ \hline \boldsymbol{y}(k) \end{array}\right) = \left(\begin{array}{c|c} \boldsymbol{A} & \boldsymbol{B} \\ \hline \boldsymbol{C} & \boldsymbol{D} \end{array}\right) \left(\begin{array}{c} \boldsymbol{x}(k) \\ \hline \boldsymbol{u}(k) \end{array}\right) \tag{3.5}$$

On note $\boldsymbol{Z}$ cette matrice qui contient tous les coefficients : $\boldsymbol{Z} \triangleq \left(\begin{array}{c|c} \boldsymbol{A} & \boldsymbol{B} \\ \hline \boldsymbol{C} & \boldsymbol{D} \end{array}\right)$. Un *State-Space* est complètement décrit par cette matrice $\boldsymbol{Z}$ (à condition d'expliciter le découpage par blocs, ou de manière équivalente, connaître n_u, n_y ou n_x). Dans la suite, un *State-Space* sera donc souvent donné sous cette forme.

***State-Space* SISO.** Un filtre SISO peut bien sûr être représenté par un *State-Space* : il suffit de le voir comme un filtre MIMO dont l'entrée et la sortie sont de taille 1. On dira donc qu'un *State-Space* est SISO lorsque $n_u = n_y = 1$. On peut quand même avoir besoin de stocker plusieurs valeurs d'une étape sur l'autre donc on peut avoir $n_x > 1$. La matrice $\boldsymbol{B}$ est donc un vecteur colonne, $\boldsymbol{C}$ est un vecteur ligne, et $\boldsymbol{D}$ est de taille 1×1. Cela explique qu'on rencontre parfois les notations $(\boldsymbol{A},\boldsymbol{b},\boldsymbol{c},d)$ pour un *State-Space* SISO (notations que je n'utilise pas).

Algorithme. En réécrivant l'équation (3.3) composante par composante, on obtient l'algorithme 5. Notons qu'en pratique, les calculs sont plus légers que ce que suggère l'algorithme, car les matrices de coefficients sont souvent creuses.

Algorithme 5 : *State-Space*

foreach k **do**
  for $i \in \{1, \dots, n_x\}$ **do**
    $\boldsymbol{x}_i(k+1) \leftarrow \sum_{j=1}^{n_x} \boldsymbol{A}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{B}_{ij}\boldsymbol{u}_j(k)$
  end
  for $i \in \{1, \dots, n_y\}$ **do**
    $\boldsymbol{y}_i(k) \leftarrow \sum_{j=1}^{n_x} \boldsymbol{C}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{D}_{ij}\boldsymbol{u}_j(k)$
  end
end

Filtre implémenté. Énonçons maintenant le filtre implémenté (présenté en section 1.4.4) associé à un *State-Space* pour un format de précision finie $\mathbb{F}$ et un arrondi $\bigcirc$ vers ce format donnés (comme défini en section 1.3.3). Notons qu'on considère un seul format et un seul arrondi pour simplifier les notations, mais ce n'est pas difficile à étendre à des formats et arrondis différents pour chaque variable ou opération. On utilise les notations introduites en section 1.4.4 : le filtre implémenté $\mathscr{H}^*$ produit la sortie $\boldsymbol{y}^*$, qui est une approximation de la sortie $\boldsymbol{y}$ du filtre modèle $\mathscr{H}$ pour une même entrée $\boldsymbol{u}$. De plus, on note $\boldsymbol{x}^*$ le signal d'état calculé en précision finie par $\mathscr{H}^*$. L'objectif est d'exprimer $\boldsymbol{x}^*$ et $\boldsymbol{y}^*$ à partir de l'entrée $\boldsymbol{u}$ et des coefficients $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$. Pour cela, on doit prendre en compte les deux effets suivants de la précision finie.

D'une part, toutes les opérations dans les calculs de $\boldsymbol{x}^*$ et $\boldsymbol{y}^*$ doivent être effectuées en précision finie. On veut utiliser des opérations de somme de produits autant que possible plutôt que des additions ou multiplications individuelles. On remarque dans l'algorithme 5 que chaque composante des vecteurs à calculer peut être obtenue avec une seule grande somme de produits. On rappelle qu'on note $\odot$ le produit matriciel en précision finie, où chaque coefficient est le résultat d'une somme de produits en précision finie.

D'autre part, les coefficients des matrices $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ ne sont pas nécessairement dans le format de précision finie considéré. Pour pouvoir les utiliser dans les calculs, il faut les arrondir vers un nombre du format. On parle de *quantification des coefficients* (*quantization* en anglais). On rappelle qu'on note $\bigcirc(\boldsymbol{M})$ la matrice obtenue à partir d'une matrice $\boldsymbol{M}$ en arrondissant chaque coefficient.

Le filtre implémenté d'un *State-Space* s'exprime donc ainsi :

$$\mathscr{H}^* \quad \begin{cases} \boldsymbol{x}^*(k+1) &= \left(\bigcirc(\boldsymbol{A}) \mid \bigcirc(\boldsymbol{B}) \right) \odot \begin{pmatrix} \boldsymbol{x}^*(k) \\ \boldsymbol{u}(k) \end{pmatrix} \\ \boldsymbol{y}^*(k) &= \left(\bigcirc(\boldsymbol{C}) \mid \bigcirc(\boldsymbol{D}) \right) \odot \begin{pmatrix} \boldsymbol{x}^*(k) \\ \boldsymbol{u}(k) \end{pmatrix} \end{cases} \tag{3.6}$$

3.1.3 Définition Coq

Un *State-Space* en Coq est simplement la donnée des quatre matrices de coefficients $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ de la définition 10, rassemblées dans un type enregistrement (`Record`).

```
Context {nu ny nx : nat}.
Record StateSpace := { StSp_A : 'M[RT]_(nx)     ;
                       StSp_B : 'M[RT]_(nx, nu) ;
                       StSp_C : 'M[RT]_(ny, nx) ;
                       StSp_D : 'M[RT]_(ny, nu) }.
```

Pour définir le filtre modèle associé, on se donne un *State-Space* `stsp` et un signal d'entrée $\boldsymbol{u}$, et on construit les signaux $\boldsymbol{x}$ et $\boldsymbol{y}$ correspondants en utilisant l'équation (3.3). Pour $\boldsymbol{x}$, on a besoin de la construction récursive usuelle `build_signal_peano_rec` présentée en section 2.1.5. Le signal $\boldsymbol{y}$ est directement construit à partir de la fonction appropriée, qui est trivialement causale. Ces définitions sont placées dans une section de Coq, dont le mécanisme a été expliqué en section 1.2.4.

```
Section Build_output.

Variables (stsp : StateSpace) (u : vsignal nu).
Notation A := (StSp_A stsp). (* idem pour B, C, D *)

Definition x : vsignal nx :=
  build_signal_peano_rec (fun k (xk : 'cV_nx) ⇒
                                        (A *m xk) + (B *m u k)).

Fact y_causal : causal (fun k ⇒ C *m x k + D *m u k).
Definition y := Build_signal y_causal.

End Build_output.
```

Une fois la section `Build_output` fermée, les variables `stsp` et `u` deviennent des arguments de la fonction `y`, qui est alors de type `StateSpace → signal → signal`, c'est-à-dire `StateSpace → filter`. On a donc construit une fonction qui donne le filtre modèle associé à un *State-Space*. On la renomme `filter_from_StSp`.

```
Definition filter_from_StSp : StateSpace → filter := y.
```

On pourrait définir le filtre implémenté pour des formats et arrondis donnés en ajoutant des annotations à chaque calcul, mais ce n'est pas très intéressant pour ma formalisation. En effet, on verra en section 3.1.7 que le *State-Space* n'est pas une réalisation universelle pour les filtres implémentés. Lorsqu'on étudiera des filtres implémentés, on utilisera donc plutôt la SIF présentée en section 3.2. Si un filtre implémenté est tout de même fourni sous la forme d'un *State-Space*, il suffit de le convertir en une SIF grâce à la traduction formalisée en section 3.2.5.

3.1.4 Propriétés du filtre associé à un *State-Space*

Cette section présente quelques propriétés du filtre modèle correspondant à un *State-Space*.

Filtre LTI. On prouve facilement que le filtre associé à un *State-Space* est LTI.

```
Theorem filter_from_StSp_is_LTI (stsp : StateSpace) :
  LTI_filter (filter_from_StSp stsp).
```

Réponse impulsionnelle. La réponse impulsionnelle (section 2.2.3) du filtre associé au *State-Space* $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ vérifie :

$$\boldsymbol{h}(k) = \begin{cases} \boldsymbol{D} & \text{si } k = 0 \\ \boldsymbol{C}\boldsymbol{A}^{k-1}\boldsymbol{B} & \text{si } k > 0 \\ \boldsymbol{0} & \text{si } k < 0 \end{cases} \tag{3.7}$$

```
Theorem StateSpace_impulse_response (stsp : StateSpace) (k : int) :
  MIMO_impulse_response (filter_from_StSp stsp) k
    = match k with
        | Posz 0     ⇒ D
        | Posz (S _) ⇒ C *m expmx A (k-1) *m B
        | Negz _     ⇒ 0
      end.
```

Cette propriété est facile à prouver. Une étape intermédiaire consiste à montrer que si un signal d'entrée $\boldsymbol{u}$ vérifie $\forall k > 0, \boldsymbol{u}(k) = \boldsymbol{0}$, alors le signal d'état $\boldsymbol{x}$ correspondant vérifie : $\forall k > 0$, $\boldsymbol{x}(k) = \boldsymbol{A}^{k-1}\boldsymbol{B}\boldsymbol{u}(0)$.

On remarque que si le filtre est SISO alors $\boldsymbol{D}$ et $\boldsymbol{C}\boldsymbol{A}^{k-1}\boldsymbol{B}$ sont de taille 1×1 donc on obtient bien une réponse impulsionnelle scalaire.

Fonction de transfert. Comme expliqué en section 2.2.6, la fonction de transfert n'est ni étudiée ni formalisée dans cette thèse. Néanmoins, on peut noter qu'elle s'exprime facilement à partir des coefficients d'un *State-Space* :

$$\forall z \in \mathbb{C}, \quad \boldsymbol{H}(z) = \boldsymbol{C}(z\boldsymbol{I}_n - \boldsymbol{A})^{-1}\boldsymbol{B} + \boldsymbol{D} \tag{3.8}$$

3.1.5 D'un filtre modèle donné à un *State-Space* : exemple

On a vu que le *State-Space* est une réalisation universelle pour les filtres modèles LTI. Cette section explique informellement comment traduire n'importe quelle réalisation vers un *State-Space* en conservant le filtre modèle décrit, et illustre ce procédé sur un exemple. La section 3.1.6 appliquera ensuite cette méthode à la forme directe II, afin de donner un *State-Space* décrivant le filtre correspondant à des coefficients de fonction de transfert donnés. Enfin, la section 3.1.7 expliquera pourquoi on ne peut généralement pas construire un *State-Space* décrivant un filtre implémenté donné.

Voici comment traduire une réalisation donnée vers un *State-Space* décrivant le même filtre modèle :

— Mettre cette réalisation sous forme de graphe de flot de données (section 2.2.5).

— Associer une variable d'état $\boldsymbol{x}_i(k)$ à chaque délai unitaire du graphe ; l'ordre n'a pas d'importance.

— Exprimer chaque coefficient de $\boldsymbol{x}(k+1)$ et $\boldsymbol{y}(k)$ en fonction de ceux de $\boldsymbol{x}(k)$ et $\boldsymbol{u}(k)$. En comparant avec l'équation (3.3), en déduire les coefficients des matrices $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$.

On peut exprimer le même filtre modèle avec beaucoup de *State-Spaces* différents. Par exemple, si on permute les coefficients de $\boldsymbol{x}(k)$, on obtient un autre *State-Space* décrivant le même filtre modèle. Il s'agit donc simplement de construire un *State-Space* convenable parmi d'autres.

Illustrons cette méthode sur notre exemple jouet de filtre à deux entrées et deux sorties :

$$\begin{cases} \boldsymbol{y}_1(k) = 2\,(\,3\,\boldsymbol{u}_1(k) + \boldsymbol{u}_2(k-2)\,) - \frac{1}{2}\,\boldsymbol{y}_1(k-1) \\ \boldsymbol{y}_2(k) = 3\,(\,-4\,(\,7\,\boldsymbol{u}_2(k) + \boldsymbol{u}_2(k-1)\,) - \frac{1}{2}\,\boldsymbol{y}_1(k-1) + \frac{1}{5}\,\boldsymbol{y}_2(k-1)\,) \end{cases}$$

(rappel de (1.20))

Le graphe de flot de données correspondant, donné en figure 2.3, est rappelé en figure 3.3.

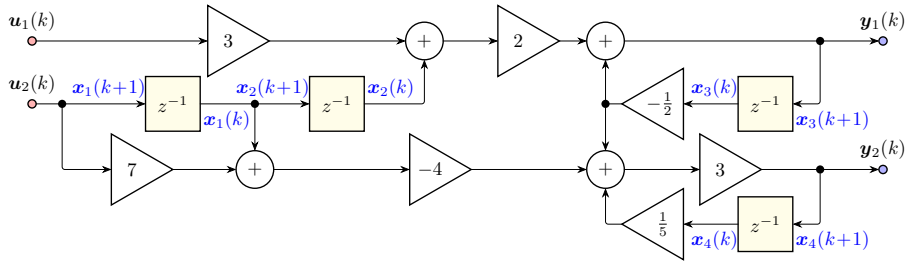


FIGURE 3.3 – Graphe de flot de données du second exemple jouet avec les $\boldsymbol{x}_i$ pour le *State-Space*

On associe une composante de $\boldsymbol{x}$ à chacun des quatre retards unitaires de ce graphe, comme indiqué sur la figure 3.3. Le signal d'état $\boldsymbol{x}$ peut donc être défini par :

$$\boldsymbol{x}(k+1) = \begin{pmatrix} \boldsymbol{u}_2(k) \\ \boldsymbol{u}_2(k-1) \\ \boldsymbol{y}_1(k) \\ \boldsymbol{y}_2(k) \end{pmatrix} \tag{3.9}$$

On veut maintenant exprimer $\boldsymbol{x}(k+1)$ en fonction de seulement $\boldsymbol{x}(k)$ et $\boldsymbol{u}(k)$, pas de $\boldsymbol{y}(k)$. En lisant sur le graphe :

$$\begin{aligned} \boldsymbol{x}(k+1) &= \begin{pmatrix} \boldsymbol{u}_2(k) \\ \boldsymbol{x}_1(k) \\ 2\,(\,3\,\boldsymbol{u}_1(k) + \boldsymbol{x}_2(k)\,) - \frac{1}{2}\,\boldsymbol{x}_3(k) \\ 3\,(\,-4\,(\,7\,\boldsymbol{u}_2(k) + \boldsymbol{x}_1(k)\,) - \frac{1}{2}\,\boldsymbol{x}_3(k) + \frac{1}{5}\,\boldsymbol{x}_4(k)\,) \end{pmatrix} \\ &= \begin{pmatrix} \boldsymbol{u}_2(k) \\ \boldsymbol{x}_1(k) \\ 6\,\boldsymbol{u}_1(k) + 2\,\boldsymbol{x}_2(k) - \frac{1}{2}\,\boldsymbol{x}_3(k) \\ -84\,\boldsymbol{u}_2(k) - 12\,\boldsymbol{x}_1(k) - \frac{3}{2}\,\boldsymbol{x}_3(k) + \frac{3}{5}\,\boldsymbol{x}_4(k) \end{pmatrix} \end{aligned} \tag{3.10}$$

On en déduit les matrices $\boldsymbol{A}$ et $\boldsymbol{B}$ du *State-Space* qu'on est en train de construire :

$$\boldsymbol{x}(k+1) = \underbrace{\begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 2 & -\frac{1}{2} & 0 \\ -12 & 0 & -\frac{3}{2} & \frac{3}{5} \end{pmatrix}}_{\boldsymbol{A}} \boldsymbol{x}(k) + \underbrace{\begin{pmatrix} 0 & 1 \\ 0 & 0 \\ 6 & 0 \\ 0 & -84 \end{pmatrix}}_{\boldsymbol{B}} \boldsymbol{u}(k) \tag{3.11}$$

Pour déterminer $\boldsymbol{C}$ et $\boldsymbol{D}$, on fait la même chose avec la sortie $\boldsymbol{y}(k)$ exprimée en fonction de $\boldsymbol{x}(k)$ et $\boldsymbol{u}(k)$. On remarque que $\boldsymbol{y}_1(k) = \boldsymbol{x}_3(k+1)$ et $\boldsymbol{y}_2(k) = \boldsymbol{x}_4(k+1)$, donc il suffit de prendre les deux dernières lignes de $\boldsymbol{x}(k+1)$ pour avoir $\boldsymbol{y}(k)$, et les matrices $\boldsymbol{C}$ et $\boldsymbol{D}$ sont les deux dernières lignes de $\boldsymbol{A}$ et $\boldsymbol{B}$ respectivement :

$$\begin{aligned} \boldsymbol{y}(k) &= \begin{pmatrix} 6\,\boldsymbol{u}_1(k) + 2\,\boldsymbol{x}_2(k) - \frac{1}{2}\,\boldsymbol{x}_3(k) \\ -84\,\boldsymbol{u}_2(k) - 12\,\boldsymbol{x}_1(k) - \frac{3}{2}\,\boldsymbol{x}_3(k) + \frac{3}{5}\,\boldsymbol{x}_4(k) \end{pmatrix} \\ \boldsymbol{y}(k) &= \underbrace{\begin{pmatrix} 0 & 2 & -\frac{1}{2} & 0 \\ -12 & 0 & -\frac{3}{2} & \frac{3}{5} \end{pmatrix}}_{\boldsymbol{C}} \boldsymbol{x}(k) + \underbrace{\begin{pmatrix} 6 & 0 \\ 0 & -84 \end{pmatrix}}_{\boldsymbol{D}} \boldsymbol{u}(k) \end{aligned} \tag{3.12}$$

On a ainsi construit le *State-Space* suivant, qui décrit le filtre modèle défini par l'équation (1.20) et illustré par la figure 3.3 :

$$\boldsymbol{Z} = \left(\begin{array}{cccc|cc} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & -\frac{1}{2} & 0 & 6 & 0 \\ -12 & 0 & -\frac{3}{2} & \frac{3}{5} & 0 & -84 \\ \hline 0 & 2 & -\frac{1}{2} & 0 & 6 & 0 \\ -12 & 0 & -\frac{3}{2} & \frac{3}{5} & 0 & -84 \end{array}\right) \tag{3.13}$$

Comme mentionné en section 3.1.2, on remarque que cette matrice est assez creuse.

3.1.6 *State-Space* correspondant à la forme directe II

Un filtre est souvent donné sous la forme des coefficients de sa fonction de transfert. Il est donc utile de connaître un *State-Space* décrivant le filtre modèle correspondant à de tels coefficients.

Soit $(a_i)_{1\leq i\leq n}$, $(b_i)_{0\leq i\leq n}$ des coefficients de fonction de transfert. On veut exprimer le filtre correspondant sous forme de *State-Space* en utilisant la construction présentée en section 3.1.5. Comme expliqué en section 2.2.6, le filtre associé à ces coefficients est SISO, alors que le filtre associé à un *State-Space* est MIMO. On va donc voir ce filtre SISO comme un filtre MIMO avec $n_u = n_y = 1$. On notera u et y les signaux scalaires d'entrée et sortie du filtre SISO, et $\boldsymbol{u}$ et $\boldsymbol{y}$ les signaux vectoriels de taille 1 correspondants du point de vue MIMO :

$$\boldsymbol{u}(k) = \Big(u(k)\Big) \qquad \boldsymbol{y}(k) = \Big(y(k)\Big) \tag{3.14}$$

La première étape consiste à mettre le filtre sous forme de graphe de flot de données. On a le choix entre plusieurs réalisations qui décrivent toutes le filtre modèle associé aux coefficients de la fonction de transfert, notamment les formes directes I et II et leurs transposées (sections 2.3.1 à 2.3.2). *On choisit la forme directe II* (section 2.3.1), définie par la relation suivante, où e est un signal auxiliaire :

$$\begin{cases} e(k) = u(k) - \sum_{i=1}^{n} a_i e(k-i) \\ y(k) = \sum_{i=0}^{n} b_i e(k-i) \end{cases} \qquad \text{(rappel de l'équation (2.22))}$$

L'intérêt de ce choix est que la forme directe II isole déjà les informations à mémoriser d'une étape sur l'autre dans le signal auxiliaire e. La forme directe II est donc facile à mettre sous forme de *State-Space* sans même regarder le graphe de flot de données (qu'on peut quand même consulter dans la figure 2.4b de la section 2.3.1).

La deuxième étape consiste à déterminer le vecteur d'état $\boldsymbol{x}(k)$. On voit dans la relation ci-dessus que pour calculer $e(k)$ et $y(k)$, on a besoin des valeurs passées $e(k-1)$, $e(k-2)$, …, $e(k-n)$, où n est l'ordre du filtre. On pose donc :

$$\boldsymbol{x}(k) = \begin{pmatrix} e(k-n) \\ \vdots \\ e(k-1) \end{pmatrix} \tag{3.15}$$

Encore une fois, l'ordre des composantes de $\boldsymbol{x}$ n'a pas d'importance mathématiquement. D'un point de vue algorithmique, l'ordre choisi facilite un peu la mise à jour du vecteur $\boldsymbol{x}(k)$ en place. On peut vérifier sur le graphe de la figure 2.4b que $\boldsymbol{x}_1, \ldots, x_n$ correspondent bien aux retards unitaires (de bas en haut à cause de l'ordre choisi).

Enfin, on détermine les matrices de coefficients en regardant comment $\boldsymbol{x}(k+1)$ et $\boldsymbol{y}(k)$ s'obtiennent à partir de $\boldsymbol{x}(k)$ et $u(k)$. Pour les matrices $\boldsymbol{A}$ et $\boldsymbol{B}$, on s'intéresse à $\boldsymbol{x}(k+1)$.

$$\boldsymbol{x}(k+1) = \begin{pmatrix} e(k+1-n) \\ e(k+1-(n-1)) \\ \vdots \\ e(k+1-2) \\ e(k+1-1) \end{pmatrix} = \begin{pmatrix} e(k-(n-1)) \\ e(k-(n-2)) \\ \vdots \\ e(k-1) \\ e(k) \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_2(k) \\ \boldsymbol{x}_3(k) \\ \vdots \\ \boldsymbol{x}_n(k) \\ e(k) \end{pmatrix}$$

or $e(k) = u(k) - \sum_{i=1}^{n} a_i e(k-i) = u(k) - a_1 \boldsymbol{x}_n(k) - \cdots - a_n \boldsymbol{x}_1(k)$ donc

$$\boldsymbol{x}(k+1) = \begin{pmatrix} \boldsymbol{x}_2(k) \\ \boldsymbol{x}_3(k) \\ \vdots \\ \boldsymbol{x}_n(k) \\ -a_1\boldsymbol{x}_n(k) - \cdots - a_n\boldsymbol{x}_1(k) \end{pmatrix} + \begin{pmatrix} 0 \\ \vdots \\ 0 \\ u(k) \end{pmatrix}$$

$$\boldsymbol{x}(k+1) = \underbrace{\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ & & & & 0 \\ 0 & \cdots & & 0 & 1 \\ -a_n & -a_{n-1} & \cdots & & -a_1 \end{pmatrix}}_{\boldsymbol{A}} \boldsymbol{x}(k) + \underbrace{\begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}}_{\boldsymbol{B}} \boldsymbol{u}(k) \tag{3.16}$$

Pour les matrices $\boldsymbol{C}$ et $\boldsymbol{D}$, on regarde le calcul de $y(k)$.

$$\begin{aligned} y(k) &= \sum_{i=0}^{n} b_i e(k-i) \\ &= b_0\Big(u(k) - a_1\boldsymbol{x}_n(k) - \cdots - a_n\boldsymbol{x}_1(k)\Big) + b_1\boldsymbol{x}_n(k) + \cdots + b_n\boldsymbol{x}_1(k) \\ \boldsymbol{y}(k) &= \underbrace{\begin{pmatrix} b_n - b_0a_n & b_{n-1} - b_0a_{n-1} & \cdots & b_1 - b_0a_1 \end{pmatrix}}_{\boldsymbol{C}} \boldsymbol{x}(k) \\ &\qquad + \underbrace{\begin{pmatrix} b_0 \end{pmatrix}}_{\boldsymbol{D}} \boldsymbol{u}(k) \end{aligned} \tag{3.17}$$

On a ainsi construit un *State-Space* correspondant au filtre dont les coefficients de la fonction de transfert sont les a_i et b_i donnés. Ce *State-Space* est caractérisé par la matrice $\boldsymbol{Z}$ suivante :

$$\boldsymbol{Z} = \left(\begin{array}{cccc|c} 0 & 1 & 0 & 0 & 0 \\ \vdots & & \ddots & \vdots & \vdots \\ & & & 0 & 0 \\ 0 & \cdots & 0 & 1 & 0 \\ -a_n & -a_{n-1} & \cdots & -a_1 & 1 \\ \hline b_n - b_0a_n & b_{n-1} - b_0a_{n-1} & \cdots & b_1 - b_0a_1 & b_0 \end{array}\right) \tag{3.18}$$

Formalisation. Pour des coefficients de fonction de transfert donnés, on définit le *State-Space* obtenu ci-dessus, et on prouve que le filtre modèle décrit par ce *State-Space* est bien le filtre correspondant à ces coefficients.

```
Variable tfc : @TFCoeffs RT.
Notation n := (TFC_n tfc).
Notation a := (TFC_a tfc).
Notation b := (TFC_b tfc).
Definition StSp_from_TFC : StateSpace := @Build_StateSpace RT 1 1 n
  (* StSp_A *) (\matrix_(i,j) (if (i:nat) == (n-1)%nat then - a (n-j)
                                 else if (i+1)%nat == j then 1 else 0))
  (* StSp_B *) (\col_i (if (i:nat) == (n-1)%nat then 1 else 0))
  (* StSp_C *) (\row_j (b (n-j) - a (n-j) * b 0))
  (* StSp_D *) (b 0)%:M1.
Theorem StSp_from_TFC_correct :
  to_SISO (filter_from_StSp StSp_from_TFC) = filter_from_TFC tfc.
```

La condition `(i:nat) == (n-1)%nat` caractérise la dernière ligne de $\boldsymbol{A}$ et $\boldsymbol{B}$ puisque `i` est un ordinal de `'I_n` $= \{0, \ldots, n-1\}$ converti en entier. La fonction `filter_from_TFC` définie en section 2.3.1 produit directement un filtre SISO, alors que `filter_from_StSp` donne toujours un filtre MIMO, qui ici vérifie $n_u = n_y = 1$: on utilise `to_SISO` (section 2.2.1) pour passer du second au premier. De plus, `filter_from_TFC` se base sur la forme directe I. Comme on a construit le *State-Space* à partir de la forme directe II, la preuve utilise `DFI_DFII_same_filter` (section 2.3.1) pour passer d'une forme directe à l'autre. Ensuite, la preuve consiste principalement à montrer récursivement que le signal $\boldsymbol{x}$ est bien constitué des valeurs antérieures de signal e de la forme directe II.

3.1.7 Le *State-Space* ne représente pas tous les filtres implémentés

Dans la section 3.1.5, pour mettre l'exemple sous forme de *State-Space*, on a écrit :

$$\begin{aligned}\boldsymbol{x}(k+1) &= \begin{pmatrix} \boldsymbol{u}_2(k) \\ \boldsymbol{x}_1(k) \\ 2\,(\,3\,\boldsymbol{u}_1(k) + \boldsymbol{x}_2(k)\,) - \frac{1}{2}\,\boldsymbol{x}_3(k) \\ 3\,(\,-4\,(\,7\,\boldsymbol{u}_2(k) + \boldsymbol{x}_1(k)\,) - \frac{1}{2}\,\boldsymbol{x}_3(k) + \frac{1}{5}\,\boldsymbol{x}_4(k)\,) \end{pmatrix} \\ &= \begin{pmatrix} \boldsymbol{u}_2(k) \\ \boldsymbol{x}_1(k) \\ 6\,\boldsymbol{u}_1(k) + 2\,\boldsymbol{x}_2(k) - \frac{1}{2}\,\boldsymbol{x}_3(k) \\ -84\,\boldsymbol{u}_2(k) - 12\,\boldsymbol{x}_1(k) - \frac{3}{2}\,\boldsymbol{x}_3(k) + \frac{3}{5}\,\boldsymbol{x}_4(k) \end{pmatrix} \end{aligned}$$

(rappel de (3.10))

On a eu besoin de développer les expressions des deux dernières lignes. En effet, dans un *State-Space*, chaque composante de $\boldsymbol{x}(k+1)$ et $\boldsymbol{y}(k)$ est exprimée directement comme une combinaison linéaire des composantes de $\boldsymbol{u}(k)$ et $\boldsymbol{x}(k)$. La dépendance de $\boldsymbol{x}_3(k)$ en $\boldsymbol{u}_1(k)$ doit donc être exprimée par un seul coefficient : on doit mettre le produit $2 \times 3 = 6$ dedans, ce qui force le développement de l'expression $2\,(\,3\boldsymbol{u}_1(k) + \boldsymbol{u}_2(k-2)\,)$. Cela ne pose pas de problème lorsqu'on

s'intéresse au filtre modèle. Mais en précision finie, on risque de ne pas obtenir le même résultat selon qu'on calcule $2\,(\,3\,\boldsymbol{u}_1(k)+\boldsymbol{x}_2(k)\,)-\frac{1}{2}\,\boldsymbol{x}_3(k)$ ou $6\,\boldsymbol{u}_1(k)+2\,\boldsymbol{x}_2(k)-\frac{1}{2}\,\boldsymbol{x}_3(k)$.

Ainsi, dès qu'un filtre implémenté fait intervenir des calculs qui ne sont pas équivalents à une somme de produits d'entrées ou de valeurs obtenues à des étapes antérieures (donc stockables dans des états) par des constantes, ce filtre implémenté ne peut pas être représenté par un *State-Space*.

Pour pouvoir exprimer n'importe quel filtre implémenté, il faut pouvoir transcrire la structure des calculs. C'est exactement ce que fera la SIF dans la section 3.2 grâce à des variables intermédiaires.

3.2 La SIF : réalisation universelle des filtres implémentés

Comme expliqué au début de ce chapitre, l'objectif est de trouver une forme universelle à laquelle on peut se ramener depuis n'importe quelle réalisation, pour qu'une analyse d'erreurs d'arrondi effectuée sur cette forme puisse être appliquée à n'importe quel filtre LTI. On a vu que pour n'importe quelle réalisation, on peut construire un *State-Space* qui définit le même filtre modèle. Cependant, ce n'est pas le cas pour le filtre implémenté : comme expliqué en section 3.1.7, le *State-Space* risque de modifier la structure des calculs et donc le comportement en précision finie. Or quand on étudie les erreurs d'arrondi, c'est le filtre implémenté qui nous intéresse. On a donc besoin d'une réalisation qui est universelle aussi pour les filtre implémentés : la SIF (*Specialized Implicit Form*) [54, 55, 56].

La SIF est une extension du *State-Space*, beaucoup plus complexe avec neuf matrices de coefficients au lieu de quatre. Elle garde le vecteur d'état $\boldsymbol{x}(k)$ du *State-Space* pour mémoriser les valeurs nécessaires. Mais elle utilise aussi un *vecteur auxiliaire* $\boldsymbol{t}(k+1)$ accompagné d'une matrice $\boldsymbol{J}$ de forme particulière, qui permettent à eux deux de décrire n'importe quels calculs intermédiaires. Ce mécanisme, détaillé dans la section 3.2.1, permet à la SIF de reproduire fidèlement n'importe quelle structure de calculs. La section 3.2.2 énonce la définition mathématique de la SIF, et la section 3.2.3 présente sa définition en Coq. La section 3.2.4 donne un exemple de construction d'une SIF décrivant un filtre implémenté donné. Enfin, la section 3.2.5 explique comment passer d'un *State-Space* à une SIF et réciproquement ; notamment, cela permet à la SIF de récupérer les propriétés du *State-Space* présentées en section 3.1.4.

3.2.1 Principe : vecteur auxiliaire $\boldsymbol{t}(k+1)$ et matrice $\boldsymbol{J}$ pour structurer les calculs

Reprenons le premier exemple jouet de filtre à deux entrées et une sortie :

$$\boldsymbol{y}_1(k) = 2\,(\,3\,\boldsymbol{u}_1(k)+\boldsymbol{u}_2(k-2)\,)-\frac{1}{2}\,\boldsymbol{y}_1(k-1) \qquad \text{(rappel de (1.19))}$$

On a vu en section 3.1.1 que pour le mettre sous forme de *State-Space*, on peut poser :

$$\boldsymbol{x}(k) = \begin{pmatrix} \boldsymbol{u}_2(k-1) \\ \boldsymbol{u}_2(k-2) \\ \boldsymbol{y}_1(k-1) \end{pmatrix} \qquad \text{(rappel de (3.1))}$$

et on a alors :

$$\begin{cases} \boldsymbol{x}_1(k+1) = \boldsymbol{u}_2(k) \\ \boldsymbol{x}_2(k+1) = \boldsymbol{x}_1(k) \\ \boldsymbol{x}_3(k+1) = 2\,(\,3\,\boldsymbol{u}_1(k) + \boldsymbol{x}_2(k)\,) - \dfrac{1}{2}\,\boldsymbol{x}_3(k) \\ \quad \boldsymbol{y}_1(k) = 2\,(\,3\,\boldsymbol{u}_1(k) + \boldsymbol{x}_2(k)\,) - \dfrac{1}{2}\,\boldsymbol{x}_3(k) \end{cases} \qquad \text{(rappel de (3.2))}$$

On a vu en section 3.1.7 que le problème avec le *State-Space* est que chaque composante de $\boldsymbol{x}(k+1)$ et $\boldsymbol{y}(k)$ doit être exprimée directement comme une combinaison linéaire des composantes de $\boldsymbol{u}(k)$ et $\boldsymbol{x}(k)$. En particulier, on ne dispose que d'un seul coefficient pour exprimer la dépendance de $\boldsymbol{y}_1(k)$ en $\boldsymbol{u}_1(k)$. Ici, on doit mettre $2 * 3 = 6$ dans ce coefficient, ce qui nous force à développer l'expression $2\,(\,3\boldsymbol{u}_1(k) + \boldsymbol{x}_2(k)\,)$.

Des résultats intermédiaires pour structurer les calculs. Pour éviter ce problème, on introduit un vecteur $\boldsymbol{t}(k+1)$ permettant de définir des résultats intermédiaires : chaque composante de $\boldsymbol{t}(k+1)$ peut être utilisée comme une variable temporaire dans les calculs de $\boldsymbol{x}(k+1)$ et $\boldsymbol{y}(k)$ à partir de $\boldsymbol{x}(k)$ et $\boldsymbol{u}(k)$. L'indice $k+1$ dans $\boldsymbol{t}(k+1)$ sera expliqué dans la section 3.2.2. J'appelle $\boldsymbol{t}(k+1)$ le *vecteur auxiliaire* ou *vecteur intermédiaire*. En considérant les $\boldsymbol{t}(k+1)$ pour chaque k, on obtient un signal $\boldsymbol{t}$ appelé *signal auxiliaire* ou *signal intermédiaire*.

Dans l'exemple considéré, on utilise une composante de $\boldsymbol{t}(k+1)$ pour ajouter une étape au calcul de $\boldsymbol{y}_1(k)$:

$$\begin{aligned} \boldsymbol{t}_1(k+1) &\leftarrow 3\boldsymbol{u}_1(k) + \boldsymbol{x}_2(k) \\ \boldsymbol{y}_1(k) &\leftarrow 2\boldsymbol{t}_1(k+1) - \frac{1}{2}\,\boldsymbol{x}_3(k) \end{aligned} \qquad (3.19)$$

La variable intermédiaire $\boldsymbol{t}_1(k+1)$, illustrée sur la figure 3.4, assure que la structure des calculs reste la même que dans l'équation (1.19). Ainsi, même en précision finie, les résultats calculés seront les mêmes.

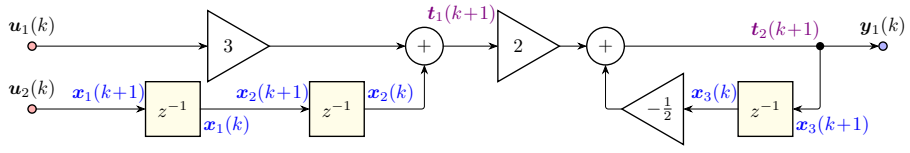


FIGURE 3.4 – Graphe de flot de données du premier exemple jouet avec les $\boldsymbol{x}_i$ et $\boldsymbol{t}_i$ pour la SIF

Notons que la SIF garde le même vecteur d'état $\boldsymbol{x}(k)$ que le *State-Space* : les valeurs des $\boldsymbol{x}_i(k)$ restent les mêmes, comme on peut le voir sur la figure 3.4. Ce qui change est la manière dont ils sont calculés.

Les signaux $\boldsymbol{t}$ et $\boldsymbol{x}$ ont des rôles très différents. Le signal d'état $\boldsymbol{x}$ sert uniquement à sauvegarder des valeurs d'une étape à la suivante : au temps k, on utilise seulement le vecteur d'état courant $\boldsymbol{x}(k)$, et on calcule seulement sa valeur suivante $\boldsymbol{x}(k+1)$. Au contraire, le même vecteur intermédiaire est à la fois calculé et utilisé au cours de la même étape, mais jamais réutilisé d'une étape sur l'autre. Par convention, le vecteur intermédiaire au temps k est noté $\boldsymbol{t}(k+1)$ plutôt que $\boldsymbol{t}(k)$ car on verra en section 3.2.2 que cela permet d'écrire la SIF comme un *State-Space* implicite [4].

Réutilisation des calculs. Un autre avantage des résultats temporaires est d'éviter des duplications inutiles de calculs. Dans l'exemple, on remarque que comme par construction $\boldsymbol{x}_3(k+1) = \boldsymbol{y}_1(k)$, on fait les mêmes calculs pour obtenir $\boldsymbol{x}_3(k+1)$ et $\boldsymbol{y}_1(k)$. Le *State-Space* n'a aucun moyen de mettre ces calculs en commun. Mais grâce au vecteur intermédiaire, on peut stocker le résultat dans $\boldsymbol{t}_2(k+1)$ puis le réutiliser directement pour $\boldsymbol{x}_3(k+1)$ et $\boldsymbol{y}_1(k)$. La succession de calculs à effectuer à l'étape k est alors :

$$\begin{aligned}
\boldsymbol{t}_1(k+1) &\leftarrow 3\,\boldsymbol{u}_1(k) + \boldsymbol{x}_2(k) \\
\boldsymbol{t}_2(k+1) &\leftarrow 2\,\boldsymbol{t}_1(k+1) - \frac{1}{2}\,\boldsymbol{x}_3(k) \\
\boldsymbol{x}_1(k+1) &\leftarrow \boldsymbol{u}_2(k) \\
\boldsymbol{x}_2(k+1) &\leftarrow \boldsymbol{x}_1(k) \\
\boldsymbol{x}_3(k+1) &\leftarrow \boldsymbol{t}_2(k+1) \\
\boldsymbol{y}_1(k) &\leftarrow \boldsymbol{t}_2(k+1)
\end{aligned} \tag{3.20}$$

Interdépendance et ordre de calcul des $\boldsymbol{t}_i(k+1)$. Dans les calculs précédents, on remarque que $\boldsymbol{t}_2(k+1)$ dépend de $\boldsymbol{t}_1(k+1)$. Pour pouvoir enchaîner des calculs intermédiaires mais éviter des dépendances circulaires, on décide que chaque $\boldsymbol{t}_i(k+1)$ peut seulement dépendre des $\boldsymbol{t}_j(k+1)$ pour $j < i$: tout se passe donc bien si on les calcule dans l'ordre des indices croissants. On souhaite exprimer cela sous une forme matricielle concise. Sur l'exemple, on remarque que les calculs de $\boldsymbol{t}_1(k+1)$ et $\boldsymbol{t}_2(k+1)$ peuvent se récrire :

$$\begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix} \boldsymbol{t}(k+1) = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{2} \end{pmatrix} \boldsymbol{x}(k) + \begin{pmatrix} 3 & 0 \\ 0 & 0 \end{pmatrix} \tag{3.21}$$

On voit que les sous-matrices supérieures concernent $\boldsymbol{t}_1(k+1)$ et les sous-matrices inférieures $\boldsymbol{t}_2(k+1)$ grâce aux 1 sur la diagonale de la matrice devant $\boldsymbol{t}(k+1)$. De plus, le 0 dans cette matrice signifie bien que $\boldsymbol{t}_1(k+1)$ ne dépend pas de $\boldsymbol{t}_2(k+1)$. En revanche, le calcul de $\boldsymbol{t}_2(k+1)$ fait intervenir un terme $2\,\boldsymbol{t}_1(k+1)$: ceci est représenté par le coefficient -2 (qui a changé de signe en passant du membre droit au membre gauche de l'égalité). Le fait que chaque $\boldsymbol{t}_i(k+1)$ ne puisse dépendre que des $\boldsymbol{t}_j(k+1)$ pour $j < i$ est donc représenté par la multiplication de $\boldsymbol{t}(k+1)$ par une matrice triangulaire inférieure.

Cette remarque s'étend au cas général de la façon suivante. Soit $\boldsymbol{J}$ une matrice carrée triangulaire inférieure avec des 1 sur la diagonale, et $\boldsymbol{w}$ un vecteur de même hauteur que $\boldsymbol{J}$. On dispose alors d'un algorithme naturel pour construire un vecteur $\boldsymbol{v}$ tel que $\boldsymbol{Jv} = \boldsymbol{w}$. De plus, cet algorithme fait que chaque $\boldsymbol{v}_i$ peut dépendre des $\boldsymbol{v}_j$ pour $j < i$ mais pas pour $j > i$. Détaillons cet algorithme dans

le cas où les vecteurs sont de taille 3. On a alors :

$$\begin{pmatrix} 1 & 0 & 0 \\ \boldsymbol{J}_{2,1} & 1 & 0 \\ \boldsymbol{J}_{3,1} & \boldsymbol{J}_{3,2} & 1 \end{pmatrix} \boldsymbol{v} = \boldsymbol{w} \tag{3.22}$$

En développant le produit matriciel et en isolant $\boldsymbol{v}_i$ dans la i-ème ligne, on obtient :

$$\begin{aligned} \boldsymbol{v}_1 &= \boldsymbol{w}_1 \\ \boldsymbol{v}_2 &= \boldsymbol{w}_2 - \boldsymbol{J}_{2,1}\boldsymbol{v}_1 \\ \boldsymbol{v}_3 &= \boldsymbol{w}_3 - \boldsymbol{J}_{3,1}\boldsymbol{v}_1 - \boldsymbol{J}_{3,2}\boldsymbol{v}_2 \end{aligned} \tag{3.23}$$

Chaque $\boldsymbol{v}_i$ dépend donc seulement de $\boldsymbol{w}_i$ et des $\boldsymbol{v}_j$ pour $j < i$. On remarque que l'algorithme est similaire à la fin d'un pivot de Gauss.

Ainsi, on va écrire :

$$\boldsymbol{J}\boldsymbol{t}(k+1) = \boldsymbol{w} \tag{3.24}$$

où $\boldsymbol{J}$ est une matrice carrée triangulaire inférieure avec des 1 sur la diagonale, et $\boldsymbol{w}$ est un vecteur qui ne dépend pas de $\boldsymbol{t}(k+1)$ mais peut tout à faire dépendre de $\boldsymbol{x}(k)$ et $\boldsymbol{u}(k)$. On aura alors un algorithme qui calcule les $\boldsymbol{t}_i(k+1)$ dans le bon ordre avec les bonnes dépendances.

Là où le *State-Space* avait deux vecteurs $\boldsymbol{x}(k+1)$ et $\boldsymbol{y}(k)$ à calculer à partir de deux vecteurs $\boldsymbol{x}(k)$ et $\boldsymbol{u}(k)$, la SIF aura donc trois vecteurs $\boldsymbol{t}(k+1)$, $\boldsymbol{x}(k+1)$ et $\boldsymbol{y}(k)$ à calculer à partir de trois vecteurs $\boldsymbol{t}(k+1)$, $\boldsymbol{x}(k)$ et $\boldsymbol{u}(k)$. La dépendance de $\boldsymbol{t}(k+1)$ en lui-même sera gérée par une matrice $\boldsymbol{J}$ comme expliqué ci-dessus.

3.2.2 Définition mathématique

On définit d'abord proprement la propriété dont on a eu besoin sur la matrice $\boldsymbol{J}$ dans la section précédente. On appelle cette propriété *Jprop*. On peut ensuite définir la SIF.

Définition 11 (*Jprop*). *Une matrice carrée vérifie la propriété* Jprop *si elle est triangulaire inférieure avec des 1 sur la diagonale.*

Définition 12 (SIF). *Une* SIF *(*Specialized Implicit Form*) est la donnée de* neuf matrices de coefficients $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$*, de tailles compatibles comme indiqué ci-dessous ; de plus, la matrice $\boldsymbol{J}$ doit vérifier la propriété Jprop ci-dessus. Le filtre modèle MIMO $\mathscr{H}$ correspondant à cette SIF est défini par la relation entrée-sortie :*

$$\mathscr{H} \quad \left\{ \begin{array}{rcl} \boldsymbol{J}\boldsymbol{t}(k+1) & = & \boldsymbol{M}\boldsymbol{x}(k) + \boldsymbol{N}\boldsymbol{u}(k) \\ \boldsymbol{x}(k+1) & = & \boldsymbol{K}\boldsymbol{t}(k+1) + \boldsymbol{P}\boldsymbol{x}(k) + \boldsymbol{Q}\boldsymbol{u}(k) \\ \boldsymbol{y}(k) & = & \boldsymbol{L}\boldsymbol{t}(k+1) + \boldsymbol{R}\boldsymbol{x}(k) + \boldsymbol{S}\boldsymbol{u}(k) \end{array} \right. \tag{3.25}$$

En notant n_t et n_x les tailles respectives de $\boldsymbol{t}$ et $\boldsymbol{x}$, les tailles des matrices sont :

$$\begin{array}{lll} \boldsymbol{J} \in \mathtt{RT}^{n_t \times n_t} & \boldsymbol{M} \in \mathtt{RT}^{n_t \times n_x} & \boldsymbol{N} \in \mathtt{RT}^{n_t \times n_u} \\ \boldsymbol{K} \in \mathtt{RT}^{n_x \times n_t} & \boldsymbol{P} \in \mathtt{RT}^{n_x \times n_x} & \boldsymbol{Q} \in \mathtt{RT}^{n_x \times n_u} \\ \boldsymbol{L} \in \mathtt{RT}^{n_y \times n_t} & \boldsymbol{R} \in \mathtt{RT}^{n_y \times n_x} & \boldsymbol{S} \in \mathtt{RT}^{n_y \times n_u} \end{array} \tag{3.26}$$

On remarque que si on enlève $\boldsymbol{t}(k+1)$ de l'équation (3.25) ainsi que toutes les matrices qui sont liées à $\boldsymbol{t}(k+1)$: $\boldsymbol{J}$, $\boldsymbol{M}$, $\boldsymbol{N}$, $\boldsymbol{K}$ et $\boldsymbol{L}$, on retombe exactement sur l'équation (3.3) définissant le *State-Space*, avec $(\boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ comme matrices de coefficients.

Écriture implicite et matrice $\boldsymbol{Z}$. Comme pour le *State-Space*, l'équation (3.25) a une écriture implicite, avec une seule égalité faisant intervenir des matrices par blocs :

$$\left(\begin{array}{c|c|c} \boldsymbol{J} & \boldsymbol{0} & \boldsymbol{0} \\ \hline -\boldsymbol{K} & \boldsymbol{I} & \boldsymbol{0} \\ \hline -\boldsymbol{L} & \boldsymbol{0} & \boldsymbol{I} \end{array}\right) \left(\begin{array}{c} \boldsymbol{t}(k+1) \\ \hline \boldsymbol{x}(k+1) \\ \hline \boldsymbol{y}(k) \end{array}\right) = \left(\begin{array}{c|c|c} \boldsymbol{0} & \boldsymbol{M} & \boldsymbol{N} \\ \hline \boldsymbol{0} & \boldsymbol{P} & \boldsymbol{Q} \\ \hline \boldsymbol{0} & \boldsymbol{R} & \boldsymbol{S} \end{array}\right) \left(\begin{array}{c} \boldsymbol{t}(k) \\ \hline \boldsymbol{x}(k) \\ \hline \boldsymbol{u}(k) \end{array}\right) \tag{3.27}$$

On remarque que le vecteur $\boldsymbol{t}(k)$ apparaissant dans le membre droit est multiplié seulement par des coefficients nuls : on aurait pu le remplacer par n'importe quoi, mais écrire $\boldsymbol{t}(k)$ a plus de sens par rapport au reste de l'équation. De plus, c'est en vue de cette écriture implicite que le vecteur intermédiaire pour le temps k, figurant ici dans le membre gauche, est noté $\boldsymbol{t}(k+1)$ plutôt que $\boldsymbol{t}(k)$. En effet, cela permet de toujours concaténer verticalement des valeurs de $\boldsymbol{t}$ et $\boldsymbol{x}$ pour le même indice temporel. En rassemblant $\boldsymbol{t}$ et $\boldsymbol{x}$ dans un même vecteur, on retrouve alors un *State-Space* mais avec un terme à gauche : on parle de *State-Space implicite* [4], d'où vient le nom *Specialized Implicit Form*.

Cette fois-ci, la matrice $\boldsymbol{Z}$ choisie pour décrire complètement la réalisation n'apparaît pas directement dans l'égalité ci-dessus. On l'obtient en groupant les coefficients variables des deux matrices par blocs de part et d'autre de l'égalité (en prenant l'opposé de ceux qui était à gauche, car on peut considérer qu'ils changent de côté pour rejoindre ceux à droite) :

$$\boldsymbol{Z} \triangleq \left(\begin{array}{c|c|c} -\boldsymbol{J} & \boldsymbol{M} & \boldsymbol{N} \\ \hline \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{Q} \\ \hline \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{S} \end{array}\right) \tag{3.28}$$

Cette écriture de la matrice $\boldsymbol{Z}$, en particulier le signe « $-$ » devant $\boldsymbol{J}$, permet aussi d'obtenir des formules plus simples pour la mesure de sensibilité de la fonction de transfert en fonction des coefficients [106].

Dans la suite, une SIF particulière sera souvent donnée sous la forme de juste cette matrice $\boldsymbol{Z}$.

Algorithme. Voici l'algorithme détaillé obtenu à partir de l'équation (3.25). Le calcul des $\boldsymbol{t}_i(k+1)$ a été expliqué en section 3.2.1. Ici aussi, les calculs sont généralement plus simples que ce que suggère l'algorithme, car les matrices de coefficients sont souvent creuses et beaucoup des coefficients non nuls sont des 1.

Algorithme 6 : SIF

foreach k **do**
  for $i \in \{1, \dots, n_t\}$ **do**
    $\boldsymbol{t}_i(k+1) \leftarrow \sum_{j=1}^{i-1}(-\boldsymbol{J}_{ij}\boldsymbol{t}_j(k+1)) + \sum_{j=1}^{n_x}\boldsymbol{M}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u}\boldsymbol{N}_{ij}\boldsymbol{u}_j(k)$
  end
  for $i \in \{1, \dots, n_x\}$ **do**
    $\boldsymbol{x}_i(k+1) \leftarrow \sum_{j=1}^{n_t}\boldsymbol{K}_{ij}\boldsymbol{t}_j(k) + \sum_{j=1}^{n_x}\boldsymbol{P}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u}\boldsymbol{Q}_{ij}\boldsymbol{u}_j(k)$
  end
  for $i \in \{1, \dots, n_y\}$ **do**
    $\boldsymbol{y}_i(k) \leftarrow \sum_{j=1}^{n_t}\boldsymbol{L}_{ij}\boldsymbol{t}_j(k) + \sum_{j=1}^{n_x}\boldsymbol{R}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u}\boldsymbol{S}_{ij}\boldsymbol{u}_j(k)$
  end
end

Filtre implémenté. Intéressons-nous maintenant au filtre implémenté associé à une SIF pour un format de précision finie $\mathbb{F}$ (dont on suppose qu'il contient 0 et -1, ce qui est généralement le cas) et un arrondi $\bigcirc$ vers ce format. Ceci est similaire au filtre implémenté d'un *State-Space* en section 3.1.2 et on utilise les mêmes notations. La seule difficulté supplémentaire est le calcul du vecteur intermédiaire $\boldsymbol{t}^*(k+1)$ du filtre implémenté $\mathscr{H}^*$, puisque ses composantes peuvent dépendre les unes des autres.

Supposons pour l'instant que les coefficients des matrices $\boldsymbol{J}$, $\boldsymbol{M}$ et $\boldsymbol{N}$ sont dans le format de précision finie $\mathbb{F}$. On voit dans l'algorithme 6 qu'on peut caractériser $\boldsymbol{t}_i^*(k+1)$ ainsi :

$$\boldsymbol{t}_i^*(k+1) = \left(-\boldsymbol{J}_{i,1} \quad \cdots \quad -\boldsymbol{J}_{i,i-1} \;\middle|\; \mathrm{row}_i(\boldsymbol{M}) \;\middle|\; \mathrm{row}_i(\boldsymbol{N})\right) \odot \left(\begin{array}{c} \boldsymbol{t}_1^*(k+1) \\ \vdots \\ \vdots \\ \boldsymbol{t}_{i-1}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array}\right) \tag{3.29}$$

où $\mathrm{row}_i(\boldsymbol{M})$ est la i-ième ligne de $\boldsymbol{M}$. Mais cette relation ne s'étend pas très bien au vecteur $\boldsymbol{t}^*(k+1)$ en entier, vu que le nombre de termes dépend de i. Cependant, considérons que lorsqu'on utilise le symbole $\odot$ de produit matriciel en précision finie, on ignore les termes faisant intervenir un coefficient constant nul. On peut alors écrire, afin de faire apparaître des matrices de la même taille pour tous les indices i :

$$\boldsymbol{t}_i^*(k+1) = \left(-\boldsymbol{J}_{i,1} \quad \cdots \quad -\boldsymbol{J}_{i,i-1} \quad 0 \quad \cdots \quad 0 \;\middle|\; \mathrm{row}_i(\boldsymbol{M}) \;\middle|\; \mathrm{row}_i(\boldsymbol{N})\right) \odot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array}\right) \tag{3.30}$$

De plus, la matrice $\boldsymbol{J}$ vérifie la propriété *Jprop*, c'est-à-dire qu'elle est triangulaire supérieure avec des 1 sur la diagonale. On sait donc que $\boldsymbol{J}_{i,i} = 1$ et $\boldsymbol{J}_{i,j} = 0$ pour $j > i$. On en déduit que la matrice $\left(-\boldsymbol{J}_{i,1} \quad \cdots \quad -\boldsymbol{J}_{i,i-1} \quad 0 \quad \cdots \quad 0\right)$ est

la i-ième ligne de la matrice $\boldsymbol{I} - \boldsymbol{J}$. On a donc :

$$\boldsymbol{t}_i^*(k+1) = \left(\mathrm{row}_i(\boldsymbol{I} - \boldsymbol{J}) \;\middle|\; \mathrm{row}_i(\boldsymbol{M}) \;\middle|\; \mathrm{row}_i(\boldsymbol{N}) \right) \odot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array} \right) \tag{3.31}$$

On en déduit enfin une relation pour tout le vecteur $\boldsymbol{t}^*(k+1)$:

$$\boldsymbol{t}^*(k+1) = \left(\boldsymbol{I} - \boldsymbol{J} \;\middle|\; \boldsymbol{M} \;\middle|\; \boldsymbol{N} \right) \odot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array} \right) \tag{3.32}$$

On avait supposé que les coefficients des matrices de $\boldsymbol{J}$, $\boldsymbol{M}$ et $\boldsymbol{N}$ sont dans le format de précision finie. Enlevons maintenant cette hypothèse. Comme en section 3.1.2 pour le *State-Space*, il faut arrondir chaque matrice. Encore une fois, le cas de $\boldsymbol{J}$ est particulier. Ce ne sont pas directement les coefficients de $\boldsymbol{J}$ qui sont utilisés, mais leurs opposés. Arrondir directement ces opposés a donc plus de sens, et facilitera l'analyse des erreurs d'arrondi en section 4.2. De plus, comme on a supposé que 0 et -1 sont dans le format de précision finie, et vu que la diagonale de $\boldsymbol{J}$ ne contient que des 1, les matrices $\boldsymbol{I} + \bigcirc(-\boldsymbol{J})$ et $\bigcirc(\boldsymbol{I} - \boldsymbol{J})$ sont égales.

Voici donc la relation définissant le filtre implémenté correspondant à une SIF (les autres matrices de coefficients et les calculs de $\boldsymbol{x}^*(k+1)$ et $\boldsymbol{y}^*(k)$ étant gérés exactement comme en section 3.1.2) :

$$\mathscr{H}^* \quad \left\{ \begin{array}{rcl} \boldsymbol{t}^*(k+1) & = & \left(\bigcirc(\boldsymbol{I} - \boldsymbol{J}) \;\middle|\; \bigcirc(\boldsymbol{M}) \;\middle|\; \bigcirc(\boldsymbol{N}) \right) \odot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array} \right) \\ \boldsymbol{x}^*(k+1) & = & \left(\bigcirc(\boldsymbol{K}) \;\middle|\; \bigcirc(\boldsymbol{P}) \;\middle|\; \bigcirc(\boldsymbol{Q}) \right) \odot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array} \right) \\ \boldsymbol{y}^*(k) & = & \left(\bigcirc(\boldsymbol{L}) \;\middle|\; \bigcirc(\boldsymbol{R}) \;\middle|\; \bigcirc(\boldsymbol{S}) \right) \odot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array} \right) \end{array} \right. \tag{3.33}$$

3.2.3 Définition Coq

On a vu que la condition *Jprop* (définition 11) implique que $\boldsymbol{J}$ est inversible. Afin de pouvoir travailler avec son inverse, on demande désormais à l'anneau commutatif `RT` d'être aussi unitaire.

```
Context {RT : comUnitRingType}.
```

Condition *Jprop*. On commence par définir la propriété *Jprop*.

```
Definition Jprop {n : nat} (A : 'M_n) :=
  (forall i : 'I_n, A i i = (1 : RT)) ∧
  (forall i j : 'I_n, (i < j)%nat → A i j = 0).
```

Comme on a vu en section 3.2.1, si $\boldsymbol{J}$ vérifie *Jprop* alors on peut facilement résoudre $\boldsymbol{J}\boldsymbol{v} = \boldsymbol{w}$ en $\boldsymbol{v}$. Cela ne demande même pas de division ou d'inverse. On

définit cette procédure à l'aide d'une récurrence forte sur `nat`, similaire à celle sur `int` présentée en section 2.1.5. Cette construction est définie pour n'importe quelle $\boldsymbol{J}$, mais renvoie bien le $\boldsymbol{v}$ attendu lorsque $\boldsymbol{J}$ vérifie *Jprop*, comme on prouve dans le lemme `solve_J_mult_correct`. On en déduit que cela revient à multiplier par l'inverse de $\boldsymbol{J}$: par la suite, on utilisera plutôt le corollaire `solve_J_mult_invmx`, qui est plus simple à manipuler grâce aux lemmes préexistants sur l'inverse d'une matrice. La notation `v _[i]` ou `A _[i, j]` représente le coefficient en position `i` ou $(\texttt{i}, \texttt{j})$ du vecteur `v` ou de la matrice `A` lorsque `i` et `j` sont des entiers de type `nat` (plus de détails en section 6.2.2).

```
Definition solve_J_mult {n : nat}
                        (J : 'M[RT]_n) (w : 'cV_n) : 'cV_n :=
  \matrix_(i,_) @strongrec RT 0 (fun i v_before_i ⇒
    w _[i] - \sum_(0 ≤ k < i) J _[i, k] * v_before_i k)   i.

Lemma solve_J_mult_correct (J : 'M_n) (w : 'cV_n) :
  Jprop J → J *m solve_J_mult J w = w.

Corollary solve_J_mult_invmx {n : nat} (J : 'M[RT]_n) (w : 'cV_n) :
  Jprop J → solve_J_mult J w = (invmx J) *m w.
```

Définition d'une SIF. Comme pour le *State-Space* en section 3.1.3, une SIF en Coq est donnée par ses neuf matrices de coefficients, de tailles appropriées. On demande aussi une preuve que $\boldsymbol{J}$ vérifie la condition *Jprop*.

```
Context {nu ny nx nt : nat}.
Record SIF := { SIF_J     : 'M[RT]_nt       ;
                SIF_Jprop : Jprop SIF_J     ;
                SIF_K     : 'M[RT]_(nx, nt) ;
                SIF_L     : 'M[RT]_(ny, nt) ;
                SIF_M     : 'M[RT]_(nt, nx) ;
                SIF_N     : 'M[RT]_(nt, nu) ;
                SIF_P     : 'M[RT]_nx       ;
                SIF_Q     : 'M[RT]_(nx, nu) ;
                SIF_R     : 'M[RT]_(ny, nx) ;
                SIF_S     : 'M[RT]_(ny, nu) ;
              }.
```

Filtre modèle correspondant. Étant donnés une SIF `sif` et un signal d'entrée `u`, on construit le vecteur d'état `x` récursivement, puis la sortie `y`, comme pour le *State-Space* en section 3.1.3. La différence est que les deux dépendent aussi du terme $\boldsymbol{t}(k+1)$. Encore une fois, quand on ferme la section `Build_output`, `y` devient de type `SIF → signal → signal` : c'est la fonction `filter_from_SIF` qu'on voulait construire.

```
Section Build_output.

Variable sif : SIF.
Notation J := (SIF_J sif). (* idem pour K, L, M, N, P, Q, R, S *)
Variable u : vsignal nu.
```

```
Notation compute_tS uk xk := ((invmx J) *m (M *m xk + N *m uk)).

Definition x := signal_peanorec
  (fun k xk ⇒ K *m compute_tS (u k) xk + P *m xk + Q *m u k).

Fact y_causal :
  causal (fun k ⇒
    L *m compute_tS (u k) (x k) + R *m x k + S *m u k).
Definition y := Build_signal y_causal.

End Build_output.

Definition filter_from_SIF : SIF → filter := y.
```

Un filtre implémenté est caractérisé par deux éléments principaux : la structure des calculs (choix, ordre et parenthésage des opérations) et l'arithmétique en précision finie utilisée (format et arrondi pour chaque variable et opération). À travers la SIF, on a déjà formalisé la structure des calculs. On pourrait aussi définir le filtre implémenté associé à une SIF étant donnés des formats et arrondis : il suffirait d'ajouter des annotations à chaque calcul, mais ce n'est pas très intéressant dans le cadre de ma formalisation. En effet, ce qui nous intéresse est l'erreur finale $\boldsymbol{\Delta y} = \boldsymbol{y}^* - \boldsymbol{y}$, et on verra en section 4.2.4 qu'on préfère l'exprimer sans passer par le filtre implémenté de toute façon.

3.2.4 D'un filtre implémenté donné à une SIF : exemple

On a vu que l'intérêt de la SIF est que n'importe quelle réalisation de filtre LTI peut être représentée sous forme de SIF, donc bénéficier de l'analyse des erreurs d'arrondi dans la SIF (qui sera l'objet du chapitre 4). Cette section explique comment s'y prendre en pratique pour traduire une réalisation donnée en une SIF, et illustre ce procédé sur un exemple. Cette traduction doit bien sûr conserver le filtre implémenté, afin que le comportement des erreurs d'arrondi soit le même dans la réalisation initiale et dans la SIF qui sera analysée. Un algorithme de traduction similaire est détaillé et implémenté en Python par Volkova [108].

Voici comment traduire une réalisation donnée vers une SIF décrivant le même filtre implémenté :

- Mettre cette réalisation sous forme de graphe de flot de données (section 2.2.5).
- Associer une variable d'état $\boldsymbol{x}_i(k)$ à chaque délai unitaire du graphe ; l'ordre n'a pas d'importance.
- Associer une variable intermédiaire $\boldsymbol{t}_i(k+1)$ à chaque fil du graphe qui en a besoin : résultat d'une addition ou d'une multiplication par une constante qui doit encore se faire additionner ou multiplier (structuration des calculs), ou qui est utilisé plusieurs fois (réutilisation des calculs). Il faut les ordonner de manière à ce que $\boldsymbol{t}_i(k+1)$ dépende éventuellement des $\boldsymbol{t}_j(k+1)$ avec $j < i$, mais jamais de ceux avec $j \geq i$. C'est toujours

possible vu qu'un graphe correct d'une réalisation n'aura pas de cycle de dépendance.

— Exprimer chaque $\boldsymbol{t}_i(k+1)$, $\boldsymbol{x}_i(k+1)$ et $\boldsymbol{y}_i(k)$ en fonction des $\boldsymbol{t}_i(k+1)$, $\boldsymbol{x}_i(k)$ et $\boldsymbol{u}_i(k)$. En comparant avec l'équation (3.25), en déduire les coefficients des matrices $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$. Si les $\boldsymbol{t}_i(k+1)$ ont été correctement ordonnés, la matrice $\boldsymbol{J}$ vérifiera bien la propriété *Jprop*.

On remarque que les deux premières étapes sont identiques à la traduction d'un filtre vers le *State-Space* (section 3.1.5). En effet, on a vu dans la section 3.2.1 que la SIF garde le même vecteur d'état $\boldsymbol{x}(k)$ que le *State-Space*. La troisième étape est bien sûr nouvelle. La quatrième est similaire au cas du *State-Space*, mais avec beaucoup plus de variables et donc de coefficients à déterminer.

On peut exprimer le même filtre implémenté avec beaucoup de SIF différentes. Par exemple, si on permute les coefficients de $\boldsymbol{x}(k)$, ou même certains des coefficients de $\boldsymbol{t}(k+1)$ dans la mesure où les dépendances le permettent, on obtient d'autres SIF qui décrivent le même filtre implémenté. Il s'agit donc simplement de construire une SIF convenable parmi d'autres.

Illustrons cette traduction vers la SIF sur notre exemple jouet de filtre à deux entrées et deux sorties. Ce filtre est défini par l'équation (1.20), à laquelle correspond l'algorithme suivant :

Algorithme 7 : Exemple de filtre implémenté à traduire vers une SIF

foreach k **do**
$\quad \boldsymbol{y}_1(k) \leftarrow 2\,(\,3\,\boldsymbol{u}_1(k) + \boldsymbol{u}_2(k-2)\,) - \frac{1}{2}\,\boldsymbol{y}_1(k-1)$
$\quad \boldsymbol{y}_2(k) \leftarrow 3\,(\,-4\,(\,7\,\boldsymbol{u}_2(k) + \boldsymbol{u}_2(k-1)\,) - \frac{1}{2}\,\boldsymbol{y}_1(k-1) + \frac{1}{5}\,\boldsymbol{y}_2(k-1)\,)$
end

Les calculs doivent être effectués en utilisant des sommes de produits autant que possible, mais sans développer les expressions.

Les deux premières étapes, le graphe de flot de données et le vecteur d'état $\boldsymbol{x}(k)$, sont les mêmes que pour le *State-Space* : on peut les reprendre directement de la section 3.1.5, où on avait traduit le même exemple vers un *State-Space*. Le graphe est à nouveau donné dans la figure 3.5, sur laquelle sont indiqués les $\boldsymbol{x}_i$, mais aussi les $\boldsymbol{t}_i$ choisis plus bas. On rappelle que le signal d'état qui avait été choisi est défini par :

$$\boldsymbol{x}(k+1) = \begin{pmatrix} \boldsymbol{u}_2(k) \\ \boldsymbol{u}_2(k-1) \\ \boldsymbol{y}_1(k) \\ \boldsymbol{y}_2(k) \end{pmatrix} \qquad \text{(rappel de (3.9))}$$

Ensuite, on veut déterminer les variables intermédiaires $\boldsymbol{t}_i(k+1)$. On a vu dans la section 3.2.1 que pour le sous-filtre avec une seule sortie $\boldsymbol{y}_1(k)$, on veut deux variables intermédiaires :

$$\begin{aligned} \boldsymbol{t}_1(k+1) &= 3\,\boldsymbol{u}_1(k) + \boldsymbol{u}_2(k-2) \\ \boldsymbol{t}_2(k+1) &= \boldsymbol{y}_1(k) \end{aligned} \qquad \text{(rappel de (3.20))}$$

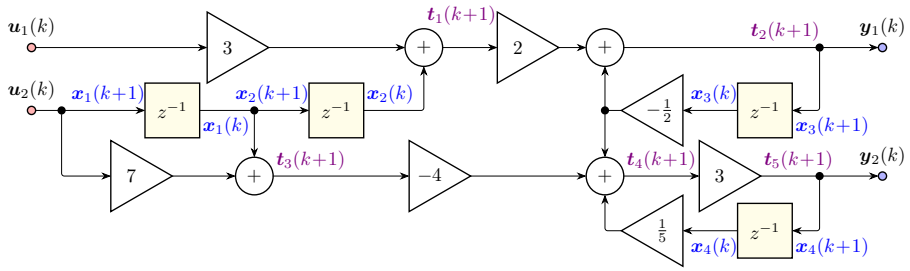


FIGURE 3.5 – Graphe de flot de données du second exemple jouet avec les $\boldsymbol{x}_i$ et $\boldsymbol{t}_i$ pour la SIF

La première sert à éviter de développer le facteur 2 dans le calcul de $\boldsymbol{y}_1(k)$, et la seconde à faire une seule fois le calcul de $\boldsymbol{y}_1(k)$ et $\boldsymbol{x}_3(k+1)$ qui sont égaux.

Dans le calcul de $\boldsymbol{y}_2(k)$, afin de ne pas développer les expressions, on veut stocker deux autres résultats intermédiaires :

$$\begin{aligned}
\boldsymbol{t}_3(k+1) &= 7\,\boldsymbol{u}_2(k) + \boldsymbol{u}_2(k-1) \\
\boldsymbol{t}_4(k+1) &= -4\,(\,7\,\boldsymbol{u}_2(k) + \boldsymbol{u}_2(k-1)\,) - \frac{1}{2}\,\boldsymbol{y}_1(k-1) + \frac{1}{5}\,\boldsymbol{y}_2(k-1) \\
&= -4\,\boldsymbol{t}_3(k+1) - \frac{1}{2}\,\boldsymbol{y}_1(k-1) + \frac{1}{5}\,\boldsymbol{y}_2(k-1)
\end{aligned} \tag{3.34}$$

On remarque sur le graphe que les valeurs $-\frac{1}{2}\,\boldsymbol{x}_3(k)$ et $3\,\boldsymbol{t}_4(k+1)$ sont utilisées deux fois chacune. Cependant, on ne peut pas mettre en commun le calcul de $-\frac{1}{2}\,\boldsymbol{x}_3(k)$ dans une variable intermédiaire sans modifier la structure des calculs. En effet, ce calcul n'est jamais effectué séparément ; à la place, $-\frac{1}{2}\,\boldsymbol{x}_3(k)$ apparaît dans des sommes de produits qui sont censées être calculées en une seule opération : $2\,\boldsymbol{t}_1(k+1) - \frac{1}{2}\,\boldsymbol{x}_3(k)$ et $-4\,\boldsymbol{t}_3(k+1) - \frac{1}{2}\,\boldsymbol{x}_3(k) - \frac{1}{5}\,\boldsymbol{x}_4(k)$. En revanche, calculer $3\,\boldsymbol{t}_4(k+1)$ dans une variable intermédiaire pour utiliser le résultat plusieurs fois ne pose pas de problème. On définit donc :

$$\boldsymbol{t}_5(k+1) = 3\,\boldsymbol{t}_4(k+1) \tag{3.35}$$

On doit maintenant exprimer chaque composante de $\boldsymbol{t}(k+1)$ en fonction de seulement les $\boldsymbol{t}_i(k+1)$ d'indice plus petit, les $\boldsymbol{x}_i(k)$ et les $\boldsymbol{u}_i(k)$:

$$\begin{aligned}
\boldsymbol{t}_1(k+1) &= 3\,\boldsymbol{u}_1(k) + \boldsymbol{u}_2(k-2) = 3\,\boldsymbol{u}_1(k) + \boldsymbol{x}_2(k) \\
\boldsymbol{t}_2(k+1) &= \boldsymbol{y}_1(k) = 2\,\boldsymbol{t}_1(k+1) - \frac{1}{2}\,\boldsymbol{x}_3(k) \\
\boldsymbol{t}_3(k+1) &= 7\,\boldsymbol{u}_2(k) + \boldsymbol{x}_1(k) \\
\boldsymbol{t}_4(k+1) &= -4\,\boldsymbol{t}_3(k+1) - \frac{1}{2}\,\boldsymbol{x}_3(k) + \frac{1}{5}\,\boldsymbol{x}_4(k) \\
\boldsymbol{t}_5(k+1) &= 3\,\boldsymbol{t}_4(k+1)
\end{aligned} \tag{3.36}$$

On en déduit les matrices de coefficients $\boldsymbol{J}$, $\boldsymbol{M}$ et $\boldsymbol{N}$ (la dépendance de $\boldsymbol{t}_i(k+1)$ en $\boldsymbol{t}_j(k+1)$ pour $j < i$ étant gérée par la matrice $\boldsymbol{J}$ comme expliqué en sec-

tion 3.2.1) :

$$\underbrace{\begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ & \ddots & \ddots & & \vdots \\ -2 & & \ddots & \ddots & \vdots \\ 0 & 0 & & \ddots & 0 \\ 0 & 0 & 4 & -3 & 1 \end{pmatrix}}_{\boldsymbol{J}} \boldsymbol{t}(k+1) = \underbrace{\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{1}{2} & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{2} & \frac{1}{5} \\ 0 & 0 & 0 & 0 \end{pmatrix}}_{\boldsymbol{M}} \boldsymbol{x}(k) + \underbrace{\begin{pmatrix} 3 & 0 \\ 0 & 0 \\ 0 & 7 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}}_{\boldsymbol{N}} \boldsymbol{u}(k) \tag{3.37}$$

Ensuite, on exprime $\boldsymbol{x}(k+1)$ en fonction de $\boldsymbol{t}(k+1)$, $\boldsymbol{x}(k)$ et $\boldsymbol{u}(k)$. Notons que ses composantes ont les mêmes valeurs que dans le *State-Space* de la section 3.1.5, mais ne sont pas calculées de la même manière.

$$\begin{aligned} \boldsymbol{x}_1(k+1) &= \boldsymbol{u}_2(k) \\ \boldsymbol{x}_2(k+1) &= \boldsymbol{u}_2(k-1) = \boldsymbol{x}_1(k) \\ \boldsymbol{x}_3(k+1) &= \boldsymbol{y}_1(k) = \boldsymbol{t}_2(k+1) \\ \boldsymbol{x}_4(k+1) &= \boldsymbol{y}_2(k) = \boldsymbol{t}_5(k+1) \end{aligned} \tag{3.38}$$

$$\boldsymbol{x}(k+1) = \underbrace{\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}}_{\boldsymbol{K}} \boldsymbol{t}(k+1) + \underbrace{\begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}}_{\boldsymbol{P}} \boldsymbol{x}(k) + \underbrace{\begin{pmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}}_{\boldsymbol{Q}} \boldsymbol{u}(k) \tag{3.39}$$

Enfin, on fait la même chose pour $\boldsymbol{y}(k)$. Comme $\boldsymbol{y}(k)$ est égal aux deux dernières lignes de $\boldsymbol{x}(k+1)$, les matrices $\boldsymbol{L}$, $\boldsymbol{R}$ et $\boldsymbol{S}$ obtenues sont les deux dernières lignes de $\boldsymbol{K}$, $\boldsymbol{P}$ et $\boldsymbol{Q}$ respectivement.

$$\begin{aligned} \boldsymbol{y}_1(k) &= \boldsymbol{t}_2(k+1) \\ \boldsymbol{y}_2(k) &= \boldsymbol{t}_5(k+1) \end{aligned} \tag{3.40}$$

$$\boldsymbol{y}(k) = \underbrace{\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}}_{\boldsymbol{L}} \boldsymbol{t}(k+1) + \underbrace{\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}}_{\boldsymbol{R}} \boldsymbol{x}(k) + \underbrace{\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}}_{\boldsymbol{S}} \boldsymbol{u}(k) \tag{3.41}$$

On a ainsi construit une SIF qui décrit le filtre implémenté défini par l'équation (1.20), également représenté par la figure 3.5. Voici cette SIF écrite sous la forme d'une seule matrice par blocs :

$$\boldsymbol{Z} = \left(\begin{array}{ccccc|cccc|cc} -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 3 & 0 \\ 2 & -1 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 7 \\ 0 & 0 & -4 & -1 & 0 & 0 & 0 & -\frac{1}{2} & \frac{1}{5} & 0 & 0 \\ 0 & 0 & 0 & 3 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) \tag{3.42}$$

Comme mentionné en section 3.2.2, on remarque que cette matrice est très creuse, et que beaucoup des coefficients non nuls sont des 1.

Voici l'algorithme correspondant à cette SIF. Il calcule le même signal de sortie que l'algorithme 7, même en précision finie.

Algorithme 8 : SIF construite pour l'exemple

foreach k **do**
$\boldsymbol{t}_1(k+1) \leftarrow \boldsymbol{x}_2(k) + 3\,\boldsymbol{u}_1(k)$
$\boldsymbol{t}_2(k+1) \leftarrow 2\,\boldsymbol{t}_1(k+1) - \frac{1}{2}\,\boldsymbol{x}_3(k)$
$\boldsymbol{t}_3(k+1) \leftarrow \boldsymbol{x}_1(k) + 7\,\boldsymbol{u}_2(k)$
$\boldsymbol{t}_4(k+1) \leftarrow -4\,\boldsymbol{t}_3(k+1) - \frac{1}{2}\,\boldsymbol{x}_3(k) + \frac{1}{5}\,\boldsymbol{x}_4(k)$
$\boldsymbol{t}_5(k+1) \leftarrow 3\,\boldsymbol{t}_4(k+1)$
$\boldsymbol{x}_1(k+1) \leftarrow \boldsymbol{u}_2(k)$
$\boldsymbol{x}_2(k+1) \leftarrow \boldsymbol{x}_1(k)$
$\boldsymbol{x}_3(k+1) \leftarrow \boldsymbol{t}_2(k+1)$
$\boldsymbol{x}_4(k+1) \leftarrow \boldsymbol{t}_5(k+1)$
$\boldsymbol{y}_1(k) \leftarrow \boldsymbol{t}_2(k+1)$
$\boldsymbol{y}_2(k) \leftarrow \boldsymbol{t}_5(k+1)$
end

3.2.5 Traductions depuis et vers le *State-Space*

Cette section explique comment passer d'un *State-Space* à une SIF et réciproquement. La traduction des autres réalisations définies précédemment vers une SIF fera l'objet de la section 3.3. Celle vers le *State-Space* est présentée ici car elle est rendue très simple par le fait que la SIF soit une extension du *State-Space*. Et exceptionnellement, on s'intéresse aussi à la traduction réciproque, de la SIF vers le *State-Space* : elle permet à la SIF de récupérer des propriétés déjà connues du *State-Space*.

Du *State-Space* à la SIF. Soit $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ un *State-Space*. On a vu en section 3.2.1 que si on enlève tout ce qui concerne $\boldsymbol{t}$ de la définition de la SIF, on retombe sur un *State-Space*. Il suffit donc de choisir $n_t = 0$, ce qui signifie que les matrices $\boldsymbol{J}$, $\boldsymbol{M}$, $\boldsymbol{N}$, $\boldsymbol{K}$ et $\boldsymbol{L}$ ont au moins une dimension nulle, et de poser :

$$\begin{array}{ll} \boldsymbol{P} = \boldsymbol{A}, & \boldsymbol{Q} = \boldsymbol{B}, \\ \boldsymbol{R} = \boldsymbol{C}, & \boldsymbol{S} = \boldsymbol{D}. \end{array} \tag{3.43}$$

La matrice $\boldsymbol{Z}$ correspondante est :

$$\left(\begin{array}{c|c} \boldsymbol{A} & \boldsymbol{B} \\ \hline \boldsymbol{C} & \boldsymbol{D} \end{array}\right) \tag{3.44}$$

On voit facilement dans l'équation (3.25) définissant la SIF qu'on obtient exactement l'équation (3.3) définissant le *State-Space*. Les filtres modèle et implémenté sont donc bien sûr les mêmes.

Comme on expliquera en section 3.3, on prouve seulement la conservation du filtre modèle en Coq, puisque la notion de filtre implémenté n'a pas été définie. Pour les matrices qui ont une dimension nulle, on doit quand même donner une matrice explicite, qui n'a bien sûr pas d'importance. Pour $\boldsymbol{J}$, on

choisit une matrice identité car on a déjà prouvé que celle-ci vérifie toujours *Jprop* ; pour les autres, on utilise simplement une matrice nulle. La preuve ne pose pas de difficulté. On prouve que les vecteurs d'état $\boldsymbol{x}(k)$ sont les mêmes par récurrence, puis que les sorties sont les mêmes, pour le *State-Space* initial et la SIF construite.

```
Variable stsp : @StateSpace RT nu ny nx.
Notation A := (StSp_A stsp). (* idem pour B, C, D *)

Definition SIF_from_StateSpace : SIF :=
  @Build_SIF RT nu ny nx 0
    (* SIF_J *)       (1%:M : 'M[RT]_0)
    (* SIF_Jprop *)   Jprop_1
    (* SIF_K *)       0
    (* SIF_L *)       0
    (* SIF_M *)       0
    (* SIF_N *)       0
    (* SIF_P *)       A
    (* SIF_Q *)       B
    (* SIF_R *)       C
    (* SIF_S *)       D.

Theorem SIF_from_StateSpace_same_filter :
  filter_from_SIF (SIF_from_StateSpace stsp) =
  filter_from_StSp stsp.
```

De la SIF au *State-Space*. Soit $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ une SIF. Le *State-Space* suivant décrit le même filtre modèle (rappelons que la propriété *Jprop* implique que la matrice $\boldsymbol{J}$ est inversible et son inverse est facile à calculer) :

$$\begin{aligned} \boldsymbol{A} &= \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{P}, \quad \boldsymbol{B} = \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{Q}, \\ \boldsymbol{C} &= \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{R}, \quad \boldsymbol{D} = \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{S}. \end{aligned} \tag{3.45}$$

Cependant, le filtre implémenté n'est pas forcément conservé. Ce n'est pas surprenant, étant donné qu'un filtre LTI implémenté peut toujours être mis sous forme de SIF, mais pas toujours sous forme de *State-Space*.

La définition en Coq est immédiate. La preuve est similaire à celle de la traduction d'un *State-Space* vers une SIF vue précédemment.

```
Variable sif : SIF.
Notation J := (SIF_J sif). (* idem pour K, L, M, N, P, Q, R, S *)

Definition StateSpace_from_SIF : StateSpace :=
  Build_StateSpace
    (* StSp_A *) (K *m invmx J *m M + P)
    (* StSp_B *) (K *m invmx J *m N + Q)
    (* StSp_C *) (L *m invmx J *m M + R)
    (* StSp_D *) (L *m invmx J *m N + S).

Theorem StateSpace_from_SIF_same_filter :
  filter_from_StSp (StateSpace_from_SIF sif) = filter_from_SIF sif.
```

On peut ainsi appliquer à la SIF des propriétés connues sur le *State-Space*. Cela permet par exemple d'obtenir la réponse impulsionnelle et la fonction de transfert d'une SIF en passant par les expressions pour le *State-Space* vues en section 3.1.4. La réponse impulsionnelle en particulier sera utile en section 4.3.5. On en déduit aussi trivialement que le filtre défini par une SIF est LTI (même si cela ne serait pas difficile à montrer directement sur la SIF).

```
Corollary filter_from_SIF_is_LTI :
  LTI_filter (filter_from_SIF sif).
```

Cette traduction d'une SIF vers un *State-Space* permettra aussi de prouver très facilement des propriétés sur la transformation d'une SIF en d'autres SIF dans la section 3.4, à partir de propriétés similaires sur le *State-Space*.

3.3 Des réalisations usuelles vers la SIF

La section 3.2.4 explique comment traduire n'importe quelle réalisation de filtre LTI vers une SIF en conservant le filtre implémenté, afin de pouvoir profiter de l'analyse des erreurs d'arrondi dans la SIF du chapitre 4. La section courante explicite cette traduction vers la SIF pour des familles de réalisations usuelles présentées dans la section 2.3 : les formes directes I et II, la forme directe II transposée et les décompositions parallèle et en cascade.

Ces traductions sont formalisées en Coq et sont accompagnées de la preuve que le filtre modèle est bien conservé. Mais la conservation du filtre implémenté n'est pas exprimée en Coq, puisque le filtre implémenté n'est pas directement défini comme expliqué en section 3.2.3. Il serait pertinent de prouver la conservation du filtre implémenté si celui-ci était défini par une source extérieure à ma formalisation. Relier ma formalisation à d'autres formalisations existantes est une perspective à long terme discutée en section 6.3.3.

3.3.1 Forme directe I

Soit $(a_i)_{1\leq i\leq n}$ et $(b_i)_{0\leq i\leq n}$ des coefficients de fonction de transfert. Rappelons l'algorithme 9 définissant le filtre implémenté correspondant en utilisant la forme directe I, où le calcul doit être effectué avec une seule grande somme de produits. On veut construire une SIF décrivant le même filtre implémenté.

Algorithme 9 : Forme directe I (rappel de l'algorithme 3)

foreach k **do**
  $y(k) \leftarrow \sum_{i=0}^{n} b_i u(k-i) + \sum_{i=1}^{n} (-a_i) y(k-i)$
end

Ce filtre est SISO, mais la SIF définit des filtres MIMO. On va donc le représenter par un filtre MIMO avec $n_u = n_y = 1$. Comme en section 3.1.6, on note $u(k)$ l'entrée scalaire et $\boldsymbol{u}(k)$ est le vecteur de taille 1 correspondant, et de même pour la sortie :

$$\boldsymbol{u}(k) = \begin{pmatrix} u(k) \end{pmatrix} \qquad \boldsymbol{y}(k) = \begin{pmatrix} y(k) \end{pmatrix} \tag{3.46}$$

On voit sur le graphe de flot de données de la figure 2.4a qu'il y a $2n$ valeurs à retenir, qui sont les $u(k-i)$ et $y(k-i)$ pour $1 \leq i \leq n$. On pose donc :

$$\boldsymbol{x}(k) = \begin{pmatrix} u(k-n) \\ u(k-n+1) \\ \vdots \\ u(k-1) \\ y(k-n) \\ y(k-n+1) \\ \vdots \\ y(k-1) \end{pmatrix} \qquad (\text{donc } n_x = 2n) \tag{3.47}$$

L'ordre n'a pas d'importance, puisque les calculs des composantes de $\boldsymbol{x}(k+1)$ seront indépendants. Comme en section 3.1.6, on a choisi un ordre qui simplifie un petit peu les choses pour une mise à jour de $\boldsymbol{x}(k)$ vers $\boldsymbol{x}(k+1)$ en place, mais cela n'affecte pas la sortie du filtre implémenté.

Ici, on n'a pas besoin du vecteur intermédiaire $\boldsymbol{t}(k+1)$ pour représenter la structure des calculs, vu que l'algorithme 9 est déjà une somme de produits de coefficients constants par des valeurs disponibles soit dans $\boldsymbol{u}(k)$, soit dans $\boldsymbol{x}(k)$. En revanche, on remarque que $\boldsymbol{x}_{2n}(k+1) = y(k)$. Le résultat de cette somme de produits est donc utilisé deux fois. On décide alors de stocker ce résultat dans $\boldsymbol{t}_1(k+1)$ pour éviter une duplication inutile du calcul :

$$\boldsymbol{t}(k+1) = \Big(y(k)\Big) \qquad (\text{donc } n_t = 1) \tag{3.48}$$

Maintenant qu'on a déterminé le signal d'état $\boldsymbol{x}$ et le signal intermédiaire $\boldsymbol{t}$, il ne reste plus qu'à exprimer $\boldsymbol{t}(k+1)$, $\boldsymbol{x}(k+1)$ et $\boldsymbol{y}(k)$ en fonction de $\boldsymbol{t}(k+1)$, $\boldsymbol{x}(k)$ et $\boldsymbol{u}(k)$, puis comparer ces expressions avec l'équation (3.25) pour en déduire les matrices de la SIF.

Le calcul pour $\boldsymbol{t}(k+1)$ est :

$$\begin{aligned} \boldsymbol{t}_1(k+1) &= \sum_{i=0}^{n} b_i u(k-i) + \sum_{i=1}^{n} (-a_i) y(k-i) \\ &= b_0 u(k) + \sum_{i=1}^{n} b_i \boldsymbol{x}_{n+1-i}(k) + \sum_{i=1}^{n} (-a_i) \boldsymbol{x}_{2n+1-i}(k) \\ &= b_0 \boldsymbol{u}_1(k) + \sum_{i=1}^{n} b_{n+1-i}\, \boldsymbol{x}_i(k) + \sum_{i=n+1}^{2n} (-a_{2n+1-i}) \boldsymbol{x}_i(k) \\ \underbrace{\Big(1\Big)}_{\boldsymbol{J}} \boldsymbol{t}(k+1) &= \underbrace{\begin{pmatrix} b_n & \cdots & b_1 & -a_n & \cdots & -a_1 \end{pmatrix}}_{\boldsymbol{M}} \boldsymbol{x}(k) + \underbrace{\Big(b_0\Big)}_{\boldsymbol{N}} \boldsymbol{u}(k) \end{aligned} \tag{3.49}$$

Pour $\boldsymbol{x}(k+1)$, les anciennes valeurs de la forme $u(k-i)$ et $y(k-i)$ sont récupérées dans $\boldsymbol{x}(k)$. Les seules nouvelles valeurs sont $u(k) = \boldsymbol{u}_1(k)$ à laquelle

on a directement accès, et $y(k)$ qui a été calculée dans $\boldsymbol{t}_1(k+1)$.

$$\boldsymbol{x}(k+1) = \begin{pmatrix} u(k-n+1) \\ \vdots \\ u(k-1) \\ u(k) \\ y(k-n+1) \\ \vdots \\ y(k-1) \\ y(k) \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_2(k) \\ \vdots \\ \boldsymbol{x}_n(k) \\ \boldsymbol{u}_1(k) \\ \boldsymbol{x}_{n+2}(k) \\ \vdots \\ \boldsymbol{x}_{2n}(k) \\ \boldsymbol{t}_1(k+1) \end{pmatrix}$$

$$\boldsymbol{x}(k+1) = \underbrace{\begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}}_{\boldsymbol{K}} \boldsymbol{t}(k+1) + \underbrace{\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & & \vdots \\ & & 1 & \ddots & \\ & & & 0 & \\ & & & 1 & 0 \\ 0 & \cdots & \cdots & \cdots & 0 \end{pmatrix}}_{\boldsymbol{P}} \boldsymbol{x}(k) + \underbrace{\begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}}_{\boldsymbol{Q}} \boldsymbol{u}(k) \tag{3.50}$$

Enfin, comme on a déjà calculé $y(k)$ dans $\boldsymbol{t}_1(k+1)$, on a :

$$\boldsymbol{y}(k) = \underbrace{\begin{pmatrix} 1 \end{pmatrix}}_{\boldsymbol{L}} \boldsymbol{t}(k+1) + \underbrace{\begin{pmatrix} 0 & \cdots & \cdots & 0 \end{pmatrix}}_{\boldsymbol{R}} \boldsymbol{x}(k) + \underbrace{\begin{pmatrix} 0 \end{pmatrix}}_{\boldsymbol{S}} \boldsymbol{u}(k) \tag{3.51}$$

Sans surprise vu que $\boldsymbol{x}_{2n}(k+1) = y(k)$, les matrices $\boldsymbol{L}$, $\boldsymbol{R}$ et $\boldsymbol{S}$ sont identiques à la dernière ligne de $\boldsymbol{K}$, $\boldsymbol{P}$ et $\boldsymbol{Q}$ respectivement. Mais grâce au calcul intermédiaire dans $\boldsymbol{t}(k+1)$, ces matrices sont très simples. Sans celui-ci, $\boldsymbol{R}$ et la dernière ligne de $\boldsymbol{P}$ seraient notre matrice actuelle $\boldsymbol{M}$ au lieu d'être nulles.

La forme directe I est donc décrite par la SIF suivante :

$$\boldsymbol{Z} = \left(\begin{array}{c|ccccccc|c} -1 & b_n & \cdots & b_1 & -a_n & \cdots & -a_1 & b_0 \\ \hline 0 & 0 & 1 & 0 & \cdots & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & & & \vdots & \vdots \\ & \vdots & & 1 & \ddots & & & 0 \\ & \vdots & & & 0 & \ddots & & 1 \\ & \vdots & & & & 1 & \ddots & 0 \\ & \vdots & & & & & 0 & \vdots \\ 0 & \vdots & & & & & 1 & \vdots \\ 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 & 0 \\ \hline 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 & 0 \end{array}\right) \tag{3.52}$$

Encore une fois, on remarque que cette matrice est très creuse, et beaucoup des coefficients non nuls sont des 1.

Formalisation. Pour formaliser cette traduction vers la SIF, on se donne des coefficients et on définit la SIF correspondant à la matrice $\boldsymbol{Z}$ ci-dessus. Encore une fois, on fait attention aux indices de matrice qui commencent à 0 en Coq.

```
Variable tfc : TFCoeffs.
Notation n := (TFC_n tfc).
Notation a := (TFC_a tfc).
Notation b := (TFC_b tfc).

Definition SIF_from_DFI : SIF :=
@Build_SIF RT 1 1
(* nx *)          (2*n)
(* nt *)          1
(* SIF_J *)       1%:M
(* SIF_Jprop *)   (Jprop_1)
(* SIF_K *)       (\matrix_(i,_)
                    if ((i:nat) == 2*n-1)%nat then 1 else 0)
(* SIF_L *)       1%:M
(* SIF_M *)       (\matrix_(_,j) if ((j:nat) < n)%nat
                    then b (n-j)%nat else - a (2*n-j)%nat)
(* SIF_N *)       (b 0)%:M
(* SIF_P *)       (\matrix_(i,j)
                    if ((i:nat) != (n-1)%nat) && (i+1 == j)%nat
                    then 1 else 0)
(* SIF_Q *)       (\matrix_(i,_)
                    if (i:nat) == (n-1)%nat then 1 else 0)
(* SIF_R *)       0
(* SIF_S *)       0.
```

On prouve ensuite la conservation du filtre modèle. Supposer $n > 0$ facilite la preuve en assurant que le vecteur $\boldsymbol{x}(k)$ n'est pas de dimension nulle ; si $n = 0$, le filtre est de toute façon réduit à une multiplication par une constante.

```
Hypothesis n_gt0 : (n > 0)%nat.

Theorem SIF_from_DFI_same_filter :
to_SISO (filter_from_SIF SIF_from_DFI) = filter_from_TFC tfc.
```

La preuve est assez longue car on montre que chaque coefficient des signaux $\boldsymbol{t}$ et $\boldsymbol{x}$ joue le bon rôle, mais elle ne présente pas de difficulté particulière.

3.3.2 Forme directe II

On considère à nouveau des coefficients $(a_i)_{1\leq i\leq n}$ et $(b_i)_{0\leq i\leq n}$. On s'intéresse au filtre implémenté correspondant en utilisant la forme directe II, décrit par l'algorithme 10. Comme d'habitude, on représente le filtre SISO d'entrée u et sortie y par un filtre MIMO où $n_u = n_y = 1$, d'entrée $\boldsymbol{u}$ et sortie $\boldsymbol{y}$.

Algorithme 10 : Forme directe II (rappel de l'algorithme 4)

foreach k **do**

$$e(k) \leftarrow u(k) + \sum_{i=1}^{n}(-a_i)e(k-i)$$

$$y(k) \leftarrow \sum_{i=0}^{n} b_i e(k-i)$$

end

Le vecteur d'état est le même que celui du *State-Space*, qu'on a déjà déterminé dans la section 3.1.6 :

$$\boldsymbol{x}(k) = \begin{pmatrix} e(k-n) \\ \vdots \\ e(k-1) \end{pmatrix} \qquad (\text{donc } n_x = n) \tag{3.53}$$

On a vu dans la section 3.1.7 que l'algorithme de la forme directe II ne peut pas être décrit sous forme de *State-Space*, car celui-ci force le développement du terme $e(k)$ dans le calcul de $y(k)$. La SIF résout ce problème en calculant $e(k)$ dans $\boldsymbol{t}_1(k+1)$:

$$\boldsymbol{t}(k) = \Big(e(k)\Big) \qquad (\text{donc } n_t = 1) \tag{3.54}$$

D'ailleurs, ce résultat servira pour $y(k)$ mais aussi pour $\boldsymbol{x}_n(k+1) = e(k)$. Ici, une seule variable intermédiaire $\boldsymbol{t}_1(k+1)$ sert donc à la fois à structurer les calculs et à éviter de les dupliquer.

Maintenant que les signaux $\boldsymbol{x}$ et $\boldsymbol{t}$ ont été décidés, on peut déterminer les matrices de la SIF correspondante :

$$\begin{aligned}
\boldsymbol{t}_1(k+1) = e(k) &= u(k) + \sum_{i=1}^{n} (-a_i) e(k-i) \\
&= u(k) + \sum_{i=1}^{n} (-a_i) \boldsymbol{x}_{n+1-i}(k) \\
&= \boldsymbol{u}_1(k) + \sum_{i=1}^{n} (-a_{n+1-i}) \boldsymbol{x}_i(k)
\end{aligned} \tag{3.55}$$

$$\underbrace{\begin{pmatrix} 1 \end{pmatrix}}_{\boldsymbol{J}} \boldsymbol{t}(k+1) = \underbrace{\begin{pmatrix} -a_n & \cdots & -a_1 \end{pmatrix}}_{\boldsymbol{M}} \boldsymbol{x}(k) + \underbrace{\begin{pmatrix} 1 \end{pmatrix}}_{\boldsymbol{N}} \boldsymbol{u}(k)$$

$$\boldsymbol{x}(k+1) = \begin{pmatrix} e(k-n+1) \\ \vdots \\ e(k-1) \\ e(k) \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_2(k) \\ \vdots \\ \boldsymbol{x}_n(k) \\ \boldsymbol{t}_1(k+1) \end{pmatrix}$$

$$\boldsymbol{x}(k+1) = \underbrace{\begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}}_{\boldsymbol{K}} \boldsymbol{t}(k+1) + \underbrace{\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ & & \ddots & \ddots & 0 \\ & & & \ddots & 1 \\ 0 & \cdots & \cdots & 0 & 0 \end{pmatrix}}_{\boldsymbol{P}} \boldsymbol{x}(k) + \underbrace{\begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}}_{\boldsymbol{Q}} \boldsymbol{u}(k) \tag{3.56}$$

$$\begin{aligned}
y(k) &= \sum_{i=0}^{n} b_i e(k-i) = b_0\, \boldsymbol{t}_1(k+1) + \sum_{i=1}^{n} b_{n+1-i}\, \boldsymbol{x}_i(k) \\
\boldsymbol{y}(k) &= \underbrace{\begin{pmatrix} b_0 \end{pmatrix}}_{\boldsymbol{L}} \boldsymbol{t}(k+1) + \underbrace{\begin{pmatrix} b_n & \cdots & b_1 \end{pmatrix}}_{\boldsymbol{R}} \boldsymbol{x}(k) + \underbrace{\begin{pmatrix} 0 \end{pmatrix}}_{\boldsymbol{S}} \boldsymbol{u}(k)
\end{aligned} \tag{3.57}$$

La SIF obtenue est :

$$\boldsymbol{Z} = \left(\begin{array}{c|ccc|c} -1 & -a_n & \cdots & -a_1 & 1 \\ \hline 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \vdots & \ddots & 1 & \vdots \\ 1 & 0 & \cdots & 0 & 0 \\ \hline b_0 & b_n & \cdots & b_1 & 0 \end{array}\right) \tag{3.58}$$

Formalisation. La formalisation est similaire à celle de la traduction vers la forme directe I en section 3.3.1. La preuve est cependant plus rapide car le vecteur d'état est plus simple.

```
Variable tfc : TFCoeffs.
Notation n := (TFC_n tfc).
Notation a := (TFC_a tfc).
Notation b := (TFC_b tfc).

Definition SIF_from_DFII : SIF :=
@Build_SIF RT 1 1
(* nx *)          n
(* nt *)          1
(* SIF_J *)       1%:M
(* SIF_Jprop *)   Jprop_1
(* SIF_K *)       (\matrix_(i,_)
                     if (i:nat) == (n-1)%nat then 1 else 0)
(* SIF_L *)       (b 0)%:M
(* SIF_M *)       (\matrix_(_,j) - a (n-j)%nat)
(* SIF_N *)       1%:M
(* SIF_P *)       (\matrix_(i,j) if (i:int)+1 == j then 1 else 0)
(* SIF_Q *)       0
(* SIF_R *)       (\matrix_(_,j) b (n-j)%nat)
(* SIF_S *)       0.

Hypothesis n_gt0 : (n > 0)%nat.

Theorem SIF_from_DFII_same_filter :
to_SISO (filter_from_SIF SIF_from_DFII) = filter_by_DFII tfc.
```

3.3.3 Forme directe II transposée

Toujours à partir des mêmes coefficients, on considère la forme directe II transposée présentée dans la section 2.3.2, dont le graphe de flot de données a été donné en figure 2.5b. On utilise toujours les mêmes notations que dans les sections 3.3.1 et 3.3.2 pour l'entrée et la sortie vues comme des scalaires ou des vecteurs de tailles 1.

On assigne les $\boldsymbol{x}_i$ aux délais unitaires de ce graphe comme indiqué en figure 3.6. On remarque que le résultat de $b_0u(k) + \boldsymbol{x}_1(k)$ donne $y(k)$, mais va aussi être multiplié par les $(-a_i)$ respectifs pour obtenir les $\boldsymbol{x}_i(k+1)$. On calcule donc ce résultat intermédiaire dans $\boldsymbol{t}_1(k+1)$, comme illustré sur la figure 3.6. Cette même variable intermédiaire permet donc à la fois de structurer le calcul en évitant de développer la multiplication par $(-a_i)$ par l'addition, et de réutiliser le même résultat plusieurs fois.

On lit alors les calculs sur le graphe :

$$\begin{aligned} \boldsymbol{t}_1(k+1) &= b_0u(k) + \boldsymbol{x}_1(k) \\ \underbrace{\begin{pmatrix}1\end{pmatrix}}_{\boldsymbol{J}} \boldsymbol{t}(k+1) &= \underbrace{\begin{pmatrix}1 & 0 \cdots\cdots 0\end{pmatrix}}_{\boldsymbol{M}} \boldsymbol{x}(k) + \underbrace{\begin{pmatrix}b_0\end{pmatrix}}_{\boldsymbol{N}} \boldsymbol{u}(k) \end{aligned} \tag{3.59}$$

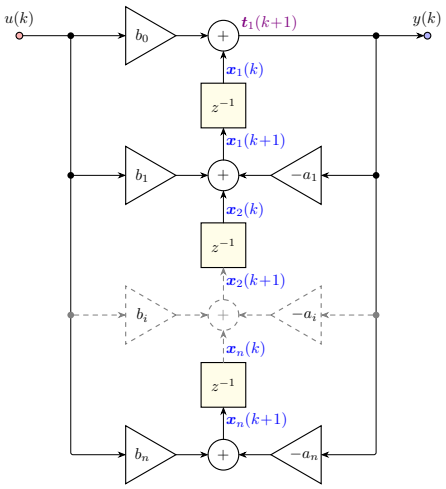


FIGURE 3.6 – Graphe de la forme directe II transposée avec les $\boldsymbol{x}_i$ et $\boldsymbol{t}_i$ choisis

$$\boldsymbol{x}(k+1) = \begin{pmatrix} -a_1\boldsymbol{t}_1(k+1) + \boldsymbol{x}_2(k) + b_1u(k) \\ \vdots \\ -a_{n-1}\boldsymbol{t}_1(k+1) + \boldsymbol{x}_n(k) + b_{n-1}u(k) \\ -a_n\boldsymbol{t}_1(k+1) + b_nu(k) \end{pmatrix}$$

$$\boldsymbol{x}(k+1) = \underbrace{\begin{pmatrix} -a_1 \\ \vdots \\ -a_{n-1} \\ -a_n \end{pmatrix}}_{\boldsymbol{K}} \boldsymbol{t}(k+1) + \underbrace{\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ & & & & 0 \\ & & & & 1 \\ 0 & \cdots & \cdots & \cdots & 0 \end{pmatrix}}_{\boldsymbol{P}} \boldsymbol{x}(k) + \underbrace{\begin{pmatrix} b_1 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix}}_{\boldsymbol{Q}} \boldsymbol{u}(k) \tag{3.60}$$

$$\begin{aligned} y(k) &= \boldsymbol{t}_1(k+1) \\ \boldsymbol{y}(k) &= \underbrace{\begin{pmatrix} 1 \end{pmatrix}}_{\boldsymbol{L}} \boldsymbol{t}(k+1) + \underbrace{\begin{pmatrix} 0 & \cdots & 0 \end{pmatrix}}_{\boldsymbol{R}} \boldsymbol{x}(k) + \underbrace{\begin{pmatrix} 0 \end{pmatrix}}_{\boldsymbol{S}} \boldsymbol{u}(k) \end{aligned} \tag{3.61}$$

La forme directe II transposée est donc décrite par la SIF suivante :

$$\boldsymbol{Z} = \left(\begin{array}{c|ccc|c} -1 & 1 & 0 \cdots & 0 & b_0 \\ \hline -a_1 & 0 & \ddots & 0 & b_1 \\ \vdots & & & \vdots & \vdots \\ & & & 0 & \\ -a_{n-1} & & & 1 & b_{n-1} \\ -a_n & 0 & \cdots & 0 & b_n \\ \hline 1 & 0 & \cdots & 0 & 0 \end{array}\right) \tag{3.62}$$

Formalisation. La formalisation est similaires aux sections précédentes. La forme directe II n'avait pas été définie en Coq. Cependant, le filtre modèle associé à la forme directe II est le même que celui obtenu à partir des mêmes coefficients

par la forme directe I, défini par `filter_from_TFC`. En effet, ces coefficients caractérisent la fonction de transfert du filtre comme expliqué en section 2.2.6. Encore une fois, seule la conservation du filtre modèle a été formalisée.

```
Variable tfc : TFCoeffs.
Notation n := (TFC_n tfc).
Notation a := (TFC_a tfc).
Notation b := (TFC_b tfc).

Definition SIF_from_DFIIt : SIF :=
@Build_SIF RT 1 1
(* nx *)          n
(* nt *)          1
(* SIF_J *)       1%:M
(* SIF_Jprop *)   Jprop_1
(* SIF_K *)       (\matrix_(i,_) - a (i+1)%nat)
(* SIF_L *)       1%:M
(* SIF_M *)       (\matrix_(_,j) if (j:nat) == 0 then 1 else 0)
(* SIF_N *)       (b 0)%:M
(* SIF_P *)       (\matrix_(i,j) if (i:int)+1 == j then 1 else 0)
(* SIF_Q *)       (\matrix_(i,_) b (i+1)%nat)
(* SIF_R *)       0
(* SIF_S *)       0.

Hypothesis n_gt0 : (n > 0)%nat.

Theorem SIF_from_DFIIt_same_filter :
  to_SISO (filter_from_SIF SIF_from_DFIIt) = filter_from_TFC tfc.
```

3.3.4 Décomposition parallèle

On veut traduire vers une SIF une réalisation donnée sous forme d'une décomposition parallèle en deux sous-réalisations (section 2.3.3). On suppose qu'on a traduit chaque sous-réalisation vers une SIF séparément, obtenant les SIF $(\boldsymbol{J_1}, \boldsymbol{K_1}, \boldsymbol{L_1}, \boldsymbol{M_1}, \boldsymbol{N_1}, \boldsymbol{P_1}, \boldsymbol{Q_1}, \boldsymbol{R_1}, \boldsymbol{S_1})$ et $(\boldsymbol{J_2}, \boldsymbol{K_2}, \boldsymbol{L_2}, \boldsymbol{M_2}, \boldsymbol{N_2}, \boldsymbol{P_2}, \boldsymbol{Q_2}, \boldsymbol{R_2}, \boldsymbol{S_2})$ respectivement. On note $\mathscr{H}_1$ et $\mathscr{H}_2$ les filtres modèles associés aux deux sous-réalisations, et $\mathscr{H}$ leur assemblage en parallèle. Encore une fois, on raisonne sur les filtres modèles pour alléger l'écriture, mais on s'interdit de modifier la structure des calculs donc les filtres implémentés correspondants.

Considérons un signal d'entrée $\boldsymbol{u}$. On note $\boldsymbol{y_1}$ la sortie correspondante de $\mathscr{H}_1$, $\boldsymbol{y_2}$ celle de $\mathscr{H}_2$, et $\boldsymbol{y}$ celle de $\mathscr{H}$. Attention : exceptionnellement, les indices représentent des signaux distincts plutôt que les composantes d'un signal vectoriel (c'est d'ailleurs pour cela que ces indices sont en gras). On a donc :

$$\boldsymbol{y} = \boldsymbol{y_1} + \boldsymbol{y_2} \tag{3.63}$$

Toutes ces sorties ont la même taille, notée n_y. Pour la même entrée $\boldsymbol{u}$, on note aussi $\boldsymbol{x_1}$ le signal d'état et $\boldsymbol{t_1}$ le signal intermédiaire de la première SIF, et $\boldsymbol{x_2}$ et $\boldsymbol{t_2}$ ceux de la seconde SIF, de tailles respectives n_{x1}, n_{t1}, n_{x2} et n_{t2}. Pour la

SIF qu'on va construire pour décrire $\mathscr{H}$, on garde les notations habituelles : $\boldsymbol{x}$ et $\boldsymbol{t}$, de tailles n_x et n_t.

À chaque temps k, pour obtenir la sortie $\boldsymbol{y}(k)$ de $\mathscr{H}$ ainsi que les valeurs dont on aura besoin au temps $k+1$, on calcule :

- $\boldsymbol{t_1}(k+1)$ à partir de $\boldsymbol{x_1}(k)$ et $\boldsymbol{u}(k)$
- $\boldsymbol{x_1}(k+1)$ et $\boldsymbol{y_1}(k)$ (indépendants l'un de l'autre) à partir de $\boldsymbol{t_1}(k+1)$, $\boldsymbol{x_1}(k)$ et $\boldsymbol{u}(k)$
- $\boldsymbol{t_2}(k+1)$ à partir de $\boldsymbol{x_2}(k)$ et $\boldsymbol{u}(k)$
- $\boldsymbol{x_2}(k+1)$ et $\boldsymbol{y_2}(k)$ (indépendants l'un de l'autre) à partir de $\boldsymbol{t_2}(k+1)$, $\boldsymbol{x_2}(k)$ et $\boldsymbol{u}(k)$
- $\boldsymbol{y}(k)$ à partir de $\boldsymbol{y_1}(k)$ et $\boldsymbol{y_2}(k)$

Les informations qu'on a besoin de conserver d'une étape sur l'autre sont à la fois $\boldsymbol{x_1}(k)$ et $\boldsymbol{x_2}(k)$. On pose donc :

$$\boldsymbol{x} = \left(\begin{array}{c} \boldsymbol{x_1} \\ \hline \boldsymbol{x_2} \end{array}\right) \qquad (\text{donc } n_x = n_{x1} + n_{x2}) \tag{3.64}$$

Les valeurs intermédiaires à calculer incluent naturellement les coefficients de $\boldsymbol{t_1}(k+1)$ et $\boldsymbol{t_2}(k+1)$. Mais on remarque aussi que les vecteurs $\boldsymbol{y_1(k)}$ et $\boldsymbol{y_1(k)}$ sont calculés, puis utilisés pour $\boldsymbol{y}(k)$: ils doivent donc également être inclus dans $\boldsymbol{t}(k+1)$. On pose alors :

$$\boldsymbol{t}(k+1) = \left(\begin{array}{c} \boldsymbol{t_1}(k+1) \\ \hline \boldsymbol{t_2}(k+1) \\ \hline \boldsymbol{y_1}(k) \\ \hline \boldsymbol{y_2}(k) \end{array}\right) \qquad (\text{donc } n_t = n_{t1} + n_{t2} + 2n_y) \tag{3.65}$$

On aurait pu par exemple échanger les positions de $\boldsymbol{t_2}(k+1)$ et $\boldsymbol{y_1}(k)$; il faut juste que $\boldsymbol{t_1}(k+1)$ soit au-dessus de $\boldsymbol{y_1}(k)$ et que $\boldsymbol{t_2}(k+1)$ soit au-dessus de $\boldsymbol{y_2}(k)$.

Les calculs sont ensuite :

$$\begin{aligned}
\boldsymbol{J_1 t_1}(k+1) &= \boldsymbol{M_1 x_1}(k) + \boldsymbol{N_1 u}(k) \\
\boldsymbol{J_2 t_2}(k+1) &= \boldsymbol{M_2 x_2}(k) + \boldsymbol{N_2 u}(k) \\
\boldsymbol{y_1}(k) &= \boldsymbol{L_1 t_1}(k+1) + \boldsymbol{R_1 x_1}(k) + \boldsymbol{S_1 u}(k) \\
\boldsymbol{y_2}(k) &= \boldsymbol{L_2 t_2}(k+1) + \boldsymbol{R_2 x_2}(k) + \boldsymbol{S_2 u}(k)
\end{aligned}$$

$$\underbrace{\left(\begin{array}{c|c|c|c} \boldsymbol{J_1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{J_2} & \boldsymbol{0} & \boldsymbol{0} \\ \hline -\boldsymbol{L_1} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} \\ \hline \boldsymbol{0} & -\boldsymbol{L_2} & \boldsymbol{0} & \boldsymbol{I} \end{array}\right)}_{\boldsymbol{J}} \boldsymbol{t}(k+1) = \underbrace{\left(\begin{array}{c|c} \boldsymbol{M_1} & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{M_2} \\ \hline \boldsymbol{R_1} & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{R_2} \end{array}\right)}_{\boldsymbol{M}} \boldsymbol{x}(k) + \underbrace{\left(\begin{array}{c} \boldsymbol{N_1} \\ \hline \boldsymbol{N_2} \\ \hline \boldsymbol{S_1} \\ \hline \boldsymbol{S_2} \end{array}\right)}_{\boldsymbol{N}} \boldsymbol{u}(k) \tag{3.66}$$

On remarque que comme $\boldsymbol{J_1}$ et $\boldsymbol{J_2}$ vérifient la propriété *Jprop*, la matrice $\boldsymbol{J}$ obtenue la vérifie bien aussi.

$$\begin{aligned}
\boldsymbol{x_1}(k+1) &= \boldsymbol{K_1 t_1}(k+1) + \boldsymbol{P_1 x_1}(k) + \boldsymbol{Q_1 u}(k) \\
\boldsymbol{x_2}(k+1) &= \boldsymbol{K_2 t_2}(k+1) + \boldsymbol{P_2 x_2}(k) + \boldsymbol{Q_2 u}(k) \\
\boldsymbol{x}(k+1) &= \underbrace{\left(\begin{array}{c|c|c|c} \boldsymbol{K_1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{K_2} & \boldsymbol{0} & \boldsymbol{0} \end{array}\right)}_{\boldsymbol{K}} \boldsymbol{t}(k+1) + \underbrace{\left(\begin{array}{c|c} \boldsymbol{P_1} & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{P_2} \end{array}\right)}_{\boldsymbol{P}} \boldsymbol{x}(k) + \underbrace{\left(\begin{array}{c} \boldsymbol{Q_1} \\ \hline \boldsymbol{Q_2} \end{array}\right)}_{\boldsymbol{Q}} \boldsymbol{u}(k)
\end{aligned} \tag{3.67}$$

$$\begin{aligned}
\boldsymbol{y}(k) &= \boldsymbol{y_1}(k) + \boldsymbol{y_2}(k) \\
\boldsymbol{y}(k) &= \underbrace{\left(\begin{array}{c|c|c|c} \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{I} \end{array}\right)}_{\boldsymbol{L}} \boldsymbol{t}(k+1) + \underbrace{\left(\begin{array}{c|c} \boldsymbol{0} & \boldsymbol{0} \end{array}\right)}_{\boldsymbol{R}} \boldsymbol{x}(k) + \underbrace{\left(\begin{array}{c} \boldsymbol{0} \end{array}\right)}_{\boldsymbol{S}} \boldsymbol{u}(k)
\end{aligned} \tag{3.68}$$

Une SIF décrivant la décomposition parallèle des SIF $(\boldsymbol{J_1}, \boldsymbol{K_1}, \boldsymbol{L_1}, \boldsymbol{M_1}, \boldsymbol{N_1}, \boldsymbol{P_1}, \boldsymbol{Q_1}, \boldsymbol{R_1}, \boldsymbol{S_1})$ et $(\boldsymbol{J_2}, \boldsymbol{K_2}, \boldsymbol{L_2}, \boldsymbol{M_2}, \boldsymbol{N_2}, \boldsymbol{P_2}, \boldsymbol{Q_2}, \boldsymbol{R_2}, \boldsymbol{S_2})$ est donc :

$$\boldsymbol{Z} = \left(\begin{array}{cccc|cc|c} -\boldsymbol{J_1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{M_1} & \boldsymbol{0} & \boldsymbol{N_1} \\ \boldsymbol{0} & -\boldsymbol{J_2} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{M_2} & \boldsymbol{N_2} \\ \boldsymbol{L_1} & \boldsymbol{0} & -\boldsymbol{I} & \boldsymbol{0} & \boldsymbol{R_1} & \boldsymbol{0} & \boldsymbol{S_1} \\ \boldsymbol{0} & \boldsymbol{L_2} & \boldsymbol{0} & -\boldsymbol{I} & \boldsymbol{0} & \boldsymbol{R_2} & \boldsymbol{S_2} \\ \hline \boldsymbol{K_1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{P_1} & \boldsymbol{0} & \boldsymbol{Q_1} \\ \boldsymbol{0} & \boldsymbol{K_2} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{P_2} & \boldsymbol{Q_2} \\ \hline \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \end{array}\right) \tag{3.69}$$

Formalisation. On se donne des SIF `sif1` et `sif2` dont les entrées et les sorties ont respectivement les mêmes tailles. On définit d'abord la matrice $\boldsymbol{J}$ obtenue dans l'équation (3.66), appelée `Jcascade`, afin de prouver qu'elle vérifie bien la propriété *Jprop*. On peut ensuite définir la SIF donnée par l'équation (3.69) ci-dessus.

```
Context {nu ny   nx1 nt1   nx2 nt2   : nat}.

Variable sif1 : @SIF nu ny nx1 nt1.
Variable sif2 : @SIF nu ny nx2 nt2.

Notation J1 := (SIF_J sif1).
  (* idem pour K1, L1, M1, N1, P1, Q1, R1, S1 *)
Notation J2 := (SIF_J sif2).
  (* idem pour K2, L2, M2, N2, P2, Q2, R2, S2 *)
Notation I := 1%:M. (* matrice identité *)

Definition Jparallel :=
  block_mx (block_mx (block_mx  J1    0
                                0   J2)  0
                     (row_mx -L1    0)  I) 0
              (row_mx (row_mx   0  -L2)  0) I.
Lemma Jprop_parallel : Jprop Jparallel.
```

```
Definition SIF_parallel : @SIF nu ny _ _ :=
  Build_SIF
  (* SIF_Jprop *)  Jprop_parallel
  (* SIF_K *)      (col_mx (row_mx (row_mx (row_mx K1  0) 0) 0)
                           (row_mx (row_mx (row_mx  0 K2) 0) 0))
  (* SIF_L *)      (row_mx (row_mx (row_mx 0 0) I) I)
  (* SIF_M *)      (col_mx (col_mx (col_mx (row_mx M1  0)
                                           (row_mx  0 M2))
                                           (row_mx R1  0))
                                           (row_mx  0 R2))
  (* SIF_N *)      (col_mx (col_mx (col_mx N1
                                           N2)
                                           S1)
                                           S2)
  (* SIF_P *)      (col_mx (row_mx P1  0)
                           (row_mx  0 P2))
  (* SIF_Q *)      (col_mx Q1
                           Q2)
  (* SIF_R *)      0
  (* SIF_S *)      0.
```

On prouve facilement que le filtre correspondant à la SIF construite est bien la somme des filtres des deux SIF considérées.

```
Theorem SIF_parallel_preserves_filter :
  filter_from_SIF SIF_parallel =
  (filter_from_SIF sif1) \+ (filter_from_SIF sif2).
```

3.3.5 Décomposition en cascade

On veut traduire vers une SIF une réalisation donnée sous forme d'une cascade de deux réalisations (section 2.3.3). Comme dans la section précédente, on suppose qu'on a traduit chacune vers une SIF séparément, obtenant les SIF $(\boldsymbol{J_1}, \boldsymbol{K_1}, \boldsymbol{L_1}, \boldsymbol{M_1}, \boldsymbol{N_1}, \boldsymbol{P_1}, \boldsymbol{Q_1}, \boldsymbol{R_1}, \boldsymbol{S_1})$ et $(\boldsymbol{J_2}, \boldsymbol{K_2}, \boldsymbol{L_2}, \boldsymbol{M_2}, \boldsymbol{N_2}, \boldsymbol{P_2}, \boldsymbol{Q_2}, \boldsymbol{R_2}, \boldsymbol{S_2})$ respectivement. On note à nouveau $\mathscr{H}_1$ et $\mathscr{H}_2$ les filtres modèles associés à ces deux réalisations, et $\mathscr{H}$ leur assemblage en cascade.

Considérons un signal d'entrée $\boldsymbol{u}$. On note $\boldsymbol{v}$ la sortie de $\mathscr{H}_1$ correspondante. Lorsqu'on donne $\boldsymbol{v}$ en entrée à $\mathscr{H}_2$, on récupère par définition la sortie correspondant à $\boldsymbol{u}$ à travers $\mathscr{H}$, notée $\boldsymbol{y}$ comme d'habitude :

$$\boldsymbol{v} = \mathscr{H}_1\{\boldsymbol{u}\} \qquad \boldsymbol{y} = \mathscr{H}_2\{\boldsymbol{v}\} \tag{3.70}$$

On note n_{y1} la taille de $\boldsymbol{v}$, c'est-à-dire la taille à la fois de la sortie de $\mathscr{H}_1$ et de l'entrée de $\mathscr{H}_2$. On note à nouveau $\boldsymbol{x_1}$ le signal d'état et $\boldsymbol{t_1}$ le signal intermédiaire de la première SIF pour l'entrée $\boldsymbol{u}$, et $\boldsymbol{x_2}$ et $\boldsymbol{t_2}$ ceux de la seconde SIF pour l'entrée $\boldsymbol{v}$, de tailles respectives n_{x1}, n_{t1}, n_{x2} et n_{t2}. Comme à la section précédente, les indices ne représentent pas des composantes mais des signaux différents. Pour la SIF qu'on va construire pour décrire $\mathscr{H}$, on garde encore les notations habituelles : $\boldsymbol{x}$ et $\boldsymbol{t}$, de tailles n_x et n_t.

À chaque temps k, pour obtenir la sortie $\boldsymbol{y}(k)$ de $\mathscr{H}$ ainsi que les valeurs dont on aura besoin au temps $k+1$, on calcule :

— $\boldsymbol{t_1}(k+1)$ à partir de $\boldsymbol{x_1}(k)$ et $\boldsymbol{u}(k)$
— $\boldsymbol{x_1}(k+1)$ et $\boldsymbol{v}(k)$ (indépendants l'un de l'autre) à partir de $\boldsymbol{t_1}(k+1)$, $\boldsymbol{x_1}(k)$ et $\boldsymbol{u}(k)$
— $\boldsymbol{t_2}(k+1)$ à partir de $\boldsymbol{x_2}(k)$ et $\boldsymbol{v}(k)$
— $\boldsymbol{x_2}(k+1)$ et $\boldsymbol{y_2}(k)$ (indépendants l'un de l'autre) à partir de $\boldsymbol{t_2}(k+1)$, $\boldsymbol{x_2}(k)$ et $\boldsymbol{v}(k)$
— $\boldsymbol{y}(k)$ à partir de $\boldsymbol{t_2}(k+1)$, $\boldsymbol{x_2}(k)$ et $\boldsymbol{v}(k)$

Les informations qu'on a besoin de conserver d'une étape sur l'autre sont à la fois $\boldsymbol{x_1}(k)$ et $\boldsymbol{x_2}(k)$. On pose donc :

$$\boldsymbol{x} = \begin{pmatrix} \boldsymbol{x_1} \\ \hline \boldsymbol{x_2} \end{pmatrix} \qquad (\text{donc } n_x = n_{x1} + n_{x2}) \tag{3.71}$$

On remarque que le vecteur $\boldsymbol{v}(k)$ est utilisé pour calculer $\boldsymbol{t_2}(k+1)$, $\boldsymbol{x_2}(k+1)$, et $\boldsymbol{y}(k)$, et que lui-même dépend de $\boldsymbol{t_1}(k+1)$. On pose donc, et l'ordre est important :

$$\boldsymbol{t}(k+1) = \begin{pmatrix} \boldsymbol{t_1}(k+1) \\ \hline \boldsymbol{v}(k) \\ \hline \boldsymbol{t_2}(k+1) \end{pmatrix} \qquad (\text{donc } n_t = n_{t1} + n_{y1} + n_{t2}) \tag{3.72}$$

Les calculs sont ensuite :

$$\begin{aligned} \boldsymbol{J_1 t_1}(k+1) &= \boldsymbol{M_1 x_1}(k) + \boldsymbol{N_1 u}(k) \\ \boldsymbol{v}(k) &= \boldsymbol{L_1 t_1}(k+1) + \boldsymbol{R_1 x_1}(k) + \boldsymbol{S_1 u}(k) \\ \boldsymbol{J_2 t_2}(k+1) &= \boldsymbol{M_2 x_2}(k) + \boldsymbol{N_2 v}(k) \\ \underbrace{\left(\begin{array}{c|c|c} \boldsymbol{J_1} & \boldsymbol{0} & \boldsymbol{0} \\ \hline -\boldsymbol{L_1} & \boldsymbol{I} & \boldsymbol{0} \\ \hline \boldsymbol{0} & -\boldsymbol{N_2} & \boldsymbol{J_2} \end{array}\right)}_{\boldsymbol{J}} \boldsymbol{t}(k+1) &= \underbrace{\left(\begin{array}{c|c} \boldsymbol{M_1} & \boldsymbol{0} \\ \hline \boldsymbol{R_1} & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{M_2} \end{array}\right)}_{\boldsymbol{M}} \boldsymbol{x}(k) + \underbrace{\left(\begin{array}{c} \boldsymbol{N_1} \\ \hline \boldsymbol{S_1} \\ \hline \boldsymbol{0} \end{array}\right)}_{\boldsymbol{N}} \boldsymbol{u}(k) \end{aligned} \tag{3.73}$$

Encore une fois, comme $\boldsymbol{J_1}$ et $\boldsymbol{J_2}$ vérifient la propriété *Jprop*, la matrice $\boldsymbol{J}$ obtenue la vérifie bien aussi.

$$\begin{aligned} \boldsymbol{x_1}(k+1) &= \boldsymbol{K_1 t_1}(k+1) + \boldsymbol{P_1 x_1}(k) + \boldsymbol{Q_1 u}(k) \\ \boldsymbol{x_2}(k+1) &= \boldsymbol{K_2 t_2}(k+1) + \boldsymbol{P_2 x_2}(k) + \boldsymbol{Q_2 v}(k) \\ \boldsymbol{x}(k+1) &= \underbrace{\left(\begin{array}{c|c|c} \boldsymbol{K_1} & \boldsymbol{0} & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{Q_2} & \boldsymbol{K_2} \end{array}\right)}_{\boldsymbol{K}} \boldsymbol{t}(k+1) + \underbrace{\left(\begin{array}{c|c} \boldsymbol{P_1} & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{P_2} \end{array}\right)}_{\boldsymbol{P}} \boldsymbol{x}(k) + \underbrace{\left(\begin{array}{c} \boldsymbol{Q_1} \\ \hline \boldsymbol{0} \end{array}\right)}_{\boldsymbol{Q}} \boldsymbol{u}(k) \end{aligned} \tag{3.74}$$

$$\begin{aligned} \boldsymbol{y}(k) &= \boldsymbol{L_2 t_2}(k+1) + \boldsymbol{R_2 x_2}(k) + \boldsymbol{S_2 v}(k) \\ \boldsymbol{y}(k) &= \underbrace{\left(\begin{array}{c|c|c} \boldsymbol{0} & \boldsymbol{S_2} & \boldsymbol{L_2} \end{array}\right)}_{\boldsymbol{L}} \boldsymbol{t}(k+1) + \underbrace{\left(\begin{array}{c|c} \boldsymbol{0} & \boldsymbol{R_2} \end{array}\right)}_{\boldsymbol{R}} \boldsymbol{x}(k) + \underbrace{\left(\boldsymbol{0}\right)}_{\boldsymbol{S}} \boldsymbol{u}(k) \end{aligned} \tag{3.75}$$

Une SIF décrivant la décomposition en cascade des SIF $(\boldsymbol{J_1}, \boldsymbol{K_1}, \boldsymbol{L_1}, \boldsymbol{M_1}, \boldsymbol{N_1}, \boldsymbol{P_1}, \boldsymbol{Q_1}, \boldsymbol{R_1}, \boldsymbol{S_1})$ et $(\boldsymbol{J_2}, \boldsymbol{K_2}, \boldsymbol{L_2}, \boldsymbol{M_2}, \boldsymbol{N_2}, \boldsymbol{P_2}, \boldsymbol{Q_2}, \boldsymbol{R_2}, \boldsymbol{S_2})$ est donc :

$$\boldsymbol{Z} = \left(\begin{array}{ccc|ccc} -\boldsymbol{J_1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{M_1} & \boldsymbol{0} & \boldsymbol{N_1} \\ \boldsymbol{L_1} & -\boldsymbol{I} & \boldsymbol{0} & \boldsymbol{R_1} & \boldsymbol{0} & \boldsymbol{S_1} \\ \boldsymbol{0} & \boldsymbol{N_2} & -\boldsymbol{J_2} & \boldsymbol{0} & \boldsymbol{M_2} & \boldsymbol{0} \\ \hline \boldsymbol{K_1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{P_1} & \boldsymbol{0} & \boldsymbol{Q_1} \\ \boldsymbol{0} & \boldsymbol{Q_2} & \boldsymbol{K_2} & \boldsymbol{0} & \boldsymbol{P_2} & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{S_2} & \boldsymbol{L_2} & \boldsymbol{0} & \boldsymbol{R_2} & \boldsymbol{0} \end{array}\right) \tag{3.76}$$

Formalisation. La formalisation est similaire à celle de la décomposition parallèle en section 3.3.4. On se donne des SIF `sif1` et `sif2` telles que la sortie de `sif1` et l'entrée de `sif2` ont la même taille. On définit d'abord la matrice $\boldsymbol{J}$ obtenue dans l'équation (3.73), appelée `Jcascade`, afin de prouver qu'elle vérifie bien la propriété *Jprop*. On peut alors définir la SIF déterminée ci-dessus.

```
Context {nu ny   ny1 nx1 nt1   nx2 nt2   : nat}.

Variable sif1 : @SIF nu  ny1 nx1 nt1.
Variable sif2 : @SIF ny1 ny  nx2 nt2.

Notation J1 := (SIF_J sif1).
  (* idem pour K1, L1, M1, N1, P1, Q1, R1, S1 *)
Notation J2 := (SIF_J sif2).
  (* idem pour K2, L2, M2, N2, P2, Q2, R2, S2 *)

Definition Jcascade := block_mx (block_mx    J1     0
                                           (-L1) 1%:M )   0
                                 (row_mx     0   (-N2))  J2.
Lemma Jprop_cascade : Jprop Jcascade.

Definition SIF_cascade : @SIF nu ny _ _ :=
  Build_SIF
  (* SIF_Jprop *)  Jprop_cascade
  (* SIF_K *)       (block_mx (row_mx K1  0)  0
                               (row_mx  0 Q2) K2)
  (* SIF_L *)       (row_mx (row_mx 0 S2) L2)
  (* SIF_M *)       (col_mx (col_mx (row_mx M1  0)
                                    (row_mx R1  0))
                                    (row_mx  0 M2))
  (* SIF_N *)       (col_mx (col_mx N1
                                    S1)
                                     0)
  (* SIF_P *)       (block_mx P1  0
                                0 P2)
  (* SIF_Q *)       (col_mx Q1
                           0)
  (* SIF_R *)       (row_mx 0 R2)
  (* SIF_S *)       0.
```

On prouve que la SIF construite décrit bien la cascade des filtres des deux

SIF prises en paramètre, c'est-à-dire leur composition en tant que fonctions mathématiques (`Basics.compose`). La preuve n'est pas très longue. Le seul lemme intermédiaire consiste à montrer que le signal d'état de la nouvelle SIF est bien donné par l'équation (3.71).

```
Theorem SIF_cascade_preserves_filter :
  filter_from_SIF SIF_cascade =
  Basics.compose (filter_from_SIF sif2) (filter_from_SIF sif1).
```

3.4 Transformations mathématiques d'une SIF

On a vu en section 1.4.5 que différentes réalisations d'un même filtre modèle peuvent être plus ou moins désirables selon de nombreux critères : complexité, coût, compatibilité matérielle, etc. Dans certains cas, on a déjà trouvé une réalisation qui offre un bon compromis entre les critères souhaités, et on cherche seulement à s'assurer que son comportement numérique en précision finie est raisonnable. On traduit donc cette réalisation en une SIF comme dans les sections 3.2.4 et 3.3 en conservant le filtre implémenté, puis on applique l'analyse des erreurs d'arrondi du chapitre 4 à la SIF obtenue. Dans d'autres cas, on part d'une réalisation décrivant le filtre modèle souhaité, mais l'implémentation décrite par cette réalisation n'est pas particulièrement bonne pour les critères désirés. On ajoute alors une étape après la traduction vers une SIF : une transformation de la SIF obtenue vers une nouvelle SIF qui conserve le filtre modèle, mais modifie le filtre implémenté dans le but de lui donner de meilleures propriétés.

Je présente deux transformations d'une SIF vers une autre en préservant le filtre modèle. La transposition en section 3.4.1 s'appuie, comme son nom l'indique, sur la transposition des matrices de coefficients. Cette transformation ne s'applique qu'aux filtres SISO. Le changement de base en section 3.4.2, nommé pour sa signification du point de vue de l'algèbre linéaire, s'applique à n'importe quel filtre. Il est notamment utilisé pour diagonaliser certaines matrices de la SIF, ce qui simplifie beaucoup les calculs correspondants dans le filtre implémenté.

Ces deux transformations sont d'abord présentées dans le cas du *State-Space*, sur lequel elles sont plus faciles à expliquer. On passe ensuite au cas qui nous intéresse, celui de la SIF. On verra d'ailleurs que les preuves de conservation du filtre modèle dans le cas du *State-Space* sont utilisées dans celles sur la SIF, qui sont en conséquence très faciles.

3.4.1 Transposition

La transposition est la même transformation que le théorème de transposition des graphes de flot de données présenté dans la section 2.3.2. Elle ne s'applique donc qu'aux filtres SISO. Je présente d'abord le cas simple du *State-Space*, puis celui plus complexe de la SIF.

Transposition du *State-Space*. Soit $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ un *State-Space* SISO. Transposer ce *State-Space* revient simplement à transposer la matrice $\boldsymbol{Z}$ qui le caractérise :

$$\boldsymbol{Z} = \left(\begin{array}{c|c} \boldsymbol{A} & \boldsymbol{B} \\ \hline \boldsymbol{C} & \boldsymbol{D} \end{array}\right) \qquad \boldsymbol{Z}^T = \left(\begin{array}{c|c} \boldsymbol{A}^T & \boldsymbol{C}^T \\ \hline \boldsymbol{B}^T & \boldsymbol{D}^T \end{array}\right) \tag{3.77}$$

Le *State-Space* transposé $(\boldsymbol{A}^T, \boldsymbol{C}^T, \boldsymbol{B}^T, \boldsymbol{D}^T)$ décrit le même filtre modèle que le *State-Space* $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$.

On remarque que comme $\boldsymbol{D}$ est de taille 1×1 pour un *State-Space* SISO, on a $\boldsymbol{D}^T = \boldsymbol{D}$.

Tout ceci est facile à exprimer en Coq.

```
Variable stsp : @StateSpace RT 1 1 nx. (* SISO \stsp *)
Notation A := (StSp_A stsp). (* idem pour B, C, D *)

Definition StSp_transpose := Build_StateSpace A^T C^T B^T D^T.

Theorem StSp_transpose_same_filter :
  filter_from_StSp StSp_transpose = filter_from_StSp stsp.
```

Pour prouver ce théorème, on montre récursivement sur $k \in \mathbb{Z}$ (en utilisant `peanoindz` de la section 2.1.5) la propriété suivante. Soit $\boldsymbol{u}$ un signal vectoriel d'entrée de taille 1. Notons $\boldsymbol{x_Z}$ le signal d'état correspondant dans le *State-Space* $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$, et $\boldsymbol{x_{Z^T}}$ celui dans le *State-Space* $(\boldsymbol{A}^T, \boldsymbol{C}^T, \boldsymbol{B}^T, \boldsymbol{D}^T)$. Alors :

$$\forall k \in \mathbb{Z}, \quad \forall \boldsymbol{M} \in \mathtt{RT}^{n_x \times n_x}, \quad \boldsymbol{AM} = \boldsymbol{MA} \;\Rightarrow\; \boldsymbol{B}^T\boldsymbol{M}^T\boldsymbol{x_{Z^T}}(k) = \boldsymbol{CMx_Z}(k) \tag{3.78}$$

Transposition de la SIF. Soit $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ une SIF. Si on transpose simplement la matrice $\boldsymbol{Z}$ correspondante :

$$\boldsymbol{Z} = \left(\begin{array}{c|c|c} -\boldsymbol{J} & \boldsymbol{M} & \boldsymbol{N} \\ \hline \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{Q} \\ \hline \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{S} \end{array}\right) \qquad \text{(rappel de (3.28))}$$

on n'obtient pas forcément une SIF. En effet, on remplace alors la matrice $\boldsymbol{J}$ par $\boldsymbol{J}^T$ qui ne vérifie pas *Jprop* (sauf cas particulier où $\boldsymbol{J}$ est l'identité).

Transposer $\boldsymbol{Z}$ produit néanmoins une structure qui décrit toujours le même filtre modèle, au sens où la relation suivante définit la même sortie $\boldsymbol{y}$ pour l'entrée $\boldsymbol{u}$ que la SIF initiale :

$$\left\{\begin{array}{rcl} \boldsymbol{J}^T\boldsymbol{t'}(k+1) & = & \boldsymbol{K}^T\boldsymbol{x'}(k) + \boldsymbol{L}^T\boldsymbol{u}(k) \\ \boldsymbol{x'}(k+1) & = & \boldsymbol{M}^T\boldsymbol{t'}(k+1) + \boldsymbol{P}^T\boldsymbol{x'}(k) + \boldsymbol{R}^T\boldsymbol{u}(k) \\ \boldsymbol{y}(k) & = & \boldsymbol{N}^T\boldsymbol{t'}(k+1) + \boldsymbol{Q}^T\boldsymbol{x'}(k) + \boldsymbol{S}^T\boldsymbol{u}(k) \end{array}\right. \tag{3.79}$$

où on utilise les notations $\boldsymbol{x'}$ et $\boldsymbol{t'}$ pour ne pas confondre avec les signaux d'état ou intermédiaire de la SIF initiale. Le problème est qu'en l'absence de la propriété *Jprop*, le nouveau vecteur intermédiaire $\boldsymbol{t'}(k+1)$ ne peut pas être calculé dans l'ordre croissant des indices. Cependant, on sait que $\boldsymbol{J}^T$ est triangulaire *supérieure* avec des 1 sur la diagonale. Le vecteur $\boldsymbol{t'}(k+1)$ peut donc être calculé dans l'ordre *décroissant* des indices. Afin de retomber sur l'ordre de calcul conventionnel de la SIF, on veut retourner ce vecteur. On doit pour cela retourner verticalement matrices qui servent à calculer $\boldsymbol{t'}(k+1)$, à savoir $\boldsymbol{J}^T$, $\boldsymbol{K}^T$ et $\boldsymbol{L}^T$, ainsi que retourner horizontalement les matrices qui se font multiplier par $\boldsymbol{t'}(k+1)$, qui sont $\boldsymbol{J}^T$, $\boldsymbol{M}^T$ et $\boldsymbol{N}^T$. On remarque qu'on a retourné $\boldsymbol{J}^T$

dans les deux sens, donc on lui a finalement appliqué une symétrie centrale ; la matrice ainsi obtenue vérifie *Jprop* donc on est bien retombé sur une SIF.

Pour pouvoir écrire facilement les transformations ci-dessus, on note $\boldsymbol{\Pi}$ la matrice anti-diagonale avec des 1 :

$$\boldsymbol{\Pi} \triangleq \begin{pmatrix} 0 & \cdots & 0 & 1 \\ \vdots & \iddots & \iddots & 0 \\ 0 & \iddots & \iddots & \vdots \\ 1 & 0 & \cdots & 0 \end{pmatrix} \tag{3.80}$$

On rappelle que pour retourner horizontalement une matrice $\boldsymbol{A}$ quelconque, il suffit de considérer $\boldsymbol{A\Pi}$, et pour la retourner verticalement $\boldsymbol{\Pi A}$.

La SIF obtenue ci-dessus, qui est le résultat de la transposition de la SIF initiale $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$, est alors décrite par :

$$\boldsymbol{Z}_{\text{transposition}} = \left(\begin{array}{c|c|c} -\boldsymbol{\Pi J}^T\boldsymbol{\Pi} & \boldsymbol{\Pi K}^T & \boldsymbol{\Pi L}^T \\ \hline \boldsymbol{M}^T\boldsymbol{\Pi} & \boldsymbol{P}^T & \boldsymbol{R}^T \\ \hline \boldsymbol{N}^T\boldsymbol{\Pi} & \boldsymbol{Q}^T & \boldsymbol{S}^T \end{array}\right) \tag{3.81}$$

Cette SIF décrit le même filtre modèle que la SIF initiale.

La preuve est facile à partir de la transposition du *State-Space*, grâce à la traduction de la SIF présentée dans la section 3.2.5. On rappelle que cette transformation conserve le filtre modèle même si pas toujours le filtre implémenté, or c'est le premier qui nous intéresse ici. En utilisant l'équation (3.45), le *State-Space* correspondant à la SIF transposée est :

$$\left(\begin{array}{c|c} \boldsymbol{M}^T\boldsymbol{\Pi}\left(\boldsymbol{\Pi J}^T\boldsymbol{\Pi}\right)^{-1}\boldsymbol{\Pi K}^T + \boldsymbol{P}^T & \boldsymbol{M}^T\boldsymbol{\Pi}\left(\boldsymbol{\Pi J}^T\boldsymbol{\Pi}\right)^{-1}\boldsymbol{\Pi L}^T + \boldsymbol{R}^T \\ \hline \boldsymbol{N}^T\boldsymbol{\Pi}\left(\boldsymbol{\Pi J}^T\boldsymbol{\Pi}\right)^{-1}\boldsymbol{\Pi K}^T + \boldsymbol{Q}^T & \boldsymbol{N}^T\boldsymbol{\Pi}\left(\boldsymbol{\Pi J}^T\boldsymbol{\Pi}\right)^{-1}\boldsymbol{\Pi L}^T + \boldsymbol{S}^T \end{array}\right) \tag{3.82}$$

En utilisant que $\boldsymbol{\Pi}^2 = \boldsymbol{I}$ et en factorisant les transpositions, on obtient :

$$\left(\begin{array}{c|c} \left(\boldsymbol{K J}^{-1}\boldsymbol{M} + \boldsymbol{P}\right)^T & \left(\boldsymbol{L J}^{-1}\boldsymbol{M} + \boldsymbol{R}\right)^T \\ \hline \left(\boldsymbol{K J}^{-1}\boldsymbol{N} + \boldsymbol{Q}\right)^T & \left(\boldsymbol{L J}^{-1}\boldsymbol{N} + \boldsymbol{S}\right)^T \end{array}\right) \tag{3.83}$$

En comparant avec l'équation (3.45), on reconnaît exactement la transposition du *State-Space* correspondant à la SIF initiale $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$.

Formalisation de la transposition de la SIF. Pour définir ce qui précède en Coq, on va utiliser `row_perm` et `col_perm` qui prennent en argument une permutation et une matrice, et appliquent cette permutation aux lignes ou colonnes de la matrice. Comme le but est de retourner ces lignes ou colonnes, on définit la permutation `s_rev` qui retourne un ensemble d'indices.

```
Definition s_rev {n : nat} := perm.perm (@rev_ord_inj n).
```

On peut alors définir la SIF transposée à partir de l'équation (3.81) :

```
Variable sif : @SIF RT 1 1 nx nt. (* SIF SISO *)

Notation J := (SIF_J sif). (* idem pour K, L, M, N, P, Q, R, S *)

Definition Jsiftr := row_perm s_rev (col_perm s_rev (J^T)).

Lemma Jprop_Jsiftr : Jprop Jsiftr.
```

```
Definition SIF_transpose := Build_SIF
(* SIF_Jprop *)  Jprop_Jsiftr
(* SIF_K *)      (col_perm s_rev M^T)
(* SIF_L *)      (col_perm s_rev N^T)
(* SIF_M *)      (row_perm s_rev K^T)
(* SIF_N *)      (row_perm s_rev L^T)
(* SIF_P *)      P^T
(* SIF_Q *)      Rsif^T
(* SIF_R *)      Q^T
(* SIF_S *)      S^T.
```

Comme expliqué précédemment, la preuve de la conservation du filtre modèle est très courte en utilisant la même propriété sur le *State-Space*.

```
Theorem SIF_transpose_same_filter :
  filter_from_SIF SIF_transpose = filter_from_SIF sif.
```

3.4.2 Changement de base

Le changement de base s'applique à n'importe quel *State-Space* ou SIF, sans nécessité d'hypothèse SISO. Encore une fois, je présente d'abord le cas plus facile du *State-Space*.

Changement de base du *State-Space*. Soit $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ un *State-Space*. Pour n'importe quelle matrice inversible $\boldsymbol{T} \in \mathtt{RT}^{n_x \times n_x}$, le *State-Space* $(\boldsymbol{T}^{-1}\boldsymbol{A}\boldsymbol{T}, \boldsymbol{T}^{-1}\boldsymbol{B}, \boldsymbol{C}\boldsymbol{T}, \boldsymbol{D})$ définit le même filtre modèle. Ce nouveau *State-Space* est caractérisé par la matrice :

$$\boldsymbol{Z}_{\text{chgt\_base}} = \left(\begin{array}{c|c} \boldsymbol{T}^{-1}\boldsymbol{A}\boldsymbol{T} & \boldsymbol{T}^{-1}\boldsymbol{B} \\ \hline \boldsymbol{C}\boldsymbol{T} & \boldsymbol{D} \end{array}\right) = \left(\begin{array}{c|c} \boldsymbol{T}^{-1} & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{I} \end{array}\right) \left(\begin{array}{c|c} \boldsymbol{A} & \boldsymbol{B} \\ \hline \boldsymbol{C} & \boldsymbol{D} \end{array}\right) \left(\begin{array}{c|c} \boldsymbol{T} & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{I} \end{array}\right) \tag{3.84}$$

On appelle cette transformation *changement de base* car on a appliqué un changement de base usuel d'algèbre linéaire à la matrice $\boldsymbol{Z}$.

Le changement de base est facile à définir en Coq.

```
Variable stsp : @StateSpace RT nu ny nx.
Notation A := (StSp_A stsp). (* idem pour B, C, D *)

Variable T : 'M[RT]_nx.
Hypothesis unitmx_T : T \in unitmx. (* càd T est inversible *)

Definition StSp_change_base :=
  Build_StateSpace (invmx T *m A *m T) (invmx T *m B) (C *m T) D.
```

La preuve est très rapide. L'étape principale consiste à montrer que le vecteur d'état du nouveau *State-Space* est $\boldsymbol{T}^{-1}\boldsymbol{x}(k)$, où $\boldsymbol{x}(k)$ est celui du *State-Space* initial.

```
Lemma StSp_change_base_x_k (u : signal) k :
  x StSp_change_base u k = invmx T *m x stsp u k.

Theorem StSp_change_base_same_filter :
  filter_from_StSp StSp_change_base = filter_from_StSp stsp.
```

Changement de base de la SIF. Soit $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ une SIF. Pour toutes matrices $\boldsymbol{Y}$, $\boldsymbol{W}$ et $\boldsymbol{U}$ telles que $\boldsymbol{U}$ est inversible et $\boldsymbol{YJW}$ vérifie la propriété *Jprop*, la SIF suivante définit le même filtre modèle :

$$\begin{aligned} \boldsymbol{Z}_{\text{chgt\_base}} &= \left(\begin{array}{c|c|c} \boldsymbol{Y} & \boldsymbol{0} & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{U}^{-1} & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{array}\right) \left(\begin{array}{c|c|c} -\boldsymbol{J} & \boldsymbol{M} & \boldsymbol{N} \\ \hline \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{Q} \\ \hline \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{S} \end{array}\right) \left(\begin{array}{c|c|c} \boldsymbol{W} & \boldsymbol{0} & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{U} & \boldsymbol{0} \\ \hline \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{array}\right) \\ &= \left(\begin{array}{c|c|c} -\boldsymbol{YJW} & \boldsymbol{YMU} & \boldsymbol{YN} \\ \hline \boldsymbol{U}^{-1}\boldsymbol{KW} & \boldsymbol{U}^{-1}\boldsymbol{PU} & \boldsymbol{U}^{-1}\boldsymbol{Q} \\ \hline \boldsymbol{LW} & \boldsymbol{RU} & \boldsymbol{S} \end{array}\right) \end{aligned} \tag{3.85}$$

Comme pour la transposition, il faut s'assurer qu'on obtient bien une SIF, c'est-à-dire que la nouvelle matrice $\boldsymbol{J}$, ici $\boldsymbol{YJW}$, vérifie bien *Jprop*. Mais ici, on a juste imposé cela comme condition à respecter lorsqu'on choisit les matrices $\boldsymbol{Y}$ et $\boldsymbol{W}$. Notons que cette condition implique que $\boldsymbol{Y}$ et $\boldsymbol{W}$ sont inversibles. En effet, on a vu que toute matrice vérifiant *Jprop* est inversible, et les facteurs d'un produit matriciel inversible sont aussi inversibles.

On remarque que les modifications sur les matrices $(\boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ correspondent à un changement de base de *State-Space* avec la matrice $\boldsymbol{U}$. Comme pour la transposition dans la section 3.4.1, la preuve de conservation du filtre modèle est très facile en passant par la traduction d'une SIF vers un *State-Space* de la section 3.2.5.

En Coq, on se donne une SIF et des matrices $\boldsymbol{Y}$, $\boldsymbol{W}$ et $\boldsymbol{U}$ vérifiant les conditions mentionnées, puis on construit la SIF ci-dessus.

```
Variable sif : @SIF nu ny nx nt.

Notation J := (SIF_J sif). (* idem pour K, L, M, N, P, Q, R, S *)

Variables (Y  W : 'M[RT]_nt) (U : 'M[RT]_nx).

Hypothesis unitmx_U : U \in unitmx.

Definition Jsifchg := Y *m J *m W.
Hypothesis Jprop_Jsifchg : Jprop Jsifchg.

Definition SIF_change_base := Build_SIF
(* SIF_Jprop *)  Jprop_Jsifchg
(* SIF_K *)      (invmx U *m K *m W)
(* SIF_L *)      (L *m W)
(* SIF_M *)      (Y *m M *m U)
(* SIF_N *)      (Y *m N)
(* SIF_P *)      (invmx U *m P *m U)
(* SIF_Q *)      (invmx U *m Q)
(* SIF_R *)      (Rsif *m U)
(* SIF_S *)      S.
```

Encore une fois, la preuve du théorème suivant est très courte grâce à son équivalent pour le *State-Space*.

```
Theorem change_base_SIF_same_filter :
  filter_from_SIF SIF_change_base = filter_from_SIF sif.
```

Le changement de base est une transformation très intéressante dans le cadre de la recherche d'une bonne implémentation d'un filtre modèle donné [44]. Cette transformation offre beaucoup d'options puisqu'on peut utiliser n'importe quelles matrices $\boldsymbol{Y}$, $\boldsymbol{W}$, et $\boldsymbol{U}$ à quelques conditions près. Elle peut par exemple être utilisée pour minimiser la sensibilité de la fonction de transfert par rapport aux coefficients du filtre [106].

La transposition et le changement de base ne sont pas utilisés dans la suite de ma formalisation, qui s'intéresse pour le moment à l'erreur finale d'une implémentation donnée sans la modifier. Ces transformations pourraient cependant contribuer à une perspective à long terme consistant à générer du code vérifié en incluant une étape d'optimisation de code (section 6.3.3).

Chapitre 4

Analyse des erreurs d'arrondi dans un filtre

On a vu dans le chapitre précédent que n'importe quelle réalisation de filtre LTI peut être ramenée à une SIF sans perdre d'information, et qu'il suffit donc d'effectuer l'analyse des erreurs d'arrondi sur la SIF. Cette analyse d'erreurs est l'objet du chapitre présent.

Les résultats de ce chapitre ne dépendent pas de l'arithmétique en précision finie choisie : ils s'appliquent entre autres à n'importe quel format de virgule flottante ou virgule fixe. Dans tout le chapitre, on considère un format de précision finie générique $\mathbb{F}$ et une fonction d'arrondi vers $\mathbb{F}$ comme expliqué en section 1.3.3. On suppose que 0 et -1 sont représentables dans ce format, ce qui est généralement le cas et est utile pour travailler avec une SIF comme on a vu en section 3.2.2. On suppose aussi donnée une implémentation de somme de produits pour ce format, idéalement correctement arrondie, mais sinon telle qu'on dispose d'une borne d'erreur raisonnable sur l'erreur locale sur la sortie (des algorithmes vérifiant l'une ou l'autre de ces conditions seront donnés dans le chapitre 5 dans le cadre de la virgule fixe).

J'ai mentionné dans les sections 1.1 et 1.4.4 qu'un problème majeur dans les filtres est la propagation des erreurs d'arrondi au cours de nombreuses itérations de calculs : cela présente d'une part un risque d'amplification de ces erreurs, et complique d'autre part leur analyse. Les sections 4.1 et 4.2 étudient cette propagation grâce à des *filtres d'erreurs*. Ceux-ci permettent de décrire l'erreur finale $\boldsymbol{\Delta y}$, c'est-à-dire la différence entre la sortie $\boldsymbol{y}$ du filtre modèle et la sortie $\boldsymbol{y}^*$ du filtre implémenté, en fonction d'erreurs locales sur un seul arrondi ou une seule somme de produits. L'intérêt est que ces erreurs locales ne dépendent pas d'autres erreurs d'arrondi, donc ne sont pas sujettes au problème de propagation. C'est bien sûr à la SIF qu'on veut appliquer ces filtres d'erreurs pour mettre à profit son universalité. Mais encore une fois, je présente d'abord le cas du *State-Space* dans la section 4.1, qui est similaire mais beaucoup plus facile à expliquer que celui de la SIF, qui constitue la section 4.2.

Ensuite, la section 4.3 présente le théorème du *Worst-Case Peak Gain*, qui permet de passer d'une borne sur l'entrée d'un filtre à une borne sur la sortie. En l'appliquant aux filtres d'erreurs, on obtiendra ainsi une borne sur l'erreur finale $\boldsymbol{\Delta y}$ en fonction de bornes sur les erreurs locales.

Dans ce chapitre, on ne dispose pas encore de borne explicite sur ces erreurs locales, vu qu'on n'a presque aucune hypothèse sur l'arithmétique en précision finie considérée. Cependant, dans les arithmétiques utilisées en pratique, on sait exprimer l'erreur sur un seul arrondi. De plus, il existe de nombreux algorithmes de somme de produits adaptés à différents formats, accompagnés d'une borne explicite sur l'erreur locale associée. Le chapitre 5 s'intéressera à de tels algorithmes pour la virgule fixe en complément à deux, qui est souvent utilisée pour implémenter des filtres.

4.1 Filtres d'erreurs d'un *State-Space*

On souhaite étudier la propagation des erreurs d'arrondi au cours des itérations d'un filtre. Encore une fois, c'est la SIF qu'on veut analyser, mais on présente d'abord le cas du *State-Space*, qui est à la fois similaire et plus simple.

Soit $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ un *State-Space*. On note $\mathscr{H}$ le filtre modèle correspondant, et $\mathscr{H}^*$ le filtre implémenté dans l'arithmétique en précision finie qu'on s'est donnée au début du chapitre. On considère aussi un signal d'entrée $\boldsymbol{u}$ donné. Comme en section 3.1.2, on note $\boldsymbol{y}$ la sortie correspondante et $\boldsymbol{x}$ le signal d'état de $\mathscr{H}$, et $\boldsymbol{y}^*$ et $\boldsymbol{x}^*$ ceux de $\mathscr{H}^*$. On rappelle que l'erreur finale à analyser est $\boldsymbol{\Delta y} = \boldsymbol{y}^* - \boldsymbol{y}$.

On a vu en section 3.1.2 que le filtre modèle $\mathscr{H}$ est défini par :

$$\mathscr{H} \quad \begin{cases} \boldsymbol{x}(k+1) &= \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}\boldsymbol{u}(k) \\ \boldsymbol{y}(k) &= \boldsymbol{C}\boldsymbol{x}(k) + \boldsymbol{D}\boldsymbol{u}(k) \end{cases} \qquad \text{(rappel de (3.3))}$$

Le filtre implémenté $\mathscr{H}^*$ est défini par l'équation (3.6), qui fait intervenir les arrondis coefficient par coefficient des matrices du *State-Space*. On définit ici des notations pour les matrices correspondantes :

$$\begin{array}{ll} \boldsymbol{A}^* \triangleq \bigcirc(\boldsymbol{A}) & \boldsymbol{B}^* \triangleq \bigcirc(\boldsymbol{B}) \\ \boldsymbol{C}^* \triangleq \bigcirc(\boldsymbol{C}) & \boldsymbol{D}^* \triangleq \bigcirc(\boldsymbol{D}) \end{array} \tag{4.1}$$

Voici l'équation (3.6) récrite en utilisant ces nouvelles notations :

$$\mathscr{H}^* \quad \begin{cases} \boldsymbol{x}^*(k+1) &= \left(\, \boldsymbol{A}^* \mid \boldsymbol{B}^* \,\right) \odot \begin{pmatrix} \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{pmatrix} \\ \boldsymbol{y}^*(k) &= \left(\, \boldsymbol{C}^* \mid \boldsymbol{D}^* \,\right) \odot \begin{pmatrix} \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{pmatrix} \end{cases} \tag{4.2}$$

Comme mentionné en section 3.2.2, on peut utiliser des formats et arrondis différents pour chaque coefficient et chaque somme de produits. Cela n'ajoute pas de difficulté conceptuelle, mais alourdit les notations.

Le problème de propagation des erreurs d'arrondi est ici mis en évidence par la dépendance de $\boldsymbol{x}^*(k+1)$ en $\boldsymbol{x}^*(k)$, qui est déjà un vecteur calculé, lui-même dépendant de $\boldsymbol{x}^*(k-1)$ et ainsi de suite.

Rappelons aussi les deux effets de la précision finie décrits en section 3.1.2, bien visibles dans la relation ci-dessus. D'une part, les opérations, ici des sommes de produits, sont calculées en précision finie. Ceci est représenté par le produit matriciel approché $\odot$, où chaque coefficient est obtenu à l'aide d'une somme de

produits. D'autre part, on a dû arrondir les coefficients du *State-Space*, d'où les matrices $(\boldsymbol{A}^*, \boldsymbol{B}^*, \boldsymbol{C}^*, \boldsymbol{D}^*)$.

On va exprimer l'erreur finale $\boldsymbol{\Delta y}$ en fonction d'*erreurs locales*, c'est-à-dire des erreurs dues à un seul calcul ou un seul arrondi, qui ne dépendent donc pas d'une propagation d'autres erreurs. Par exemple, $\boldsymbol{A}^* - \boldsymbol{A}$ est une matrice d'erreurs locales, puisque chaque coefficient est l'erreur produite par un seul arrondi. Un autre exemple de vecteur d'erreurs locales est :

$$\left(\boldsymbol{A}^* \mid \boldsymbol{B}^* \right) \odot \left(\frac{\boldsymbol{x}^*(k)}{\boldsymbol{u}(k)} \right) - \left(\boldsymbol{A}^* \mid \boldsymbol{B}^* \right) \cdot \left(\frac{\boldsymbol{x}^*(k)}{\boldsymbol{u}(k)} \right) \tag{4.3}$$

où « $\cdot$ » est le produit matriciel exact en précision infinie. En effet, chaque coefficient est l'erreur provenant d'une seule somme de produits. En revanche, le vecteur :

$$\boldsymbol{x}^*(k+1) - \boldsymbol{x}(k+1) = \left(\boldsymbol{A}^* \mid \boldsymbol{B}^* \right) \odot \left(\frac{\boldsymbol{x}^*(k)}{\boldsymbol{u}(k)} \right) - \left(\boldsymbol{A} \mid \boldsymbol{B} \right) \cdot \left(\frac{\boldsymbol{x}(k)}{\boldsymbol{u}(k)} \right) \tag{4.4}$$

n'est pas un vecteur d'erreurs locales, car les produits matriciels en précision finie et infinie ne font pas intervenir les mêmes termes : l'un utilise $\boldsymbol{A}^*$, $\boldsymbol{B}^*$ et $\boldsymbol{x}^*(k)$, l'autre $\boldsymbol{A}$, $\boldsymbol{B}$ et $\boldsymbol{x}(k)$. La différence ne provient donc pas uniquement d'une somme de produits en précision finie.

L'erreur finale $\boldsymbol{\Delta y}(k) = \boldsymbol{y}^*(k) - \boldsymbol{y}(k)$ n'a pas d'expression simple en fonctions d'erreurs locales si on se contente de comparer les équations (3.3) et (4.2) (les termes obtenus en soustrayant les expressions respectives de $\boldsymbol{y}^*(k)$ et $\boldsymbol{y}(k)$ ne se simplifient pas). Pour pouvoir décrire plus facilement cette erreur, on va la décomposer en deux morceaux : d'une part l'erreur due aux erreurs dans les sommes de produits en précision finie, et d'autre part l'erreur provenant de la quantifications des coefficients des matrices $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$.

On s'intéresse pour cela au filtre *modèle* associé au *State-Space* $(\boldsymbol{A}^*, \boldsymbol{B}^*, \boldsymbol{C}^*, \boldsymbol{D}^*)$. On note ce filtre $\mathscr{H}_c$ modèle (car sa différence avec $\mathscr{H}$ est la quantification des *coefficients*). Pour l'entrée $\boldsymbol{u}$ considérée au début de la section, on note $\boldsymbol{x}_c$ le vecteur d'état et $\boldsymbol{y}_c$ la sortie de $\mathscr{H}_c$. Ces signaux sont définis par des opérations en précision infinie puisqu'il s'agit d'un filtre modèle, mais ils font intervenir les coefficients arrondis :

$$\mathscr{H}_c \quad \left\{ \begin{array}{rcl} \boldsymbol{x}_c(k+1) & = & \boldsymbol{A}^* \boldsymbol{x}_c(k) + \boldsymbol{B}^* \boldsymbol{u}(k) \\ \boldsymbol{y}_c(k) & = & \boldsymbol{C}^* \boldsymbol{x}_c(k) + \boldsymbol{D}^* \boldsymbol{u}(k) \end{array} \right. \tag{4.5}$$

La sortie $\boldsymbol{y}_c$ prend donc en compte les erreurs dues aux arrondis des coefficients, mais pas celles dues aux calculs des sommes de produits. Ainsi, la différence entre $\mathscr{H}_c$ et $\mathscr{H}^*$ est la prise en compte des erreurs dans les opérations, tandis que la différence entre $\mathscr{H}_c$ et $\mathscr{H}$ est la prise en compte des arrondis des matrices de coefficients. On peut alors décomposer l'erreur finale en :

$$\boldsymbol{\Delta y} = \boldsymbol{y}^* - \boldsymbol{y} = \boldsymbol{y}_{\Delta op} + \boldsymbol{y}_{\Delta c} \tag{4.6}$$

où :

$$\boldsymbol{y}_{\Delta op} \triangleq \boldsymbol{y}^* - \boldsymbol{y}_c \tag{4.7}$$

est l'erreur due aux *opérations* calculées en précision finie (qui sont toutes des sommes de produits), et :

$$\boldsymbol{y}_{\Delta c} \triangleq \boldsymbol{y}_c - \boldsymbol{y} \tag{4.8}$$

est l'erreur due à la quantification des *coefficients*.

Les sections 4.1.1 et 4.1.2 expliquent comment les signaux d'erreurs $\boldsymbol{y}_{\Delta op}$ et $\boldsymbol{y}_{\Delta c}$ peuvent être exprimés comme les sorties de filtres qui seront appelés les *filtres d'erreurs*.

4.1.1 Filtre d'erreurs sur les opérations

On veut exprimer la différence $\boldsymbol{y}_{\Delta op} = \boldsymbol{y}^* - \boldsymbol{y}_c$ en fonction d'erreurs locales. On s'intéresse donc aux équations (4.2) et (4.5) définissant les filtres $\mathscr{H}^*$ et $\mathscr{H}_c$. Le produit matriciel en précision finie $\odot$ dans l'équation (4.2) les rend difficiles à comparer. On pose donc :

$$\begin{aligned}
\boldsymbol{\varepsilon}_{\boldsymbol{x}}(k) &\triangleq \left(\, \boldsymbol{A}^* \mid \boldsymbol{B}^* \,\right) \odot \left(\frac{\boldsymbol{x}^*(k)}{\boldsymbol{u}(k)}\right) - \left(\, \boldsymbol{A}^* \mid \boldsymbol{B}^* \,\right) \cdot \left(\frac{\boldsymbol{x}^*(k)}{\boldsymbol{u}(k)}\right) \\
\boldsymbol{\varepsilon}_{\boldsymbol{y}}(k) &\triangleq \left(\, \boldsymbol{C}^* \mid \boldsymbol{D}^* \,\right) \odot \left(\frac{\boldsymbol{x}^*(k)}{\boldsymbol{u}(k)}\right) - \left(\, \boldsymbol{C}^* \mid \boldsymbol{D}^* \,\right) \cdot \left(\frac{\boldsymbol{x}^*(k)}{\boldsymbol{u}(k)}\right)
\end{aligned} \tag{4.9}$$

Comme expliqué plus haut, ce sont des vecteurs d'erreurs locales, puisque chaque coefficient est l'erreur due à une seule opération : une somme de produits. On ne s'occupe pas encore ici de les borner, mais de seulement de faire apparaître des erreurs locales dans l'analyse. L'intérêt des vecteurs ci-dessus est qu'ils permettent de récrire l'équation (4.2) sous une forme plus proche de l'équation (4.5) :

$$\mathscr{H}^* \quad \left\{ \begin{array}{rcl} \boldsymbol{x}^*(k+1) & = & \boldsymbol{A}^*\boldsymbol{x}^*(k) + \boldsymbol{B}^*\boldsymbol{u}(k) + \boldsymbol{\varepsilon}_{\boldsymbol{x}}(k) \\ \boldsymbol{y}^*(k) & = & \boldsymbol{C}^*\boldsymbol{x}^*(k) + \boldsymbol{D}^*\boldsymbol{u}(k) + \boldsymbol{\varepsilon}_{\boldsymbol{y}}(k) \end{array} \right. \tag{4.10}$$

À cette équation, on soustrait alors l'équation (4.5) et on obtient :

$$\left\{ \begin{array}{rcl} \boldsymbol{x}^*(k+1) - \boldsymbol{x}_c(k+1) & = & \boldsymbol{A}^*(\boldsymbol{x}^*(k) - \boldsymbol{x}_c(k)) + \boldsymbol{\varepsilon}_{\boldsymbol{x}}(k) \\ \boldsymbol{y}^*(k) - \boldsymbol{y}_c(k) & = & \boldsymbol{C}^*(\boldsymbol{x}^*(k) - \boldsymbol{x}_c(k)) + \boldsymbol{\varepsilon}_{\boldsymbol{y}}(k) \end{array} \right. \tag{4.11}$$

On reconnaît presque une instance de l'équation (3.3) définissant le filtre associé à un *State-Space*, où le signal d'état est $\boldsymbol{x}^* - \boldsymbol{x}_c$ et la sortie est $\boldsymbol{y}^* - \boldsymbol{y}_c$ c'est-à-dire $\boldsymbol{y}_{\Delta op}$. Il reste juste à remplacer les termes $\boldsymbol{\varepsilon}_{\boldsymbol{x}}(k)$ et $\boldsymbol{\varepsilon}_{\boldsymbol{y}}(k)$ par le produit d'une matrice par un vecteur d'entrée. On pose donc :

$$\begin{aligned}
\boldsymbol{x}_{\Delta op}(k) &\triangleq \boldsymbol{x}^*(k) - \boldsymbol{x}_c(k) \\
\boldsymbol{\varepsilon}(k) &\triangleq \left(\frac{\boldsymbol{\varepsilon}_{\boldsymbol{x}}(k)}{\boldsymbol{\varepsilon}_{\boldsymbol{y}}(k)}\right)
\end{aligned} \tag{4.12}$$

Cela permet de récrire l'équation (4.11) en faisant clairement apparaître une structure de *State-Space*, avec $\boldsymbol{\varepsilon}$ comme signal d'entrée, $\boldsymbol{x}_{\Delta op}$ comme signal d'état, et $\boldsymbol{y}_{\Delta op}$ comme signal de sortie :

$$\left\{ \begin{array}{rcl} \boldsymbol{x}_{\Delta op}(k+1) & = & \boldsymbol{A}^*\boldsymbol{x}_{\Delta op}(k) + \left(\, \boldsymbol{I} \mid \boldsymbol{0} \,\right) \boldsymbol{\varepsilon}(k) \\ \boldsymbol{y}_{\Delta op}(k) & = & \boldsymbol{C}^*\boldsymbol{x}_{\Delta op}(k) + \left(\, \boldsymbol{0} \mid \boldsymbol{I} \,\right) \boldsymbol{\varepsilon}(k) \end{array} \right. \tag{4.13}$$

On appelle alors *filtre d'erreurs sur les opérations* et on note $\mathscr{H}_{\Delta op}$ le filtre modèle correspondant au *State-Space* :

$$\boldsymbol{Z}_{\Delta op} = \left(\begin{array}{c|c:c} \boldsymbol{A}^* & \boldsymbol{I} & \boldsymbol{0} \\ \hline \boldsymbol{C}^* & \boldsymbol{0} & \boldsymbol{I} \end{array}\right) \tag{4.14}$$

L'équation (4.13) signifie que l'erreur $\boldsymbol{y}_{\Delta op}$ due aux opérations est la sortie du filtre d'erreurs sur les opérations, quand on lui donne en entrée le signal $\boldsymbol{\varepsilon}$ des erreurs locales dans les opérations :

$$\boldsymbol{y}_{\Delta op} = \mathscr{H}_{\Delta op}\{\boldsymbol{\varepsilon}\} \tag{4.15}$$

4.1.2 Filtre d'erreurs sur les coefficients

On s'intéresse maintenant à l'erreur $\boldsymbol{y}_{\Delta c} = \boldsymbol{y}_c - \boldsymbol{y}$ due aux arrondis dans les matrices de coefficients. On soustrait cette fois les équations (4.5) et (3.3) définissant $\mathscr{H}_c$ et $\mathscr{H}$ respectivement :

$$\left\{\begin{array}{rcl} \boldsymbol{x}_c(k+1) - \boldsymbol{x}(k+1) & = & \boldsymbol{A}^*\boldsymbol{x}_c(k) - \boldsymbol{A}\boldsymbol{x}(k) + (\boldsymbol{B}^* - \boldsymbol{B})\boldsymbol{u}(k) \\ \boldsymbol{y}_c(k) - \boldsymbol{y}(k) & = & \boldsymbol{C}^*\boldsymbol{x}_c(k) - \boldsymbol{C}\boldsymbol{x}(k) + (\boldsymbol{D}^* - \boldsymbol{D})\boldsymbol{u}(k) \end{array}\right. \tag{4.16}$$

Encore une fois, on veut faire apparaître une structure de *State-Space* dans cette équation. Mais cela va demander plus de travail que précédemment, car il n'y a pas immédiatement de terme en $\boldsymbol{x}_c(k) - \boldsymbol{x}(k)$ dans les membres droits des égalités.

Par ailleurs, on voit des matrices comme $\boldsymbol{B}^* - \boldsymbol{B}$ qui, comme expliqué au début de la section 4.1, sont des matrices d'erreurs locales puisque chaque coefficient résulte d'un seul arrondi. On pose donc :

$$\begin{array}{ll} \boldsymbol{\Delta A} \triangleq \boldsymbol{A}^* - \boldsymbol{A} & \boldsymbol{\Delta B} \triangleq \boldsymbol{B}^* - \boldsymbol{B} \\ \boldsymbol{\Delta C} \triangleq \boldsymbol{C}^* - \boldsymbol{C} & \boldsymbol{\Delta D} \triangleq \boldsymbol{D}^* - \boldsymbol{D} \end{array} \tag{4.17}$$

On remarque alors qu'on peut faire apparaître à la fois $\boldsymbol{x}_c(k) - \boldsymbol{x}(k)$ et $\boldsymbol{\Delta A}$ en écrivant :

$$\begin{array}{rcl} \boldsymbol{A}^*\boldsymbol{x}_c(k) - \boldsymbol{A}\boldsymbol{x}(k) & = & \boldsymbol{A}^*\boldsymbol{x}_c(k) - \boldsymbol{A}^*\boldsymbol{x}(k) + \boldsymbol{A}^*\boldsymbol{x}(k) - \boldsymbol{A}\boldsymbol{x}(k) \\ & = & \boldsymbol{A}^*(\boldsymbol{x}_c(k) - \boldsymbol{x}(k)) + \boldsymbol{\Delta A}\boldsymbol{x}(k) \end{array} \tag{4.18}$$

On peut donc récrire l'équation (4.16) en :

$$\left\{\begin{array}{rcl} \boldsymbol{x}_c(k+1) - \boldsymbol{x}(k+1) & = & \boldsymbol{A}^*(\boldsymbol{x}_c(k) - \boldsymbol{x}(k)) + \boldsymbol{\Delta A}\boldsymbol{x}(k) + \boldsymbol{\Delta B}\boldsymbol{u}(k) \\ \boldsymbol{y}_c(k) - \boldsymbol{y}(k) & = & \boldsymbol{C}^*(\boldsymbol{x}_c(k) - \boldsymbol{x}(k)) + \boldsymbol{\Delta C}\boldsymbol{x}(k) + \boldsymbol{\Delta D}\boldsymbol{u}(k) \end{array}\right. \tag{4.19}$$

On a progressé, mais il reste des termes problématiques en $\boldsymbol{x}(k)$ tout seul. On pourrait dire que ceci est une structure de *State-Space* pour un filtre d'entrée $\begin{pmatrix} \boldsymbol{x}(k) \\ \hdashline \boldsymbol{u}(k) \end{pmatrix}$. Cependant, on ne souhaite pas le faire, car on ne veut pas que l'erreur étudiée $\boldsymbol{y}_{\Delta c}(k) = \boldsymbol{y}_c(k) - \boldsymbol{y}(k)$ dépende explicitement du signal $\boldsymbol{x}$, qu'on connaît seulement par sa définition récursive dans l'équation (3.3). Puisqu'on a décidé de ne pas rattacher ces termes en $\boldsymbol{x}(k)$ à l'entrée, il ne reste plus qu'à les rattacher au signal d'état de la structure de *State-Space* qu'on essaie de faire apparaître. Pour cela, on pose :

$$\boldsymbol{x}_{\Delta c} \triangleq \left(\begin{array}{c} \boldsymbol{x} \\ \hline \boldsymbol{x}_c - \boldsymbol{x} \end{array}\right) \tag{4.20}$$

En effet, $\boldsymbol{x}_{\Delta c}(k+1)$ est bien une combinaison linéaire de $\boldsymbol{x}_{\Delta c}(k)$ et $\boldsymbol{u}(k)$, en utilisant l'équation (3.3) pour décomposer $\boldsymbol{x}(k+1)$:

$$\begin{aligned}\boldsymbol{x}_{\Delta c}(k+1) &= \left(\frac{\boldsymbol{A}\boldsymbol{x}(k)+\boldsymbol{B}\boldsymbol{u}(k)}{\boldsymbol{A}^*(\boldsymbol{x}_c(k)-\boldsymbol{x}(k))+\boldsymbol{\Delta A}\boldsymbol{x}(k)+\boldsymbol{\Delta B}\boldsymbol{u}(k)}\right) \\ &= \left(\begin{array}{c|c}\boldsymbol{A} & \boldsymbol{0}\\ \hline \boldsymbol{\Delta A} & \boldsymbol{A}^*\end{array}\right)\left(\frac{\boldsymbol{x}(k)}{\boldsymbol{x}_c(k)-\boldsymbol{x}(k)}\right)+\left(\frac{\boldsymbol{B}}{\boldsymbol{\Delta B}}\right)\boldsymbol{u}(k)\end{aligned} \tag{4.21}$$

On a enfin obtenu une structure de *State-Space*, d'entrée $\boldsymbol{u}$, d'état $\boldsymbol{x}_{\Delta c}$ et de sortie $\boldsymbol{y}_{\Delta c}$:

$$\left\{\begin{array}{rcl}\boldsymbol{x}_{\Delta c}(k+1) &=& \left(\begin{array}{c|c}\boldsymbol{A} & \boldsymbol{0}\\ \hline \boldsymbol{\Delta A} & \boldsymbol{A}^*\end{array}\right)\boldsymbol{x}_{\Delta c}(k)+\left(\frac{\boldsymbol{B}}{\boldsymbol{\Delta B}}\right)\boldsymbol{u}(k)\\ \boldsymbol{y}_{\Delta c}(k) &=& \left(\boldsymbol{\Delta C} \mid \boldsymbol{C}^*\right)\boldsymbol{x}_{\Delta c}(k)+\left(\boldsymbol{\Delta D}\right)\boldsymbol{u}(k)\end{array}\right. \tag{4.22}$$

On appelle alors *filtre d'erreurs sur les coefficients*, et on note $\mathscr{H}_{\Delta c}$, le filtre modèle correspondant au *State-Space* :

$$\boldsymbol{Z}_{\Delta c} \triangleq \left(\begin{array}{c:c|c}\boldsymbol{A} & \boldsymbol{0} & \boldsymbol{B}\\ \hdashline \boldsymbol{\Delta A} & \boldsymbol{A}^* & \boldsymbol{\Delta B}\\ \hline \boldsymbol{\Delta C} & \boldsymbol{C}^* & \boldsymbol{\Delta D}\end{array}\right) \tag{4.23}$$

On a donc montré que l'erreur $\boldsymbol{y}_{\Delta c}$ due aux arrondis des coefficients est la sortie associée à l'entrée $\boldsymbol{u}$ à travers le filtre d'erreurs sur les coefficients :

$$\boldsymbol{y}_{\Delta c} = \mathscr{H}_{\Delta c}\{\boldsymbol{u}\} \tag{4.24}$$

4.1.3 Vue d'ensemble

En combinant les sections 4.1.1 et 4.1.2, on obtient le théorème suivant.

Théorème 4.1 (Filtres d'erreurs pour le *State-Space*). *Soit* $(\boldsymbol{A},\boldsymbol{B},\boldsymbol{C},\boldsymbol{D})$ *un State-Space. L'erreur finale entre les sorties des filtres modèle et implémenté pour un signal d'entrée* $\boldsymbol{u}$ *est :*

$$\boldsymbol{\Delta y} = \mathscr{H}_{\Delta op}\{\boldsymbol{\varepsilon}\} + \mathscr{H}_{\Delta c}\{\boldsymbol{u}\} \tag{4.25}$$

où $\mathscr{H}_{\Delta op}$ *est le* filtre d'erreurs sur les opérations [89, 31]*, défini par le State-Space :*

$$\boldsymbol{Z}_{\Delta op} = \left(\begin{array}{c|c:c}\boldsymbol{A}^* & \boldsymbol{I} & \boldsymbol{0}\\ \hline \boldsymbol{C}^* & \boldsymbol{0} & \boldsymbol{I}\end{array}\right) \qquad \text{(rappel de (4.14))}$$

et $\mathscr{H}_{\Delta c}$ *est le* filtre d'erreurs sur les coefficients*, défini par le State-Space :*

$$\boldsymbol{Z}_{\Delta c} \triangleq \left(\begin{array}{c:c|c}\boldsymbol{A} & \boldsymbol{0} & \boldsymbol{B}\\ \hdashline \boldsymbol{\Delta A} & \boldsymbol{A}^* & \boldsymbol{\Delta B}\\ \hline \boldsymbol{\Delta C} & \boldsymbol{C}^* & \boldsymbol{\Delta D}\end{array}\right) \qquad \text{(rappel de (4.23))}$$

et $\boldsymbol{\varepsilon}$ *est le signal des erreurs locales dans les sommes de produits :*

$$\begin{aligned}\boldsymbol{\varepsilon_x}(k) &\triangleq \left(\boldsymbol{A}^* \mid \boldsymbol{B}^*\right)\odot\left(\frac{\boldsymbol{x}^*(k)}{\boldsymbol{u}(k)}\right)-\left(\boldsymbol{A}^* \mid \boldsymbol{B}^*\right)\cdot\left(\frac{\boldsymbol{x}^*(k)}{\boldsymbol{u}(k)}\right)\\ \boldsymbol{\varepsilon_y}(k) &\triangleq \left(\boldsymbol{C}^* \mid \boldsymbol{D}^*\right)\odot\left(\frac{\boldsymbol{x}^*(k)}{\boldsymbol{u}(k)}\right)-\left(\boldsymbol{C}^* \mid \boldsymbol{D}^*\right)\cdot\left(\frac{\boldsymbol{x}^*(k)}{\boldsymbol{u}(k)}\right)\end{aligned} \qquad \text{(rappel de (4.9))}$$

$$\boldsymbol{\varepsilon}(k) \triangleq \left(\frac{\boldsymbol{\varepsilon_x}(k)}{\boldsymbol{\varepsilon_y}(k)}\right) \qquad \text{(rappel de (4.12))}$$

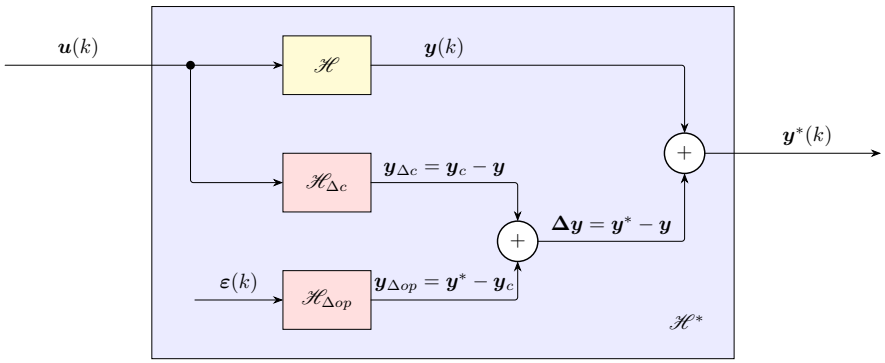


FIGURE 4.1 – Les filtres d'erreurs $\mathscr{H}_{\Delta op}$ et $\mathscr{H}_{\Delta c}$ caractérisent l'erreur finale $\boldsymbol{\Delta y}$

La figure 4.1 illustre les filtres d'erreurs $\mathscr{H}_{\Delta op}$ et $\mathscr{H}_{\Delta c}$ avec leurs entrées et sorties, qui permettent de caractériser l'erreur finale $\boldsymbol{\Delta y}$ entre les sorties respectives du filtre modèle $\mathscr{H}$ et du filtre implémenté $\mathscr{H}^*$.

Rappelons que l'objectif est de borner $\boldsymbol{\Delta y} = \mathscr{H}_{\Delta op}\{\boldsymbol{\varepsilon}\} + \mathscr{H}_{\Delta c}\{\boldsymbol{u}\}$. Les coefficients de $\mathscr{H}_{\Delta op}$ et $\mathscr{H}_{\Delta c}$ sont connus en fonction des coefficients $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ de $\mathscr{H}$ et de leurs arrondis $(\boldsymbol{A}^*, \boldsymbol{B}^*, \boldsymbol{C}^*, \boldsymbol{D}^*)$. Les entrées $\boldsymbol{u}$ qu'on considère en pratique sont bornées, et les erreurs locales $\boldsymbol{\varepsilon}$ issues de sommes de produits sont bornées quand on utilise des algorithmes appropriés comme on verra dans le chapitre 5. Ainsi, pour borner $\boldsymbol{\Delta y}$, il suffit de savoir borner la sortie d'un filtre de coefficients connus en fonction d'une borne sur l'entrée fournie : c'est ce que permettra le théorème du *Worst-Case Peak Gain* en section 4.3.

4.1.4 Formalisation

En Coq, on veut prouver le théorème 4.1 sur les filtres d'erreurs du *State-Space* :

$$\boldsymbol{\Delta y} = \mathscr{H}_{\Delta op}\{\boldsymbol{\varepsilon}\} + \mathscr{H}_{\Delta c}\{\boldsymbol{u}\} \qquad \text{(rappel de (4.25))}$$

Comme au début de la section 4.1, on se donne comme paramètres un *State-Space* $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ qui définit le filtre modèle $\mathscr{H}$, ainsi qu'un signal d'entrée $\boldsymbol{u}$.

```
Variable stsp : @StateSpace nu ny nx.
Notation A := (StSp_A stsp). (* etc. : B, C, D *)
Definition H : filter := filter_from_StSp stsp.

Variable u : vsignal nu.
```

On pourrait aussi prendre en paramètre une arithmétique en précision finie, comme on l'a fait au début du chapitre pour pouvoir présenter les définitions et propriétés mathématiques. On pourrait alors définir le filtre implémenté $\mathscr{H}^*$ comme dans l'équation (4.2), en utilisant les fonctions d'arrondi et de somme de produits associées à cette arithmétique. Mais c'est inutilement compliqué. À la place, on peut directement prendre en paramètres les erreurs locales des opérations $\boldsymbol{\varepsilon_x}$ et $\boldsymbol{\varepsilon_y}$, ainsi que les matrices des coefficients arrondis $(\boldsymbol{A}^*, \boldsymbol{B}^*, \boldsymbol{C}^*, \boldsymbol{D}^*)$ (en remplaçant le symbole « * » qu'on utilise pour signaler une valeur approchée par « ' » dans les identifiants en Coq).

```
Variable eps_x : vsignal nx.
Variable eps_y : vsignal ny.
Definition eps := col_signal eps_x eps_y.

Variable A' : 'M[RT]_nx.
Variable B' : 'M[RT]_(nx, nu).
Variable C' : 'M[RT]_(ny, nx).
Variable D' : 'M[RT]_(ny, nu).
```

En effet, on n'a pas besoin de savoir que ces valeurs viennent d'erreurs locales ou d'arrondis pour prouver l'équation (4.25). L'intérêt des paramètres choisis est qu'on prouve ici le théorème 4.1 sans dépendre d'une formalisation donnée des arithmétiques en précision finie. On n'a pas non plus besoin de préciser les formats et modes d'arrondi des calculs et quantifications de coefficients.

Il faudra plus tard exprimer les matrices arrondies (`A'` etc.) et erreurs locales de sommes de produits (`eps_x` et `eps_y`) en fonction de formats de virgules fixes et modes d'arrondis donnés (qui peuvent être différents pour chaque calcul ou coefficient) et de l'algorithme de somme de produits choisi parmi ceux qui sont formalisés dans le chapitre 5. Ceci est laissé en perspective en section 6.3.1 (même s'il s'agira surtout de le faire sur la SIF plutôt que le *State-Space*).

On peut alors définir le signal d'état $\boldsymbol{x}^*$ et la sortie $\boldsymbol{y}^*$ du filtre implémenté en passant par l'équation (4.10). Pour $\boldsymbol{x}^*$, on utilise encore une fois la construction récursive de la section 2.1.5.

```
Definition x'_fIS k (x'k : 'cV_nx) :=
  A' *m x'k + B' *m u k + eps_x k.
Definition x' := signal_peanorec x'_fIS.

Fact y'_causal :
  causal (fun k ⇒ C' *m x' k + D' *m u k + eps_y k).
Definition y' := Build_signal y'_causal.
```

Par ailleurs, on définit les filtres d'erreurs $\mathscr{H}_{\Delta op}$ et $\mathscr{H}_{\Delta c}$ à partir des équations (4.14) et (4.23).

```
Definition B_Dop : 'M[RT]_(nx, nx + ny) := row_mx 1%:M 0.
Definition D_Dop : 'M[RT]_(ny, nx + ny) := row_mx 0 1%:M.
Definition stsp_Dop := Build_StateSpace A' B_Dop C' D_Dop.
Definition H_Dop : filter := filter_from_StSp stsp_Dop.

Definition A_Dc : 'M_(nx + nx) :=
  col_mx (row_mx A 0) (row_mx (A'-A) A').
Definition B_Dc : 'M_(nx + nx, nu) := col_mx B (B'-B).
Definition C_Dc : 'M_(ny, nx + nx) := row_mx (C'-C) C'.
Definition D_Dc : 'M_(ny, nu) := D'-D.
Definition stsp_Dc := Build_StateSpace A_Dc B_Dc C_Dc D_Dc.
Definition H_Dc : filter := filter_from_StSp stsp_Dc.
```

On peut alors prouver le théorème 4.1.

```
Theorem error_filters_StSp : y' - H u = H_Dop eps + H_Dc u.
```

La preuve consiste surtout à développer des produits matriciels, et identifier des termes égaux de chaque côté des égalités. Ce n'est pas difficile, mais c'est parfois fastidieux. Cette preuve serait grandement simplifiée par une tactique similaire à `ring` mais fonctionnant sur des matrices de taille quelconque, qui ne forment pas un anneau (voir la section 6.2.4).

Rappelons que l'objectif de ce chapitre est d'analyser les erreurs d'arrondis dans la SIF, pas le *State-Space*. Le cas du *State-Space* a été présenté dans un intérêt purement didactique. On aurait pu s'attendre à ce que le théorème ci-dessus soit au moins utilisé pour prouver un théorème correspondant sur la SIF, comme dans la section 3.4. Mais ce n'est pas le cas ici. Le contenu de cette section pourrait donc être enlevé de la formalisation sans affecter le reste. Néanmoins, même si les preuves sur les filtres d'erreurs pour le *State-Space* et pour la SIF sont techniquement indépendantes, elles sont suffisamment similaires pour que je ne regrette pas d'avoir pu m'échauffer sur le cas du *State-Space* avant d'aborder celui de la SIF.

4.2 Filtres d'erreurs d'une SIF

Cette section consiste à refaire l'analyse de la section 4.1 dans le cas de la SIF, qui est le cas qui nous intéresse vraiment. On va donc à nouveau décrire l'erreur finale, affectée par la propagation des erreurs, en fonction de filtres d'erreurs et d'erreurs locales.

Soit $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ une SIF et $\boldsymbol{u}$ un signal d'entrée de la bonne taille. On utilise les mêmes notations qu'en section 4.1 : $\mathscr{H}$ est le filtre modèle, de sortie $\boldsymbol{y}$, signal d'état $\boldsymbol{x}$ et signal auxiliaire $\boldsymbol{t}$; les signaux correspondants calculés en précision finie sont $\boldsymbol{y}^*$, $\boldsymbol{x}^*$ et $\boldsymbol{t}^*$, et ils définissent le filtre implémenté $\mathscr{H}^*$.

On a vu en section 3.2.2 que le filtre modèle $\mathscr{H}$ est défini par :

$$\mathscr{H} \quad \left\{ \begin{array}{rcl} \boldsymbol{J}\boldsymbol{t}(k+1) & = & \boldsymbol{M}\boldsymbol{x}(k) + \boldsymbol{N}\boldsymbol{u}(k) \\ \boldsymbol{x}(k+1) & = & \boldsymbol{K}\boldsymbol{t}(k+1) + \boldsymbol{P}\boldsymbol{x}(k) + \boldsymbol{Q}\boldsymbol{u}(k) \\ \boldsymbol{y}(k) & = & \boldsymbol{L}\boldsymbol{t}(k+1) + \boldsymbol{R}\boldsymbol{x}(k) + \boldsymbol{S}\boldsymbol{u}(k) \end{array} \right. \qquad \text{(rappel de (3.25))}$$

et le filtre implémenté $\mathscr{H}^*$ par :

$$\mathscr{H}^* \quad \left\{ \begin{array}{rcl} \boldsymbol{t}^*(k+1) & = & \left(\circ(\boldsymbol{I}-\boldsymbol{J}) \;\middle|\; \circ(\boldsymbol{M}) \;\middle|\; \circ(\boldsymbol{N}) \right) \odot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array} \right) \\ \boldsymbol{x}^*(k+1) & = & \left(\circ(\boldsymbol{K}) \;\middle|\; \circ(\boldsymbol{P}) \;\middle|\; \circ(\boldsymbol{Q}) \right) \odot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array} \right) \\ \boldsymbol{y}^*(k) & = & \left(\circ(\boldsymbol{L}) \;\middle|\; \circ(\boldsymbol{R}) \;\middle|\; \circ(\boldsymbol{S}) \right) \odot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array} \right) \end{array} \right. \qquad \text{(rappel de (3.33))}$$

Comme dans la section 4.1, on introduit des notations pour les matrices arrondies de coefficients :

$$\begin{array}{lll} \boldsymbol{J}^* \triangleq \boldsymbol{I} - \bigcirc(\boldsymbol{I}-\boldsymbol{J}) & \boldsymbol{M}^* \triangleq \bigcirc(\boldsymbol{M}) & \boldsymbol{N}^* \triangleq \bigcirc(\boldsymbol{N}) \\ \boldsymbol{K}^* \triangleq \bigcirc(\boldsymbol{K}) & \boldsymbol{P}^* \triangleq \bigcirc(\boldsymbol{P}) & \boldsymbol{Q}^* \triangleq \bigcirc(\boldsymbol{Q}) \\ \boldsymbol{L}^* \triangleq \bigcirc(\boldsymbol{L}) & \boldsymbol{R}^* \triangleq \bigcirc(\boldsymbol{R}) & \boldsymbol{S}^* \triangleq \bigcirc(\boldsymbol{S}) \end{array} \tag{4.26}$$

Le filtre implémenté se récrit alors :

$$\mathscr{H}^* \quad \left\{ \begin{array}{rcl} \boldsymbol{t}^*(k+1) & = & \left(\boldsymbol{I}-\boldsymbol{J}^* \mid \boldsymbol{M}^* \mid \boldsymbol{N}^*\right) \odot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array}\right) \\ \boldsymbol{x}^*(k+1) & = & \left(\boldsymbol{K}^* \mid \boldsymbol{P}^* \mid \boldsymbol{Q}^*\right) \odot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array}\right) \\ \boldsymbol{y}^*(k) & = & \left(\boldsymbol{L}^* \mid \boldsymbol{R}^* \mid \boldsymbol{S}^*\right) \odot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array}\right) \end{array} \right. \tag{4.27}$$

La définition de $\boldsymbol{J}^*$ est particulière car comme on a vu en section 3.2.2, ce ne sont pas les coefficients de $\boldsymbol{J}$ qui sont utilisés dans les calculs mais ceux de $\boldsymbol{I}-\boldsymbol{J}$. Comme $\boldsymbol{J}$ vérifie la propriété *Jprop* (triangulaire inférieure avec des 1 sur la diagonale) et qu'on a supposé au début du chapitre que 0 et -1 sont dans le format de précision finie considéré (en particulier ce sont des points fixes de $\bigcirc$), on a $\boldsymbol{J}^* \triangleq \boldsymbol{I} - \bigcirc(\boldsymbol{I}-\boldsymbol{J}) = -\bigcirc(-\boldsymbol{J})$. Notons que si on suppose aussi que l'arrondi considéré est une fonction paire ($\bigcirc(-x) = -\bigcirc(x)$), on obtient $\boldsymbol{J}^* = \bigcirc(\boldsymbol{J})$, mais ce n'est pas le cas de certains modes d'arrondis très communs comme l'arrondi vers le bas donc on ne fait pas cette hypothèse. Avoir $\boldsymbol{J}^* = -\bigcirc(-\boldsymbol{J})$ suffit à assurer que $\boldsymbol{J}^*$ vérifie aussi la propriété *Jprop* (en utilisant à nouveau que 0 et -1 sont dans le format).

On définit à nouveau $\mathscr{H}_c$ comme le filtre *modèle* associé à la SIF $(\boldsymbol{J}^*, \boldsymbol{K}^*, \boldsymbol{L}^*, \boldsymbol{M}^*, \boldsymbol{N}^*, \boldsymbol{P}^*, \boldsymbol{Q}^*, \boldsymbol{R}^*, \boldsymbol{S}^*)$, qui est une SIF valide car on vient de montrer que $\boldsymbol{J}^*$ vérifie bien *Jprop*. Comme en section 4.1, ce filtre prend en compte les arrondis des coefficients, mais pas les erreurs dues aux opérations calculées en précision finie. Les signaux associés sont notés $\boldsymbol{y}_c$, $\boldsymbol{x}_c$ et $\boldsymbol{t}_c$:

$$\mathscr{H}_c \quad \left\{ \begin{array}{rcl} \boldsymbol{J}^* \boldsymbol{t}_c(k+1) & = & \boldsymbol{M}^* \boldsymbol{x}_c(k) + \boldsymbol{N}^* \boldsymbol{u}(k) \\ \boldsymbol{x}_c(k+1) & = & \boldsymbol{K}^* \boldsymbol{t}_c(k+1) + \boldsymbol{P}^* \boldsymbol{x}_c(k) + \boldsymbol{Q}^* \boldsymbol{u}(k) \\ \boldsymbol{y}_c(k) & = & \boldsymbol{L}^* \boldsymbol{t}_c(k+1) + \boldsymbol{R}^* \boldsymbol{x}_c(k) + \boldsymbol{S}^* \boldsymbol{u}(k) \end{array} \right. \tag{4.28}$$

On décompose à nouveau l'erreur finale $\boldsymbol{\Delta y} = \boldsymbol{y}^* - \boldsymbol{y}$ en :

$$\boldsymbol{\Delta y} = \boldsymbol{y}_{\Delta op} + \boldsymbol{y}_{\Delta c} \tag{4.29}$$

où :

$$\boldsymbol{y}_{\Delta op} \triangleq \boldsymbol{y}^* - \boldsymbol{y}_c \tag{4.30}$$

est l'erreur due aux *opérations* calculées en précision finie (qui sont toutes des sommes de produits), et :

$$\boldsymbol{y}_{\Delta c} \triangleq \boldsymbol{y}_c - \boldsymbol{y} \tag{4.31}$$

est l'erreur due aux arrondis des *coefficients*.

La section 4.2.1 exprime $\boldsymbol{y}_{\Delta op}$ grâce au filtre d'erreurs sur les opérations, et la section 4.2.2 exprime $\boldsymbol{y}_{\Delta c}$ grâce au filtre d'erreurs sur les coefficients.

4.2.1 Filtre d'erreurs sur les opérations

Comme dans la section 4.1.1, on définit les vecteurs d'erreurs locales dans les sommes de produits :

$$\begin{aligned}
\boldsymbol{\varepsilon_t}(k) &\triangleq \left(\boldsymbol{I}-\boldsymbol{J}^* \mid \boldsymbol{M}^* \mid \boldsymbol{N}^*\right) \odot \left(\begin{array}{c}\boldsymbol{t}^*(k+1)\\ \hline \boldsymbol{x}^*(k)\\ \hline \boldsymbol{u}(k)\end{array}\right) - \left(\boldsymbol{I}-\boldsymbol{J}^* \mid \boldsymbol{M}^* \mid \boldsymbol{N}^*\right) \cdot \left(\begin{array}{c}\boldsymbol{t}^*(k+1)\\ \hline \boldsymbol{x}^*(k)\\ \hline \boldsymbol{u}(k)\end{array}\right)\\
\boldsymbol{\varepsilon_x}(k) &\triangleq \left(\boldsymbol{K}^* \mid \boldsymbol{P}^* \mid \boldsymbol{Q}^*\right) \odot \left(\begin{array}{c}\boldsymbol{t}^*(k+1)\\ \hline \boldsymbol{x}^*(k)\\ \hline \boldsymbol{u}(k)\end{array}\right) - \left(\boldsymbol{K}^* \mid \boldsymbol{P}^* \mid \boldsymbol{Q}^*\right) \cdot \left(\begin{array}{c}\boldsymbol{t}^*(k+1)\\ \hline \boldsymbol{x}^*(k)\\ \hline \boldsymbol{u}(k)\end{array}\right)\\
\boldsymbol{\varepsilon_y}(k) &\triangleq \left(\boldsymbol{L}^* \mid \boldsymbol{R}^* \mid \boldsymbol{S}^*\right) \odot \left(\begin{array}{c}\boldsymbol{t}^*(k+1)\\ \hline \boldsymbol{x}^*(k)\\ \hline \boldsymbol{u}(k)\end{array}\right) - \left(\boldsymbol{L}^* \mid \boldsymbol{R}^* \mid \boldsymbol{S}^*\right) \cdot \left(\begin{array}{c}\boldsymbol{t}^*(k+1)\\ \hline \boldsymbol{x}^*(k)\\ \hline \boldsymbol{u}(k)\end{array}\right)\\
\boldsymbol{\varepsilon}(k) &\triangleq \left(\begin{array}{c}\boldsymbol{\varepsilon_t}(k)\\ \hline \boldsymbol{\varepsilon_x}(k)\\ \hline \boldsymbol{\varepsilon_y}(k)\end{array}\right)
\end{aligned} \tag{4.32}$$

On peut alors récrire l'équation (4.27) ainsi :

$$\mathscr{H}^* \quad \left\{\begin{array}{rcl}\boldsymbol{J}^*\,\boldsymbol{t}^*(k+1) &=& \boldsymbol{M}^*\,\boldsymbol{x}^*(k)+\boldsymbol{N}^*\,\boldsymbol{u}(k)+\boldsymbol{\varepsilon_t}(k)\\ \boldsymbol{x}^*(k+1) &=& \boldsymbol{K}^*\,\boldsymbol{t}^*(k+1)+\boldsymbol{P}^*\,\boldsymbol{x}^*(k)+\boldsymbol{Q}^*\,\boldsymbol{u}(k)+\boldsymbol{\varepsilon_x}(k)\\ \boldsymbol{y}^*(k) &=& \boldsymbol{L}^*\,\boldsymbol{t}^*(k+1)+\boldsymbol{R}^*\,\boldsymbol{x}^*(k)+\boldsymbol{S}^*\,\boldsymbol{u}(k)+\boldsymbol{\varepsilon_y}(k)\end{array}\right. \tag{4.33}$$

La première ligne a une forme pratique grâce à la définition bien choisie de $\boldsymbol{J}^*$ dans l'équation (4.26).

Comme en section 4.1.1, on soustrait l'équation précédente et l'équation (4.28) pour obtenir :

$$\left\{\begin{array}{rcl}\boldsymbol{J}^*(\boldsymbol{t}^*-\boldsymbol{t}_c)(k+1) &=& \boldsymbol{M}^*(\boldsymbol{x}^*-\boldsymbol{x}_c)(k)+\boldsymbol{\varepsilon_t}(k)\\ (\boldsymbol{x}^*-\boldsymbol{x}_c)(k+1) &=& \boldsymbol{K}^*(\boldsymbol{t}^*-\boldsymbol{t}_c)(k+1)+\boldsymbol{P}^*(\boldsymbol{x}^*-\boldsymbol{x}_c)(k)+\boldsymbol{\varepsilon_x}(k)\\ (\boldsymbol{y}^*-\boldsymbol{y}_c)(k) &=& \boldsymbol{L}^*(\boldsymbol{t}^*-\boldsymbol{t}_c)(k+1)+\boldsymbol{R}^*(\boldsymbol{x}^*-\boldsymbol{x}_c)(k)+\boldsymbol{\varepsilon_y}(k)\end{array}\right. \tag{4.34}$$

On reconnaît la relation décrivant le filtre correspondant à une SIF, où l'entrée est $\boldsymbol{\varepsilon}$, le signal intermédiaire est $\boldsymbol{t}^*-\boldsymbol{t}_c$, le signal d'état est $\boldsymbol{x}^*-xc$, et la sortie est $\boldsymbol{y}^*-\boldsymbol{y}_c=\boldsymbol{y}_{\Delta op}$. Le *filtre d'erreurs sur les opérations* $\mathscr{H}_{\Delta op}$ tel que :

$$\boldsymbol{y}_{\Delta op}=\mathscr{H}_{\Delta op}\{\boldsymbol{\varepsilon}\} \tag{4.35}$$

est donc le filtre modèle correspondant à la SIF :

$$\boldsymbol{Z}_{\Delta op}=\left(\begin{array}{c|c|c:c:c}-\boldsymbol{J}^* & \boldsymbol{M}^* & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0}\\ \hline \boldsymbol{K}^* & \boldsymbol{P}^* & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0}\\ \hline \boldsymbol{L}^* & \boldsymbol{R}^* & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I}\end{array}\right) \tag{4.36}$$

4.2.2 Filtre d'erreurs sur les coefficients

Comme en section 4.1.2, on pose les matrices d'erreurs locales sur les coefficients qui doivent être arrondis :

$$\begin{array}{ccc} \boldsymbol{\Delta J} \triangleq \boldsymbol{J}^* - \boldsymbol{J} & \boldsymbol{\Delta M} \triangleq \boldsymbol{M}^* - \boldsymbol{M} & \boldsymbol{\Delta N} \triangleq \boldsymbol{N}^* - \boldsymbol{N} \\ \boldsymbol{\Delta K} \triangleq \boldsymbol{K}^* - \boldsymbol{K} & \boldsymbol{\Delta P} \triangleq \boldsymbol{P}^* - \boldsymbol{P} & \boldsymbol{\Delta Q} \triangleq \boldsymbol{Q}^* - \boldsymbol{Q} \\ \boldsymbol{\Delta L} \triangleq \boldsymbol{L}^* - \boldsymbol{L} & \boldsymbol{\Delta R} \triangleq \boldsymbol{R}^* - \boldsymbol{R} & \boldsymbol{\Delta S} \triangleq \boldsymbol{S}^* - \boldsymbol{S} \end{array} \tag{4.37}$$

On soustrait les équations (4.28) et (3.25) définissant $\mathscr{H}_c$ et $\mathscr{H}$ respectivement, en faisant apparaître les matrices d'erreurs ci-dessus grâce à la technique de l'équation (4.18) :

$$\begin{cases} \boldsymbol{J}^*(\boldsymbol{t}_c - \boldsymbol{t})(k+1) + \boldsymbol{\Delta J}\boldsymbol{t}(k+1) = \boldsymbol{M}^*(\boldsymbol{x}_c - \boldsymbol{x})(k) + \boldsymbol{\Delta M}\boldsymbol{x}(k) + \boldsymbol{\Delta N}\boldsymbol{u}(k) \\ (\boldsymbol{x}_c - \boldsymbol{x})(k+1) = \boldsymbol{K}^*(\boldsymbol{t}_c - \boldsymbol{t})(k+1) + \boldsymbol{\Delta K}\boldsymbol{t}(k+1) + \boldsymbol{P}^*(\boldsymbol{x}_c - \boldsymbol{x})(k) + \boldsymbol{\Delta P}\boldsymbol{x}(k) + \boldsymbol{\Delta Q}\boldsymbol{u}(k) \\ (\boldsymbol{y}_c - \boldsymbol{y})(k) = \boldsymbol{L}^*(\boldsymbol{t}_c - \boldsymbol{t})(k+1) + \boldsymbol{\Delta L}\boldsymbol{t}(k+1) + \boldsymbol{R}^*(\boldsymbol{x}_c - \boldsymbol{x})(k) + \boldsymbol{\Delta R}\boldsymbol{x}(k) + \boldsymbol{\Delta S}\boldsymbol{u}(k) \end{cases} \tag{4.38}$$

Afin de faire apparaître une structure de SIF, on rappelle que $\boldsymbol{y}_{\Delta c} = \boldsymbol{y}_c - \boldsymbol{y}$, et on pose le signal auxiliaire et le signal d'état suivants :

$$\boldsymbol{t}_{\Delta c} \triangleq \left(\begin{array}{c} \boldsymbol{t} \\ \hline \boldsymbol{t}_c - \boldsymbol{t} \end{array}\right) \qquad \boldsymbol{x}_{\Delta c} \triangleq \left(\begin{array}{c} \boldsymbol{x} \\ \hline \boldsymbol{x}_c - \boldsymbol{x} \end{array}\right) \tag{4.39}$$

En utilisant l'équation (3.25) pour les sous-matrices supérieures et l'équation (4.38) pour les sous-matrices inférieures :

$$\begin{cases} \left(\begin{array}{c|c} \boldsymbol{J} & \boldsymbol{0} \\ \hline \boldsymbol{\Delta J} & \boldsymbol{J}^* \end{array}\right) \boldsymbol{t}_{\Delta c}(k+1) = \left(\begin{array}{c|c} \boldsymbol{M} & \boldsymbol{0} \\ \hline \boldsymbol{\Delta M} & \boldsymbol{M}^* \end{array}\right) \boldsymbol{x}_{\Delta c}(k) + \left(\begin{array}{c} \boldsymbol{N} \\ \hline \boldsymbol{\Delta N} \end{array}\right) \boldsymbol{u}(k) \\ \boldsymbol{x}_{\Delta c}(k+1) = \left(\begin{array}{c|c} \boldsymbol{K} & \boldsymbol{0} \\ \hline \boldsymbol{\Delta K} & \boldsymbol{K}^* \end{array}\right) \boldsymbol{t}_{\Delta c}(k+1) + \left(\begin{array}{c|c} \boldsymbol{P} & \boldsymbol{0} \\ \hline \boldsymbol{\Delta P} & \boldsymbol{P}^* \end{array}\right) \boldsymbol{x}_{\Delta c}(k) + \left(\begin{array}{c} \boldsymbol{Q} \\ \hline \boldsymbol{\Delta Q} \end{array}\right) \boldsymbol{u}(k) \\ \boldsymbol{y}_{\Delta c}(k) = \left(\begin{array}{c|c} \boldsymbol{\Delta L} & \boldsymbol{L}^* \end{array}\right) \boldsymbol{t}_{\Delta c}(k+1) + \left(\begin{array}{c|c} \boldsymbol{\Delta R} & \boldsymbol{R}^* \end{array}\right) \boldsymbol{x}_{\Delta c}(k) + \left(\boldsymbol{\Delta S}\right) \boldsymbol{u}(k) \end{cases} \tag{4.40}$$

Le *filtre d'erreurs sur les coefficients* $\mathscr{H}_{\Delta c}$ tel que :

$$\boldsymbol{y}_{\Delta c} = \mathscr{H}_{\Delta c}\{\boldsymbol{u}\} \tag{4.41}$$

est donc le filtre modèle correspondant à la SIF :

$$\boldsymbol{Z}_{\Delta c} \triangleq \left(\begin{array}{c:c|c:c|c} -\boldsymbol{J} & \boldsymbol{0} & \boldsymbol{M} & \boldsymbol{0} & \boldsymbol{N} \\ \hdashline -\boldsymbol{\Delta J} & -\boldsymbol{J}^* & \boldsymbol{\Delta M} & \boldsymbol{M}^* & \boldsymbol{\Delta N} \\ \hline \boldsymbol{K} & \boldsymbol{0} & \boldsymbol{P} & \boldsymbol{0} & \boldsymbol{Q} \\ \hdashline \boldsymbol{\Delta K} & \boldsymbol{K}^* & \boldsymbol{\Delta P} & \boldsymbol{P}^* & \boldsymbol{\Delta Q} \\ \hline \boldsymbol{\Delta L} & \boldsymbol{L}^* & \boldsymbol{\Delta R} & \boldsymbol{R}^* & \boldsymbol{\Delta S} \end{array}\right) \tag{4.42}$$

4.2.3 Vue d'ensemble

En combinant les sections 4.2.1 et 4.2.2, on obtient le théorème suivant, similaire à celui de la section 4.1.3.

Théorème 4.2 (Filtres d'erreurs pour la SIF). *Soit* $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ *une SIF. L'erreur finale entre les sorties des filtres modèle et implémenté pour un signal d'entrée* $\boldsymbol{u}$ *est :*

$$\boldsymbol{\Delta y} = \mathscr{H}_{\Delta op}\{\boldsymbol{\varepsilon}\} + \mathscr{H}_{\Delta c}\{\boldsymbol{u}\} \tag{4.43}$$

où $\mathscr{H}_{\Delta op}$ *est le* filtre d'erreurs sur les opérations [59]*, défini par la SIF :*

$$\boldsymbol{Z}_{\Delta op} = \left(\begin{array}{c|c|c:c:c} -\boldsymbol{J}^* & \boldsymbol{M}^* & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \\ \hline \boldsymbol{K}^* & \boldsymbol{P}^* & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} \\ \hline \boldsymbol{L}^* & \boldsymbol{R}^* & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{array}\right) \qquad \text{(rappel de (4.36))}$$

et $\mathscr{H}_{\Delta c}$ *est le* filtre d'erreurs sur les coefficients [85]*, défini par la SIF :*

$$\boldsymbol{Z}_{\Delta c} \triangleq \left(\begin{array}{c:c|c:c|c} -\boldsymbol{J} & \boldsymbol{0} & \boldsymbol{M} & \boldsymbol{0} & \boldsymbol{N} \\ \hdashline -\boldsymbol{\Delta J} & -\boldsymbol{J}^* & \boldsymbol{\Delta M} & \boldsymbol{M}^* & \boldsymbol{\Delta N} \\ \hline \boldsymbol{K} & \boldsymbol{0} & \boldsymbol{P} & \boldsymbol{0} & \boldsymbol{Q} \\ \hdashline \boldsymbol{\Delta K} & \boldsymbol{K}^* & \boldsymbol{\Delta P} & \boldsymbol{P}^* & \boldsymbol{\Delta Q} \\ \hline \boldsymbol{\Delta L} & \boldsymbol{L}^* & \boldsymbol{\Delta R} & \boldsymbol{R}^* & \boldsymbol{\Delta S} \end{array}\right) \qquad \text{(rappel de (4.42))}$$

et $\boldsymbol{\varepsilon}$ *est le signal des erreurs locales dans les opérations :*

$$\boldsymbol{\varepsilon_t}(k) \triangleq \left(\boldsymbol{I} - \boldsymbol{J}^* \mid \boldsymbol{M}^* \mid \boldsymbol{N}^*\right) \odot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array}\right) - \left(\boldsymbol{I} - \boldsymbol{J}^* \mid \boldsymbol{M}^* \mid \boldsymbol{N}^*\right) \cdot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array}\right)$$

$$\boldsymbol{\varepsilon_x}(k) \triangleq \left(\boldsymbol{K}^* \mid \boldsymbol{P}^* \mid \boldsymbol{Q}^*\right) \odot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array}\right) - \left(\boldsymbol{K}^* \mid \boldsymbol{P}^* \mid \boldsymbol{Q}^*\right) \cdot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array}\right)$$

$$\boldsymbol{\varepsilon_y}(k) \triangleq \left(\boldsymbol{L}^* \mid \boldsymbol{R}^* \mid \boldsymbol{S}^*\right) \odot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array}\right) - \left(\boldsymbol{L}^* \mid \boldsymbol{R}^* \mid \boldsymbol{S}^*\right) \cdot \left(\begin{array}{c} \boldsymbol{t}^*(k+1) \\ \hline \boldsymbol{x}^*(k) \\ \hline \boldsymbol{u}(k) \end{array}\right)$$

$$\boldsymbol{\varepsilon}(k) \triangleq \left(\begin{array}{c} \boldsymbol{\varepsilon_t}(k) \\ \hline \boldsymbol{\varepsilon_x}(k) \\ \hline \boldsymbol{\varepsilon_y}(k) \end{array}\right)$$

(rappel de (4.32))

Ces filtres d'erreurs sont illustrés par la même figure 4.1 que pour le *State-Space*. Leur intérêt est expliqué en section 4.1.3. Encore une fois, cela va permettre de borner l'erreur finale $\boldsymbol{\Delta y}$ grâce au théorème du *Worst-Case Peak Gain* qui fera l'objet de la section 4.3.

4.2.4 Formalisation

On veut prouver le théorème 4.2 sur les filtres d'erreurs de la SIF. On se donne en paramètres une SIF qui définit le filtre modèle $\mathscr{H}$, ainsi qu'un signal d'entrée $\boldsymbol{u}$.

```
Variable sif : @SIF nu ny nx nt.
Notation J := (SIF_J sif). (* etc. : K, L, M, N, P, Q, R, S *)
Definition H : filter := filter_from_SIF sif.

Variable u : vsignal nu.
```

Comme expliqué en section 4.1.4, on prend directement en paramètres les erreurs locales dans les opérations $\boldsymbol{\varepsilon_t}$, $\boldsymbol{\varepsilon_x}$ et $\boldsymbol{\varepsilon_y}$, et les matrices $\boldsymbol{J}^*$, $\boldsymbol{K}^*$, etc. définies par l'équation (4.26). Encore une fois, cela évite de dépendre d'une formalisation particulière des arithmétiques en précision finie et de devoir préciser les formats des différents calculs.

```
Variable eps_t : vsignal nt.
Variable eps_x : vsignal nx.
Variable eps_y : vsignal ny.
Definition eps := (col_signal (col_signal eps_t eps_x) eps_y).

Variable J' : 'M[RT]_nt.
Hypothesis Jprop_J' : Jprop J'.
Variable K' : 'M[RT]_(nx, nt).
Variable L' : 'M[RT]_(ny, nt).
(* etc. M', N', P', Q', R', S' *)
```

On a vu au début de la section 4.2 que $\boldsymbol{J}^*$ vérifie bien la propriété *Jprop* car on a supposé que 0 et -1 sont dans le format de précision finie utilisé. Ici, comme `J'` est juste un paramètre, on a besoin d'ajouter l'hypothèse `Jprop_J'` ci-dessus.

On définit alors les signaux liés au filtre implémenté $\mathscr{H}^*$. La fonction `compute_t'S` construit le vecteur $\boldsymbol{t}^*(k+1)$ à partir de k et de $\boldsymbol{x}^*(k)$. Cela permet de définir le signal $\boldsymbol{x}^*$ récursivement, puis la sortie $\boldsymbol{y}^*$.

```
Definition compute_t'S k x'k :=
  invmx J' *m (M' *m x'k + N' *m u k + eps_t k).

Definition x'_fIS k valk :=
  K' *m compute_t'S k valk + P' *m valk + Q' *m u k + eps_x k.
Definition x' := signal_peanorec x'_fIS.

Fact y'_causal : causal (fun k ⇒
  L' *m compute_t'S k (x' k) + R' *m x' k + S' *m u k + eps_y k).
Definition y' := Build_signal y'_causal.
```

On définit aussi les filtres d'erreurs $\mathscr{H}_{\Delta op}$ et $\mathscr{H}_{\Delta c}$ à partir des équations (4.36) et (4.42). On remarque que la SIF décrivant $\mathscr{H}_{\Delta op}$ contient beaucoup de matrices de la forme $\left(\begin{array}{c|c} \boldsymbol{A} & \boldsymbol{0} \\ \hline \boldsymbol{\Delta A} & \boldsymbol{A}^* \end{array}\right)$, donc simplifie sa définition avec la notation `blockDop`. Il faut aussi montrer que la première matrice de coefficients de cette SIF vérifie bien la propriété *Jprop*. Pour la SIF décrivant $\mathscr{H}_{\Delta c}$, on a déjà l'hypothèse `Jprop_J'`.

```
Notation blockDop A A' := (block_mx      A    0
                                       (A'-A)  A' ).
Lemma Jprop_Dop : Jprop (blockDop J J').
Definition sif_Dc := Build_SIF
(* SIF_Jprop *)  Jprop_Dop
(* SIF_K *)      (blockDop K K')
(* SIF_L *)      (row_mx (L'-L) L')
(* SIF_M *)      (blockDop M M')
```

```
(* SIF_N *)       (col_mx N (N'-N))
(* SIF_P *)       (blockDop P P')
(* SIF_Q *)       (col_mx Q (Q'-Q))
(* SIF_R *)       (row_mx (R'-R) R')
(* SIF_S *)       (S'-S).
Definition H_Dc := filter_from_SIF sif_Dc.

Definition sif_Dop := Build_SIF
(* SIF_Jprop *)  Jprop_J'
(* SIF_K *)       K'
(* SIF_L *)       L'
(* SIF_M *)       M'
(* SIF_N *)       (row_mx (row_mx 1%:M 0) 0)
(* SIF_P *)       P'
(* SIF_Q *)       (row_mx (row_mx 0 1%:M) 0)
(* SIF_R *)       R'
(* SIF_S *)       (row_mx (row_mx 0 0) 1%:M).
Definition H_Dop : filter := filter_from_SIF sif_Dop.
```

Enfin, on prouve le théorème 4.2.

```
Theorem error_filters : y' - H u = H_Dop eps + H_Dc u.
```

Comme en section 4.1.4, la partie longue de la preuve consiste à reconnaître et simplifier des termes égaux dans des grandes sommes de produits matriciels. La différence est que les termes sont encore plus nombreux. Cette preuve bénéficierait donc encore plus d'une tactique similaire à `ring` mais pour des matrices de taille quelconque, comme discuté en section 6.2.4.

4.3 *Worst-Case Peak Gain*

Les filtres d'erreurs présentés dans les sections 4.1 et 4.2 permettent de caractériser l'erreur finale $\boldsymbol{\Delta y}$ sur la sortie comme la somme des sorties de ces filtres d'erreurs pour des entrées particulières. Pour pouvoir borner $\boldsymbol{\Delta y}$, il suffit donc de pouvoir borner la sortie d'un filtre connu pour une entrée donnée. C'est ce que permet le *théorème du Worst-Case Peak Gain* (initialement appelé *peak gain* ou ℓ^∞*-gain* comme expliqué plus loin) [23]. La section 4.3.1 énonce, interprète et formalise ce théorème. La section 4.3.2 prouve formellement que la borne donnée par le théorème est optimale. La section 4.3.3 montre en Coq une propriété encore plus forte : cette borne est atteinte pour un signal d'entrée soigneusement construit. La section 4.3.4 présente une application classique du *Worst-Case Peak Gain* : choisir les formats des variables d'un filtre afin d'assurer qu'il n'y aura pas d'*overflows* [29]. Enfin, la section 4.3.5 étudie l'application du *Worst-Case Peak Gain* aux filtres d'erreurs vus précédemment dans le but de borner l'erreur finale sur la sortie du filtre [58, 110].

Notons que dans cette section, l'anneau commutatif `RT` sur lequel on a travaillé jusqu'ici est instancié à l'ensemble des réels $\mathbb{R}$ (type `R` en Coq). En effet, on a besoin d'inégalités et de limites de suites. Pour ces dernières, on utilise la bibliothèque Coquelicot [18] (présentée en section 1.2.3). Certaines de ces limites sont potentiellement infinies donc sont définies dans l'adhérence de $\mathbb{R}$ notée $\overline{\mathbb{R}} = \mathbb{R} \cup \{\pm\infty\}$ (type `Rbar` en Coq).

4.3.1 Théorème du *Worst-Case Peak Gain*

Commençons par le cas d'un filtre SISO, plus facile à présenter. Le cas MIMO sera traité plus loin dans cette section.

Théorème 4.3 (Worst-Case Peak Gain, cas SISO). *Soit $\mathscr{H}$ un filtre LTI SISO. Soit u un signal d'entrée qui est borné par $\overline{u} \in \overline{\mathbb{R}}$:*

$$\forall k,\ |u(k)| \leq \overline{u} \tag{4.44}$$

Alors la sortie correspondante y est bornée par $\langle\!\langle \mathscr{H} \rangle\!\rangle\, \overline{u}$:

$$\forall k,\ |y(k)| \leq \langle\!\langle \mathscr{H} \rangle\!\rangle\, \overline{u} \tag{4.45}$$

où $\langle\!\langle \mathscr{H} \rangle\!\rangle$ est le Worst-Case Peak Gain *de $\mathscr{H}$, qu'on peut obtenir comme la somme infinie des valeurs absolues de sa réponse impulsionnelle h (définie dans la section 2.2.3) :*

$$\langle\!\langle \mathscr{H} \rangle\!\rangle = \sum_{k\in\mathbb{Z}} |h(k)| \quad \in \overline{\mathbb{R}} \tag{4.46}$$

Notons que les valeurs sont considérées dans $\overline{\mathbb{R}}$. Notamment, la somme infinie peut diverger : dans ce cas $\langle\!\langle \mathscr{H} \rangle\!\rangle$ est égal à $+\infty$ et l'équation (4.45) est trivialement vraie (le seul cas problématique serait $\overline{\boldsymbol{u}} = 0$, mais u est alors le signal nul donc y aussi vu que le filtre est LTI).

Interprétation. Le *Worst-Case Peak Gain* est en fait une norme sur les filtres, définie à partir de la norme infinie sur les signaux :

$$\|x\|_\infty \triangleq \sup_{k\in\mathbb{Z}} |x(k)| \qquad \text{pour tout signal scalaire } x \tag{4.47}$$

$$\langle\!\langle \mathscr{H} \rangle\!\rangle \triangleq \sup_{u \text{ tq } 0<\|u\|_\infty<+\infty} \frac{\|\mathscr{H}\{u\}\|_\infty}{\|u\|_\infty} \qquad \text{pour tout filtre SISO } \mathscr{H} \tag{4.48}$$

On peut voir le rapport $\frac{\|\mathscr{H}\{u\}\|_\infty}{\|u\|_\infty}$ comme le *gain* pour l'entrée u, qualifié de *peak* car on considère la norme infinie. On s'intéresse ensuite à la pire valeur de ce gain, c'est-à-dire sa borne supérieure sur toutes les entrées u, d'où le nom de *Worst-Case Peak Gain*. [1].

On remarque qu'on peut remplacer les équations (4.44) et (4.45) du théorème 4.3 par :

$$\|\mathscr{H}\{u\}\|_\infty \leq \langle\!\langle \mathscr{H} \rangle\!\rangle\, \|u\|_\infty \tag{4.49}$$

ce qui est évident par définition de $\langle\!\langle \mathscr{H} \rangle\!\rangle$ (mais il faut encore prouver l'équation (4.46)). La raison pour laquelle on a choisi un énoncé un peu plus long pour le théorème 4.3 est que pour l'appliquer aux filtres d'erreurs, c'est le passage de n'importe quelle borne sur l'entrée à une borne sur la sortie qui nous intéresse.

1. Le *Worst-Case Peak Gain* présenté ici est originellement appelé *peak gain* ou ℓ^∞*-gain* par Boyd *et al.* [23], qui réservent le nom *worst-case peak gain* à la pire valeur de ce *peak gain* dans un système à incertitude diagonale (lorsqu'un contrôleur est affecté par des perturbations linéaires) [8]. Le terme *Worst-Case Peak Gain* est plus tard repris avec un sens différent [58, 109, 110], pour désigner directement ce *peak gain* originel. C'est cette terminologie plus récente que j'utilise et que j'explique ci-dessus.

Par ailleurs, on reconnaît la norme 1 de h dans l'équation (4.46) :

$$\|h\|_1 \triangleq \sum_{k\in\mathbb{Z}} |h(k)| \tag{4.50}$$

Preuve. Le théorème 4.3 est facile à prouver en utilisant la caractérisation d'un filtre par sa réponse impulsionnelle donnée par l'équation (2.12) de la section 2.2.3. En effet, avec les notations du théorème :

$$\forall k,\ |\mathscr{H}\{u\}(k)| = |(h*u)(k)| = \left|\sum_{l=0}^{k} h(l)u(k-l)\right| \leq \left(\sum_{l\in\mathbb{Z}} |h(l)|\right) \bar{u} \tag{4.51}$$

Cela prouve le théorème 4.3 tel qu'il est formulé, si on définit la valeur $\langle\!\langle \mathscr{H} \rangle\!\rangle$ à partir de l'équation (4.46).

Mais si on définit $\langle\!\langle \mathscr{H} \rangle\!\rangle$ par l'équation (4.48), on a seulement prouvé que $\langle\!\langle \mathscr{H} \rangle\!\rangle \leq \|h\|_1$, puisqu'on a montré que $\|h\|_1$ est un majorant de l'ensemble des $\frac{\|\mathscr{H}\{u\}\|_\infty}{\|u\|_\infty}$. Pour pouvoir affirmer qu'on a vraiment $\langle\!\langle \mathscr{H} \rangle\!\rangle = \|h\|_1$, on doit aussi prouver que $\|h\|_1$ est optimal, c'est-à-dire que c'est le plus petit majorant. On montrera cela dans la section 4.3.2. Puis en section 4.3.3, on prouvera une propriété encore plus forte : $\|h\|_1$ est le maximum (vraiment atteint) de cet ensemble, c'est-à-dire qu'il existe un signal u tel que $0 < \|u\|_\infty < +\infty$ et $\|\mathscr{H}\{u\}\|_\infty = \|h\|_1 \|u\|_\infty$.

Lien avec la BIBO stabilité. On a vu en section 2.2.2 qu'un filtre est dit BIBO stable si pour toute entrée bornée, la sortie est aussi bornée [70]. Le théorème du *Worst-Case Peak Gain* assure qu'un filtre de *Worst-Case Peak Gain* fini est BIBO. Réciproquement, on montrera en section 4.3.3 que si un filtre est BIBO alors son *Worst-Case Peak Gain* est fini. Ainsi, comme annoncé dans la section 2.2.3, la BIBO stabilité d'un filtre LTI est caractérisée par la convergence absolue de la série correpondant à sa réponse impulsionnelle : $\sum_{k\in\mathbb{Z}} |h(k)| < +\infty$.

Cas MIMO. Dans le cas d'un filtre MIMO, on va borner les signaux vectoriels coefficient par coefficient. La borne $\bar{\boldsymbol{u}}$ sur l'entrée est donc un vecteur. Pour que $\langle\!\langle \mathscr{H} \rangle\!\rangle \bar{\boldsymbol{u}}$ soit également un vecteur, il faut que $\langle\!\langle \mathscr{H} \rangle\!\rangle$ soit une matrice. On rappelle que la réponse impulsionnelle $\boldsymbol{h}$ d'un filtre MIMO (définition 9) est justement un signal matriciel de taille $n_y \times n_u$.

Théorème 4.4 (Worst-Case Peak Gain, cas MIMO). *Soit $\mathscr{H}$ un filtre LTI MIMO à n_u entrées et n_y sorties. Soit $\boldsymbol{u}$ un signal d'entrée qui est borné (coefficient par coefficient) par un vecteur $\bar{\boldsymbol{u}} \in \overline{\mathbb{R}}^{n_u}$, c'est-à-dire :*

$$\forall k, \forall i,\ |\boldsymbol{u}_i(k)| \leq \bar{\boldsymbol{u}}_i \tag{4.52}$$

Alors la sortie correspondante $\boldsymbol{y}$ est bornée par le vecteur $\langle\!\langle \mathscr{H} \rangle\!\rangle \bar{\boldsymbol{u}}$:

$$\forall k, \forall i,\ |\boldsymbol{y}_i(k)| \leq (\langle\!\langle \mathscr{H} \rangle\!\rangle \bar{\boldsymbol{u}})_i \tag{4.53}$$

où $\langle\!\langle \mathscr{H} \rangle\!\rangle$ est le Worst-Case Peak Gain *de $\mathscr{H}$, obtenu ainsi à partir de sa réponse impulsionnelle $\boldsymbol{h}$ (la valeur absolue puis la somme infinie étant effectuées coefficient par coefficient) :*

$$\langle\!\langle \mathscr{H} \rangle\!\rangle = \sum_{k=0}^{+\infty} |\boldsymbol{h}(k)| \quad \in \overline{\mathbb{R}}^{n_y \times n_u} \tag{4.54}$$

Comme pour la réponse impulsionnelle en section 2.2.3, le *Worst-Case Peak Gain* d'un filtre MIMO est donc une matrice dont le coefficient en (i, j) représente le *Worst-Case Peak Gain* du filtre SISO $\mathscr{H}_{ij}$ caractérisant l'effet de l'entrée numéro j sur la sortie numéro i :

$$\langle\!\langle \mathscr{H} \rangle\!\rangle_{ij} = \langle\!\langle \mathscr{H}_{ij} \rangle\!\rangle \tag{4.55}$$

Formalisation. Comme expliqué au début de la section 4.3, on instancie l'anneau `RT` sur lequel on a défini les filtres à l'ensemble des réels (type `R` en Coq).

Comme on l'a fait pour la définition mathématique, on traite d'abord le cas SISO plus facile. On se donne donc un filtre LTI SISO $\mathscr{H}$ dont les signaux sont à valeurs dans $\mathbb{R}$.

```
Variable H : @SISO_filter R.
Hypothesis LTI : LTI_filter H.
```

Dans la formalisation, on définit $\langle\!\langle \mathscr{H} \rangle\!\rangle$ directement à partir de l'équation (4.46) et non de l'équation (4.48). Cela évite de rajouter une définition qui serait inutile vu qu'on veut juste prouver le théorème 4.3. On utilise `Lim_seq` de Coquelicot pour la limite d'une suite. Comme à la fois `Lim_seq` et l'opérateur de somme de MathComp prennent des entiers naturels, on se restreint aux termes de h d'indice positif, ce qui ne change bien sûr rien vu que $h(k)$ est nul pour $k < 0$.

```
Definition sum_abs_IR (n : nat) :=
  \sum_(0 ≤ i < n) Rabs (impulse_response H i).
Definition wcpg := Lim_seq sum_abs_IR.
```

On prouve alors le théorème 4.3.

```
Theorem wcpg_theorem (u : scsignal) (M : Rbar) :
  (forall k : int, Rbar_le (Rabs (u k)) M) →
  (forall k : int, Rbar_le (Rabs (H u k)) (Rbar_mult wcpg M)).
```

La preuve est assez rapide et suit l'équation (4.51).

On passe ensuite au cas d'un filtre MIMO. Le *Worst-Case Peak Gain* est alors défini comme la matrice des *Worst-Case Peak Gains* des sous-filtres SISO $\mathscr{H}_{ij}$, comme dans l'équation (4.55).

```
Variable H : @MIMO_filter R nin nout.
Hypothesis HLTI : LTI_filter H.

Definition MIMO_wcpg : 'M[Rbar]_(nout, nin) :=
  \matrix_(i,j) wcpg (to_SISO_yi_uj H i j).
```

La preuve du théorème 4.4 est très facile en décomposant la sortie numéro i comme la somme sur j des sorties des filtres $\mathscr{H}_{ij}$, et en appliquant à chacun de ces sous-filtres le théorème du cas SISO. La notation `_[i]ord` représente le coefficient d'indice `i` d'un vecteur colonne lorsque `i` est un ordinal.

```
Theorem MIMO_wcpg_theorem (u : @vsignal R nin)
                          (ubar : 'cV[Rbar]_nin) :
  (forall (k : int) (j : 'I_nin),
     Rbar_le (Rabs ((u k) _[j]ord))  (ubar _[j]ord)) →
  (forall (k : int) (i : 'I_nout),
     Rbar_le (Rabs ((H u k) _[i]ord))
             ((Rbar_mulmx MIMO_wcpg ubar) _[i]ord)).
```

4.3.2 Optimalité du *Worst-Case Peak Gain*

Encore une fois, on considère d'abord le cas SISO. On vient de prouver le théorème 4.3, c'est-à-dire que $\|h\|_1$ est un majorant de l'ensemble $\left\{\frac{\|\mathscr{H}\{u\}\|_\infty}{\|u\|_\infty} \mid 0 < \|u\|_\infty < +\infty\right\}$. Montrons maintenant que $\|h\|_1$ est optimal, c'est-à-dire qu'il s'agit du plus petit majorant de cet ensemble :

$$\forall M \in \mathbb{R}, M < \|h\|_1 \Rightarrow \exists u, 0 < \|u\|_\infty < +\infty \wedge \|\mathscr{H}\{u\}\|_\infty > M \|u\|_\infty \quad (4.56)$$

Cela signifiera que $\|h\|_1$ est vraiment le *Worst-Case Peak Gain* défini comme une norme dans l'équation (4.48).

Pour simplifier un peu l'équation (4.56), on va chercher des signaux témoins u qui sont bornés par 1 :

$$\forall M \in \mathbb{R}, M < \|h\|_1 \Rightarrow \exists u, \|u\|_\infty \leq 1 \wedge \|\mathscr{H}\{u\}\|_\infty > M \quad (4.57)$$

Vu que le filtre est LTI, ceci est équivalent à l'équation (4.56). En effet, si u est nul alors $\mathscr{H}\{u\}$ aussi, sinon on peut multiplier les deux par une même constante. Par ailleurs, il suffit d'exhiber un terme de $\mathscr{H}\{u\}$ dont la valeur absolue dépasse M. Ce qu'on veut prouver est donc :

$$\forall M \in \mathbb{R}, M < \|h\|_1 \Rightarrow \exists u, \exists N, \|u\|_\infty \leq 1 \wedge |\mathscr{H}\{u\}(N)| > M \quad (4.58)$$

Soit M tel que $M < \|h\|_1$. Comme $\|h\|_1$ est la limite de la suite des $(\sum_{i=0}^{n} |h(i)|)_n$, on sait qu'il existe un $N \in \mathbb{N}$ tel que :

$$\sum_{i=0}^{N} |h(i)| > M \quad (4.59)$$

On définit alors le signal u tel que :

$$u(k) = \operatorname{sgn}(h(N-k)) \quad \text{si } 0 \leq k \leq N \quad (4.60)$$

où sgn est la fonction de signe :

$$\forall r \in \mathbb{R}, \quad \operatorname{sgn}(r) = \begin{cases} 1 & \text{si } r > 0 \\ -1 & \text{si } r < 0 \\ 0 & \text{si } r = 0 \end{cases} \quad (4.61)$$

Les valeurs de $u(k)$ pour $k > N$ n'ont pas d'importance tant qu'elles sont bornées par 1 ; on peut par exemple les mettre à 0. On a alors :

$$\mathscr{H}\{u\}(N) = (h * u)(N) = \sum_{0 \leq i \leq N} h(i) \operatorname{sgn}(h(i)) = \sum_{0 \leq i \leq N} |h(i)| > M \quad (4.62)$$

Comme $|\mathscr{H}\{u\}(N)| \geq \mathscr{H}\{u\}(N)$, on a bien trouvé des témoins u et N pour l'équation (4.58).

En Coq, on fait exactement la même construction pour prouver l'équation (4.58) (on montre en fait une version légèrement plus forte que cette équation vu qu'on écrit `H u N > M` plutôt que `Rabs (H u N) > M`). On se place dans

la même `Section` de Coq (dont le mécanisme a été expliqué en section 1.2.4) que pour le cas SISO de la section 4.3.1 : `H` est un filtre SISO supposé LTI, et `wcpg` a été défini comme $\|h\|_1$ pour ce filtre `H`.

```
Theorem wcpg_optimal :
  forall M : R, Rbar_lt M wcpg →
  exists (u : scsignal) (N : int),
  (forall k : int, Rabs (u k) ≤ 1) ∧ H u N > M.
```

Dans le cas d'un filtre MIMO, on montre que chaque coefficient de $\sum_{k=0}^{+\infty} |\boldsymbol{h}(k)|$ est optimal. Ceci est facile à prouver à partir du cas SISO.

```
Theorem MIMO_wcpg_optimal (i : 'I_nout) (j : 'I_nin) :
forall M : R, Rbar_lt M (MIMO_wcpg i j) →
  exists (u : vsignal nin) N,
    (forall k j', Rabs (u k _[j']ord) ≤ 1) ∧ H u N _[i]ord > M.
```

4.3.3 Atteignabilité du *Worst-Case Peak Gain*

On montre ici une propriété encore plus forte que l'optimalité de la section 4.3.2 : la borne supérieure de l'équation (4.48) est atteignable, c'est-à-dire qu'il existe un signal u borné tel que $\|\mathscr{H}\{u\}\|_\infty = \langle\!\langle \mathscr{H} \rangle\!\rangle \|u\|_\infty$ (on s'intéresse seulement au cas SISO). Cela implique bien sûr l'optimalité vue précédemment. J'aurais donc pu présenter seulement la preuve de l'atteignabilité. Néanmoins, je pense que c'est utile de voir d'abord celle de l'optimalité qui est nettement plus facile. Cela permet aussi de remarquer que l'énoncé qu'on veut prouver pour l'atteignabilité est l'équation (4.57) de l'optimalité dans laquelle on a échangé les quantifications « $\forall M$ » et « $\exists u$ » :

$$\exists u, \quad \|u\|_\infty \leq 1 \quad \wedge \quad \forall M \in \mathbb{R}, \quad M < \|h\|_1 \;\Rightarrow\; \exists N,\ |\mathscr{H}\{u\}(N)| > M \tag{4.63}$$

En effet, cela implique que $\|\mathscr{H}\{u\}\|_\infty \geq \|h\|_1 \geq \|h\|_1 \|u\|_\infty$, or on a déjà prouvé en section 4.3.1 que $\|\mathscr{H}\{u\}\|_\infty \leq \|h\|_1 \|u\|_\infty$. Sans surprise, l'échange des quantificateurs rend l'énoncé beaucoup plus difficile à prouver, vu qu'on ne peut plus choisir u en fonction de M.

On doit donc directement construire un seul signal u tel que $|\mathscr{H}\{u\}|$ se rapproche arbitrairement près de $\|h\|_1$. Pour cela, on va considérer une suite $(M_n)_{n\in\mathbb{N}}$ qui converge vers $\|h\|_1$ par valeurs strictement inférieures (par exemple, $M_n = \|h\|_1 - (\frac{1}{2})^n$ si $\|h\|_1$ est fini, et $M_n = n$ si $\|h\|_1 = +\infty$). On va construire un signal u tel qu'il existe N_0 tel que $|\mathscr{H}\{u\}(N_0)| > M_0$, et il existe N_1 tel que $|\mathscr{H}\{u\}(N_1)| > M_1$, et ainsi de suite. L'intérêt de formuler les choses ainsi est qu'un filtre LTI est causal, c'est-à-dire que la sortie $\mathscr{H}\{u\}(N_0)$ ne dépend que des valeurs $u(k)$ pour $k \leq N_0$. Si on a un premier signal u tel que $|\mathscr{H}\{u\}(N_0)| > M_0$, cette propriété restera donc vraie même si on modifie les $u(k)$ pour $k > N_0$. On peut ensuite choisir des $u(k)$ pour $N_0 < k \leq N_1$ afin d'assurer que $|\mathscr{H}\{u\}(N_1)| > M_1$, puis des $u(k)$ pour $N_1 < k \leq N_2$ pour avoir $|\mathscr{H}\{u\}(N_2)| > M_2$, etc. L'étape principale de la preuve d'atteignabilité consiste donc à montrer que pour tout $M < \|h\|_1$, on peut prolonger des valeurs initiales données (bornées par 1) en un signal u tel que $\|u\|_\infty \leq 1$ et

$\exists N \in \mathbb{Z}, |\mathscr{H}\{u\}(N)| > M$. Écrivons cette étape principale sous forme d'équation, les valeurs initiales sont fournies par un signal u_{init} à consulter seulement pour les indices 0 à N_{init} inclus :

$$\begin{aligned}&\forall u_{\text{init}}, \forall N_{\text{init}}, \ \|u_{\text{init}}\|_\infty \leq 1 \ \Rightarrow \ \forall M, M < \|h\|_1 \ \Rightarrow \\ &\quad \exists u, \exists N, \ \|u\|_\infty \leq 1 \ \wedge \ |\mathscr{H}\{u\}(N)| > M \ \wedge \ \forall k \leq N_{\text{init}}, u(k) = u_{\text{init}}(k)\end{aligned} \tag{4.64}$$

Montrons cette dernière équation. Soit un signal u_{init} borné par 1, un entier N_{init} et un réel $M < \|h\|_1$. On distingue deux cas, selon que $\|h\|_1$ est fini ou vaut $+\infty$ (il ne peut pas valoir $-\infty$ vu qu'il est positif).

— **Cas $\|h\|_1 \in \mathbb{R}$.** Comme $\frac{M+\|h\|_1}{2} < \|h\|_1$, il existe $N' \in \mathbb{N}$ tel que $\sum_{i=0}^{N'} |h(i)| > \frac{M+\|h\|_1}{2}$. Posons $N = N_{\text{init}} + N' + 1$ et :

$$u(k) = \begin{cases} u_{\text{init}}(k) & \text{si } k \leq N_{\text{init}} \\ \text{sgn}(h(N-k)) & \text{si } N_{\text{init}} < k \end{cases} \tag{4.65}$$

On a alors :

$$\begin{aligned}\mathscr{H}\{u\}(N) &= \sum_{i=0}^{N} u(i)h(N-i) \\ &= \sum_{i=0}^{N_{\text{init}}} u_{\text{init}}(i)h(N-i) + \sum_{i=N_{\text{init}}+1}^{N} |h(N-i)| \\ &= \sum_{i=0}^{N_{\text{init}}} u_{\text{init}}(i)h(N-i) + \sum_{i=0}^{N'} |h(i)|\end{aligned} \tag{4.66}$$

Or $\|u_{\text{init}}\|_\infty \leq 1$ donc :

$$\left|\sum_{i=0}^{N_{\text{init}}} u_{\text{init}}(i)\ h(N-i)\right| \leq \sum_{i=0}^{N_{\text{init}}} |h(N-i)| = \sum_{i=N'+1}^{N} |h(i)| = \|h\|_1 - \sum_{i=0}^{N'} |h(i)| \tag{4.67}$$

en particulier :

$$\sum_{i=0}^{N_{\text{init}}} u_{\text{init}}(i)\ h(N-i) \geq -\left|\sum_{i=0}^{N_{\text{init}}} u_{\text{init}}(i)\ h(N-i)\right| \geq -\|h\|_1 + \sum_{i=0}^{N'} |h(i)| \tag{4.68}$$

On en déduit :

$$\begin{aligned}\mathscr{H}\{u\}(N) &\geq -\|h\|_1 + 2\sum_{i=0}^{N'} |h(i)| \\ &> -\|h\|_1 + 2\left(\frac{M+\|h\|_1}{2}\right) = M\end{aligned} \tag{4.69}$$

De plus, u est borné par 1 et identique à u_{init} pour $0 \leq k \leq N_{\text{init}}$ par construction. On a donc bien construit u et N vérifiant l'équation (4.64).

— **Cas $\|h\|_1 = +\infty$.** Il existe $N' \in \mathbb{N}$ tel que $\sum_{i=0}^{N'} |h(i)| > 2M$. On pose à nouveau $N = N_{\text{init}} + N' + 1$. On compare $\left|\sum_{i=0}^{N_{\text{init}}} u_{\text{init}}(i)h(N-i)\right|$ à M. Si $\left|\sum_{i=0}^{N_{\text{init}}} u_{\text{init}}(i)h(N-i)\right| > M$, on pose simplement :

$$u(k) = \begin{cases} u_{\text{init}}(k) & \text{si } k < N_{\text{init}} \\ 0 & \text{si } k \geq N_{\text{init}} \end{cases} \tag{4.70}$$

En effet, on a alors :

$$|\mathscr{H}\{u\}(N)| = \left|\sum_{i=0}^{N} u(i)h(N-i)\right| = \left|\sum_{i=0}^{N_{\text{init}}} u_{\text{init}}(i)h(N-i) + 0\right| > M \tag{4.71}$$

et les autres conditions de l'équation (4.64) sont trivialement remplies. Supposons maintenant que $\left|\sum_{i=0}^{N_{\text{init}}} u_{\text{init}}(i)h(N-i)\right| \leq M$. On pose :

$$u(k) = \begin{cases} u_{\text{init}}(k) & \text{si } k < N_{\text{init}} \\ \text{sgn}(h(N-i)) & \text{si } k \geq N_{\text{init}} \end{cases} \tag{4.72}$$

On obtient :

$$\begin{aligned} \mathscr{H}\{u\}(N) &= \sum_{i=0}^{N_{\text{init}}} u_{\text{init}}(i)h(N-i) + \sum_{i=N_{\text{init}}+1}^{N} |h(N-i)| \\ &\geq -\left|\sum_{i=0}^{N_{\text{init}}} u_{\text{init}}(i)h(N-i)\right| + \sum_{i=0}^{N'} |h(i)| \\ &> -M + 2M \end{aligned} \tag{4.73}$$

À nouveau, u est facilement borné par 1 et identique à u_{init} pour $0 \leq k \leq N_{\text{init}}$, donc on a bien construit u et N vérifiant l'équation (4.64).

On a ainsi montré l'équation (4.64) dans les deux cas. On s'en sert alors pour construire progressivement un signal u et des entiers N_0, N_1, etc. tels que $|\mathscr{H}\{u\}(k)|$ excède M_0 pour $k = N_0$, excède M_1 pour $k = N_1$, et ainsi de suite. On rappelle que $(M_n)_{n\in\mathbb{N}}$ est une suite qui tend vers M (par valeurs strictement inférieures pour pouvoir appliquer l'équation (4.64)). On obtient ainsi un signal u borné par 1 tel que $\|\mathscr{H}\{u\}\|_\infty \geq M$.

Comme M était n'importe quel réel strictement inférieur à $\|h\|_1 = \langle\!\langle \mathscr{H} \rangle\!\rangle$, on a bien construit un signal u borné tel que $\|\mathscr{H}\{u\}\|_\infty \geq \langle\!\langle \mathscr{H} \rangle\!\rangle \|u\|_\infty$ (et donc $\|\mathscr{H}\{u\}\|_\infty = \langle\!\langle \mathscr{H} \rangle\!\rangle \|u\|_\infty$ puisque l'autre sens de l'inégalité est donné par le théorème du *Worst-Case Peak Gain*). Ceci montre que le *Worst-Case Peak Gain* est atteignable.

Lien avec la BIBO stabilité. Cette propriété d'atteignabilité a une conséquence particulièrement intéressante dans le cas où $\|h\|_1 = +\infty$: elle signifie alors qu'il existe une entrée u bornée pour laquelle la sortie $\mathscr{H}\{u\}$ est non bornée, c'est-à-dire que le filtre n'est pas BIBO. En prenant la contraposition, on prouve comme annoncé en section 4.3.1 que si un filtre est BIBO alors son *Worst-Case Peak Gain* est fini. Cela signifie en particulier que pour les filtres étudiés

en pratique, auxquels on impose d'être BIBO, le théorème du *Worst-Case Peak Gain* fournira toujours une borne finie.

Formalisation. La preuve en Coq suit la preuve mathématique ci-dessus. Elle est très longue et repose sur beaucoup de constructions de signaux témoins. J'ai eu besoin d'utiliser l'existentielle fonctionnelle `{ x : T | P}` qui signifie qu'il existe `x` de type `T` tel que la propriété `P` (qui fait généralement intervenir la variable `x`) est vraie (autrement dit, l'ensemble des éléments de type `T` vérifiant `P` est non vide). Cette existentielle peut être utilisée pour construire une `Definition`, contrairement à celle donnée par le quantificateur `exists`. Je n'ai pas eu besoin d'ajouter d'axiome de logique classique. Cependant, je m'appuie beaucoup sur `LPO` (*Limited Principle of Omniscience*) prouvé par la bibliothèque Coquelicot à partir des axiomes définissant les réels de la bibliothèque standard, comme expliqué en section 2.1.2.

On se place encore dans le même contexte qu'en section 4.3.1 : le filtre SISO `H` est supposé LTI et `wcpg : Rbar` est son *Worst-Case Peak Gain*.

On définit un type enregistrement indiquant qu'un signal u borné par 1 (prédicat `boundedby1`) produit une sortie $\mathscr{H}\{u\}$ qui excède (en valeur absolue) une borne donnée $M \in \mathbb{R}$ pour un certain indice temporel $N \in \mathbb{N}$ (N serait censé être de type `int` en temps qu'indice de signal, mais en pratique on le construit dans `nat` et on compte sur la coercion `nat → int` pour pouvoir écrire `(H u N)`).

```
Record exceed_witness := {
  wit_u : scsignal ;
  wit_N : nat ;
  wit_M : R ;
  wit_bnd : boundedby1 wit_u ;
  wit_exceed : wit_M < Rabs (H wit_u wit_N)
}.
```

On prouve alors l'étape principale décrite précédemment par l'équation (4.64), ici formulée comme la construction d'un nouvel `exceed_witness` excédant la borne demandée M et prolongeant l'ancien `exceed_witness` fourni.

```
Lemma extend_exceed_witness (wit0 : exceed_witness) (M : R) :
  Rbar_lt M wcpg →
  { wit : exceed_witness |
      wit_M wit = M ∧ (wit_N wit0 < wit_N wit)%nat ∧
      (forall k, (k ≤ wit_N wit0)%int → wit_u wit k = wit_u wit0 k) }.
```

On définit une suite `Mseq : nat → R` qui tend vers `wcpg` par valeurs inférieures (si `wcpg` $= +\infty$ le n-ième terme est simplement n, sinon c'est `wcpg` $-(\frac{1}{2})^n$). Puis on définit une suite `exceed_witness_seq : nat → exceed_witness` récursivement. Le n-ième terme de cette suite (pour $n > 0$) est construit avec `extend_exceed_witness` pour étendre le précédent et excèder le n-ième terme de `Mseq` (l'initialisation est simplement `wit_u` égal au signal nul, `wit_N` à 0 et `wit_M` à -1). On veut ensuite construire le signal correspondant à la « limite » de la suite des signaux `wit_u` de `exceed_witness_seq`. Or on sait que pour chaque terme de cette suite, les valeurs du signal `wit_u` pour des indices entre 0 et `wit_N` sont définitives, au sens où elles seront désormais conservées lorsqu'on prolonge

chaque `exceed_witness`. On remarque aussi que `extend_exceed_witness` produit toujours un nouveau `wit_N` strictement plus grand que l'ancien : on montre facilement par récurrence que le `wit_N` de `exceed_witness_seq n` est supérieur à `n`. Cela signifie que la valeur en `n` de `wit_u (exceed_witness_seq n)` est une valeur définitive. On construit donc le signal `u_reachability` suivant. En utilisant la propriété `wit_exceed` des termes de `exceed_witness_seq`, on peut alors prouver que $|\mathscr{H}\{$`u_reachability`$\}|$ peut excéder n'importe quel terme de la suite `Mseq`, donc n'importe quel $M <$ `wcpg`.

```
Lemma u_reachability_causal :
  causal (fun k ⇒ wit_u (exceed_witness_seq k) k).
Definition u_reachability := Build_signal u_reachability_causal.

Lemma u_reachability_exceed (M : R) :
  Rbar_lt M wcpg → exists k, M < Rabs (H u k).
```

Cela signifie bien qu'il existe une entrée u bornée telle que $\|\mathscr{H}\{u\}\|_\infty \geq \langle\!\langle \mathscr{H} \rangle\!\rangle \|u\|_\infty$.

```
Theorem wcpg_reachable :
{ u : scsignal |
  (forall k, Rabs (u k) ≤ 1) ∧
  (forall M : R, Rbar_lt M wcpg →
    exists k, M < Rabs (H u k)) }.
```

Enfin, on prouve qu'un filtre BIBO (dont la définition Coq a été donnée en section 2.2.2) a un *Worst-Case Peak Gain* fini. Ceci a été présenté plus haut comme une contraposition du théorème qu'on vient de prouver, dans le cas où `wcpg` est infini. Mais en Coq, on n'a pas accès à la contraposition sans ajouter d'axiome de logique classique. On peut tout de même prouver cette propriété, entre autres parce que la finitude d'un élément de `Rbar` est décidable.

```
Theorem BIBO_finite_wcpg : BIBO_SISO_filter → is_finite wcpg.
```

4.3.4 Application à la vérification de formats de virgule fixe : absence d'*overflows*

Pour implémenter un filtre en virgule fixe en matériel, on veut choisir des formats appropriés pour chaque variable du programme. En particulier, on veut utiliser des *msbs* (section 1.3.2) suffisamment grands pour assurer qu'il n'y aura pas d'*overflow*. Mais on essaie de les prendre tout juste assez grands pour remplir cette condition : réduire le *msb* permet de réduire la largeur w donc accélérer les calculs, ou alors, si la largeur w est imposée, cela permet de réduire le *lsb* donc améliorer la précision.

De nombreuses techniques existent pour déterminer les formats, comme des simulations avec des nombres flottants, ou encore de l'arithmétique d'intervalle. On s'intéresse ici à la solution basée sur le théorème du *Worst-Case Peak Gain*.

On considère ici que les sommes de produits sont calculées par des algorithmes étudiés séparément (chapitre 5) donc on ne tient pas compte des calculs internes à ces sommes de produits. Les variables d'un filtre implémenté sous forme de SIF sont alors les variables d'état (composantes du signal d'état $\boldsymbol{x}$), les variables auxiliaires (composantes de $\boldsymbol{t}$) et les sorties (composantes de $\boldsymbol{y}(k)$). De plus, on suppose donné un vecteur $\overline{\boldsymbol{u}}$ de bornes sur les entrées.

Rappelons l'équation définissant le filtre modèle $\mathscr{H}$ associé à une SIF :

$$\mathscr{H} \quad \left\{ \begin{array}{rcl} \boldsymbol{J}\boldsymbol{t}(k+1) & = & \boldsymbol{M}\boldsymbol{x}(k) + \boldsymbol{N}\boldsymbol{u}(k) \\ \boldsymbol{x}(k+1) & = & \boldsymbol{K}\boldsymbol{t}(k+1) + \boldsymbol{P}\boldsymbol{x}(k) + \boldsymbol{Q}\boldsymbol{u}(k) \\ \boldsymbol{y}(k) & = & \boldsymbol{L}\boldsymbol{t}(k+1) + \boldsymbol{R}\boldsymbol{x}(k) + \boldsymbol{S}\boldsymbol{u}(k) \end{array} \right. \qquad \text{(rappel de (3.25))}$$

On peut facilement l'adapter en une nouvelle SIF qui a les mêmes signaux $\boldsymbol{t}$ et $\boldsymbol{x}$, mais dont la nouvelle sortie $\boldsymbol{y}_{vars}$ est une concaténation verticale de $\boldsymbol{t}$, $\boldsymbol{x}$ et $\boldsymbol{y}$:

$$\mathscr{H}_{vars} \quad \left\{ \begin{array}{rcl} \boldsymbol{J}\boldsymbol{t}(k+1) & = & \boldsymbol{M}\boldsymbol{x}(k) + \boldsymbol{N}\boldsymbol{u}(k) \\ \boldsymbol{x}(k+1) & = & \boldsymbol{K}\boldsymbol{t}(k+1) + \boldsymbol{P}\boldsymbol{x}(k) + \boldsymbol{Q}\boldsymbol{u}(k) \\ \boldsymbol{y}_{vars}(k) & = & \begin{pmatrix} \boldsymbol{I} \\ \boldsymbol{0} \\ \boldsymbol{L} \end{pmatrix} \boldsymbol{t}(k+1) + \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{I} \\ \boldsymbol{R} \end{pmatrix} \boldsymbol{x}(k) + \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{S} \end{pmatrix} \boldsymbol{u}(k) \end{array} \right. \tag{4.74}$$

On note $\mathscr{H}_{vars}$ le filtre modèle associé à cette SIF. En appliquant le théorème du *Worst-Case Peak Gain* (théorème 4.4 pour le cas MIMO), on obtient alors (en prenant encore une fois la valeur absolue et l'inégalité composante par composante) :

$$\forall k, |\boldsymbol{y}_{vars}(k)| \leq \langle\!\langle \mathscr{H}_{vars} \rangle\!\rangle \, \overline{\boldsymbol{u}} \tag{4.75}$$

Il reste à déterminer $\langle\!\langle \mathscr{H}_{vars} \rangle\!\rangle$. On a vu en section 3.2.5 comment passer de la SIF ci-dessus à un *State-Space* $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ décrivant le même filtre modèle $\mathscr{H}_{vars}$. On peut alors utiliser l'expression de la réponse impulsionnelle d'un *State-Space* vue en section 3.1.4 :

$$\boldsymbol{h}(k) = \begin{cases} \boldsymbol{D} & \text{si } k = 0 \\ \boldsymbol{C}\boldsymbol{A}^{k-1}\boldsymbol{B} & \text{si } k > 0 \\ \boldsymbol{0} & \text{si } k < 0 \end{cases} \qquad \text{(rappel de (3.7))}$$

On en déduit que le *Worst-Case Peak Gain* de $\mathscr{H}_{vars}$:

$$\langle\!\langle \mathscr{H}_{vars} \rangle\!\rangle = |\boldsymbol{D}| + \sum_{m=0}^{+\infty} |\boldsymbol{C}\boldsymbol{A}^m\boldsymbol{B}| \tag{4.76}$$

On peut ainsi calculer le vecteur $\langle\!\langle \mathscr{H}_{vars} \rangle\!\rangle \, \overline{\boldsymbol{u}}$ qui contient une borne pour chaque variable dont on a besoin pour implémenter la SIF initiale. De ces bornes, on déduit les *msbs* dont on a besoin et donc les formats à utiliser pour ne pas avoir d'*overflows*.

Il reste plusieurs problèmes avec cette approche. La somme infinie apparaissant dans $\langle\!\langle \mathscr{H}_{vars} \rangle\!\rangle$ n'est calculée qu'approximativement. Ce calcul lui-même, et la multiplication matricielle de $\langle\!\langle \mathscr{H}_{vars} \rangle\!\rangle$ par $\overline{\boldsymbol{u}}$, sont effectués en précision finie donc eux-mêmes sujets à des erreurs d'arrondis. De plus, les bornes obtenues sont sur les signaux modèles $\boldsymbol{t}$, $\boldsymbol{x}$ et $\boldsymbol{y}$, et non les signaux implémentés $\boldsymbol{t}^*$, $\boldsymbol{x}^*$ et $\boldsymbol{y}^*$, qu'on ne connaît pas encore puisqu'on n'a pas encore décidé quels formats utiliser pour leur implémentation. Or l'accumulation d'erreurs d'arrondis en calculant ces signaux peut modifier les bornes sur ces variables et potentiellement le *msb* minimal requis pour ne pas avoir d'*overflows*. Chacun de ces problèmes a un effet relativement petit : si on les ignore, on peut concevoir un filtre qui fonctionne correctement la plupart du temps, ce qui est satisfaisant pour beaucoup d'applications. Mais ce n'est pas du tout suffisant pour les applications critiques où un dysfonctionnement peut être dangereux.

Volkova [108] décrit une approche rigoureuse pour résoudre ces problèmes. En particulier, Volkova *et al.* [109] proposent un algorithme permettant de calculer le *Worst-Case Peak Gain* d'un filtre à un ε près, pour n'importe quel $\varepsilon > 0$ voulu, à partir des coefficients $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ de son *State-Space*. La preuve (sur papier) de cette borne prend en compte à la fois la somme infinie de l'équation (4.76) et la précision finie utilisée dans les calculs. Ensuite, les formats à utiliser sont déterminés en plusieurs étapes. On choisit d'abord des formats adaptés aux signaux modèles $\boldsymbol{t}$, $\boldsymbol{x}$ et $\boldsymbol{y}$. On s'intéresse alors au filtre implémenté $\mathscr{H}^*$ calculé avec ces formats. On étudie la propagation des erreurs d'arrondi dans ce filtre (par exemple en appliquant le théorème du *Worst-Case Peak Gain* aux filtres d'erreurs comme on verra en section 4.3.5) pour vérifier si les formats choisis sont encore adaptés aux signaux implémentés $\boldsymbol{t}^*$, $\boldsymbol{x}^*$ et $\boldsymbol{y}^*$. Si on doit modifier les formats, on itère jusqu'à trouver un point fixe. Cela nécessite généralement au plus une modification des formats, vu que les erreurs d'arrondi sont assez petites devant les binades des formats considérés.

Formaliser la SIF pour $\mathscr{H}_{vars}$ et montrer qu'on obtient bien une concaténation de $\boldsymbol{t}$, $\boldsymbol{x}$ et $\boldsymbol{y}$ en sortie serait facile. En revanche, prouver l'algorithme de calcul du *Worst-Case Peak Gain* est une perspective qui risque d'être difficile et sera discutée en section 6.3.3. Par ailleurs, notons que l'itération pour trouver les bons formats n'a pas besoin d'être formalisée dans son intégralité. Cette procédure peut être effectuée par un oracle non vérifié, tant qu'on prouve ensuite formellement que les formats obtenus à la fin sont corrects (c'est-à-dire qu'ils assurent qu'il n'y aura jamais d'*overflows*).

4.3.5 Application aux filtres d'erreurs

Appliquons maintenant le théorème du *Worst-Case Peak Gain* aux filtres d'erreurs obtenus en section 4.2. On a vu que l'erreur finale s'exprime comme :

$$\boldsymbol{\Delta y} = \mathscr{H}_{\Delta op}\{\boldsymbol{\varepsilon}\} + \mathscr{H}_{\Delta c}\{\boldsymbol{u}\} \qquad \text{(rappel de (4.43))}$$

En appliquant le théorème 4.4, on obtient :

$$\forall k, |\boldsymbol{\Delta y}(k)| \leq \langle\!\langle \mathscr{H}_{\Delta op} \rangle\!\rangle \, \bar{\boldsymbol{\varepsilon}} + \langle\!\langle \mathscr{H}_{\Delta c} \rangle\!\rangle \, \bar{\boldsymbol{u}} \tag{4.77}$$

où la valeur absolue, l'inégalité et les vecteurs de bornes $\bar{\boldsymbol{\varepsilon}}$ et $\bar{\boldsymbol{u}}$ sont tous considérés composante par composante.

On a donc besoin de connaître les *Worst-Case Peak Gains* des filtres d'erreurs. Ces derniers sont les filtres modèles définis par des SIF données en section 4.2.3, dont les coefficients s'expriment en fonction des matrices $\boldsymbol{J}$, $\boldsymbol{K}$, etc. de la SIF initiale et de leurs arrondis $\boldsymbol{J}^*$, $\boldsymbol{K}^*$, etc. Comme précédemment, on peut utiliser la traduction de la SIF vers le *State-Space* (section 3.2.5) pour déterminer un *State-Space* $(\boldsymbol{A}_{\Delta op}, \boldsymbol{B}_{\Delta op}, \boldsymbol{C}_{\Delta op}, \boldsymbol{D}_{\Delta op})$ correspondant à $\mathscr{H}_{\Delta op}$. Puis, grâce à la réponse impulsionnelle d'un *State-Space* (section 3.1.4), on obtient :

$$\langle\!\langle \mathscr{H}_{\Delta op} \rangle\!\rangle = |\boldsymbol{D}_{\Delta op}| + \sum_{m=0}^{+\infty} \left| \boldsymbol{C}_{\Delta op} \boldsymbol{A}_{\Delta op}^m \boldsymbol{B}_{\Delta op} \right| \tag{4.78}$$

De même pour le second filtre d'erreurs $\mathscr{H}_{\Delta c}$. Le cas général de l'équation (4.78) (et aussi de l'équation (4.76) de la section précédente), à savoir l'expression

du *Worst-Case Peak Gain* du filtre modèle associé à un *State-Space* donné, se prouve en Coq sans difficulté particulière.

```
Variable stsp : @StateSpace R nin nout nx.
Notation A := (StSp_A stsp). (* etc. : B, C, D *)
Lemma wcpg_from_StSp :
    MIMO_wcpg (filter_from_StSp stsp)
  = add_mxRbar (to_mxRbar (absmx D))
               (Lim_seq_mx (fun m ⇒
                 \sum_(0 ≤ k < m) absmx (C *m expmx A k *m B))).
```

Les bornes $\overline{\boldsymbol{u}}$ sur les entrées sont fournies avec le filtre à étudier. Il reste donc à déterminer des bornes $\bar{\boldsymbol{\varepsilon}}$ sur les erreurs des sommes de produits : cela dépend de l'algorithme de somme de produits utilisé, qu'on étudiera dans le chapitre 5.

Perspective : expression algébrique des *Worst-Case Peak Gain* des filtres d'erreurs. Une fois qu'on peut remplacer $\bar{\boldsymbol{\varepsilon}}$ par les bornes d'erreurs issues des algorithmes de sommes de produits du chapitre 5, on obtient une expression d'une borne sur $\boldsymbol{\Delta y}$. Cependant, cette expression est encore trop compliquée pour être utilisable en pratique : elle contient par exemple des sommes infinies de matrices comme dans l'équation (4.78).

Une solution consiste à s'intéresser à un exemple concret de filtre, avec des valeurs explicites de coefficients et des formats donnés de précision finie, afin de calculer la valeur numérique de la borne sur $\boldsymbol{\Delta y}$ obtenue. Cela demande cependant de pouvoir calculer précisément la somme infinie apparaissant dans le *Worst-Case Peak Gain*. Comme mentionné en section 4.3.4 et détaillé en section 6.3.3, un algorithme existe pour cela [109] mais risque d'être difficile à formaliser.

Une autre approche possible consiste à développer des expressions algébriques pour les *Worst-Case Peak Gain* des deux filtres d'erreurs qui nous intéressent. Ceci mettrait à profit un avantage de la preuve formelle sur d'autres méthodes formelles : pouvoir prouver des propriétés générales pour tous les filtres d'une certaine famille, plutôt que vérifier des propriétés d'un exemple numérique fourni. Cependant, les travaux sur papier relatifs à cette approche ne sont eux-mêmes pas du tout mûrs. En voici un bref aperçu informel.

Pour obtenir une borne utile sur $\boldsymbol{\Delta y}$, on veut prouver que chacun des termes $\langle\!\langle \mathscr{H}_{\Delta op} \rangle\!\rangle\, \bar{\boldsymbol{\varepsilon}}$ et $\langle\!\langle \mathscr{H}_{\Delta c} \rangle\!\rangle\, \overline{\boldsymbol{u}}$ de l'équation (4.77) est relativement petit. Pour le premier terme $\langle\!\langle \mathscr{H}_{\Delta op} \rangle\!\rangle\, \bar{\boldsymbol{\varepsilon}}$, ce sont les bornes $\bar{\boldsymbol{\varepsilon}}$ sur les erreurs des sommes de produits qui sont petites : on verra dans le chapitre 5 qu'elles sont égales à l'*ulp* du format du résultat, voire la moitié de cette *ulp* selon l'algorithme choisi. Dans le second terme $\langle\!\langle \mathscr{H}_{\Delta c} \rangle\!\rangle\, \overline{\boldsymbol{u}}$, on ne contrôle pas les bornes $\overline{\boldsymbol{u}}$ sur les entrées, qui ne sont pas forcément petites. On souhaiterait donc montrer que $\langle\!\langle \mathscr{H}_{\Delta c} \rangle\!\rangle$ est assez petit. On étudie d'abord le cas plus simple du *State-Space*. Le filtre $\mathscr{H}_{\Delta c}$ est alors défini par le *State-Space* suivant :

$$\boldsymbol{Z}_{\Delta c} \triangleq \left(\begin{array}{c:c|c} \boldsymbol{A} & \boldsymbol{0} & \boldsymbol{B} \\ \hdashline \boldsymbol{\Delta A} & \boldsymbol{A}^* & \boldsymbol{\Delta B} \\ \hline \boldsymbol{\Delta C} & \boldsymbol{C}^* & \boldsymbol{\Delta D} \end{array}\right) \qquad \text{(rappel de (4.23))}$$

où $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ est le *State-Space* du filtre modèle étudié $\mathscr{H}$ et $\boldsymbol{A}^* \triangleq \bigcirc(\boldsymbol{A})$ et $\boldsymbol{\Delta A} \triangleq \boldsymbol{A}^* - \boldsymbol{A}$. Par récurrence sur $m \in \mathbb{N}$, on prouve facilement que :

$$\left(\begin{array}{c|c} \boldsymbol{A} & \boldsymbol{0} \\ \hline \boldsymbol{\Delta A} & \boldsymbol{A}^* \end{array}\right)^m = \left(\begin{array}{c|c} \boldsymbol{A}^m & \boldsymbol{0} \\ \hline (\boldsymbol{A}^*)^m - \boldsymbol{A}^m & (\boldsymbol{A}^*)^m \end{array}\right) \tag{4.79}$$

On a alors :

$$\begin{aligned} \langle\!\langle \mathscr{H}_{\Delta c} \rangle\!\rangle &= |\boldsymbol{\Delta D}| + \sum_{m=0}^{+\infty} \left| \left(\boldsymbol{\Delta C} \mid \boldsymbol{C}^* \right) \left(\begin{array}{c|c} \boldsymbol{A} & \boldsymbol{0} \\ \hline \boldsymbol{\Delta A} & \boldsymbol{A}^* \end{array}\right)^m \left(\begin{array}{c} \boldsymbol{B} \\ \hline \boldsymbol{\Delta B} \end{array}\right) \right| \\ &= |\boldsymbol{D}^* - \boldsymbol{D}| + \sum_{m=0}^{+\infty} |\boldsymbol{C}^* (\boldsymbol{A}^*)^m \boldsymbol{B}^* - \boldsymbol{C} \boldsymbol{A}^m \boldsymbol{B}| \end{aligned} \tag{4.80}$$

On voit apparaître des termes interprétables comme des termes d'erreurs, mais les multiplications et exponentiations de matrices et la somme infinie compliquent l'analyse. Il faudrait réussir à simplifier davantage cette expression pour que cette approche devienne intéressante comparée au calcul des coefficients numériques de $\mathscr{H}_{\Delta c}$ puis de son *Worst-Case Peak Gain* grâce à l'algorithme de Volkova mentionné précédemment [109]. Une étude plus approfondie de cette approche constitue une perspective à long terme (section 6.3.3).

Chapitre 5

Algorithmes de somme de produits en virgule fixe et *overflows* modulaires

Dans le chapitre précédent, on a obtenu une borne sur l'erreur finale de la sortie d'un filtre, à condition de savoir borner l'erreur locale $\varepsilon(k)$ dans chaque somme de produits d'une SIF. Le chapitre présent s'intéresse donc aux algorithmes permettant de calculer une somme de produits et à l'erreur d'arrondi sur leur sortie. L'objectif est de calculer efficacement une valeur approchée de la somme :

$$\sum_{i=0}^{n-1} s_i t_i \tag{5.1}$$

pour deux suites finies (ou vecteurs) d'entrées $(s_i)_{0\leq i<n}$ et $(t_i)_{0\leq i<n}$.

Les produits scalaires et matriciels apparaissent dans de nombreux domaines des mathématiques et de l'informatique [75]. Les algorithmes de somme de produits ont donc fait l'objet de nombreuses études, majoritairement dans le cadre de la virgule flottante. De même pour leurs cousins, les algorithmes de somme de n termes. J'en propose ci-dessous un aperçu qui est loin d'être exhaustif car cette littérature est très étendue. Des vues d'ensembles plus détaillées sont données dans [53, 84, 90].

L'accumulateur de Kulisch [75] est le plus simple conceptuellement : il s'agit d'utiliser suffisamment de précision pour que tous les calculs intermédiaires soient exacts (par exemple avec des *bins* [48]). Cet algorithme est donc très précis, au prix d'un temps de calcul très élevé. De nombreux travaux ont pour objectif d'en concevoir une implémentation la plus rapide possible, par exemple à l'aide d'optimisations matérielles ou de calculs en parallèle [24, 71, 73, 91]. Daumas et Matula [36] proposent un algorithme calculant une somme de produits en *binary64* (*double*) à un bit près (arrondi fidèle de la définition 5.4), sauf en cas de rare *soustraction catastrophique* (*catastrophic cancellation*, quand on soustrait deux nombres presque égaux) qui est alors signalée par un *flag*. Cet algorithme effectue une *sticky accumulation* basée sur une implémentation matérielle d'un format similaire à *binary128* (*quad*) mais non normalisé. Il est moins

précis mais beaucoup plus rapide que toutes les variantes de l'accumulateur de Kulisch. Ogita, Rump et Oishi [95] développent des algorithmes de somme et de somme de produits renvoyant un résultat aussi précis que si les calculs utilisaient une expansion de K flottants pour un K demandé, mais utilisant seulement des additions, soustractions et multiplications dans le même format que les entrées. Les mêmes auteurs proposent ensuite un algorithme calculant un arrondi fidèle d'une somme pour un vecteur qui peut être de très grande dimension [102], ainsi qu'un algorithme renvoyant un arrondi au plus proche correct d'une somme mais dont le temps d'exécution augmente quand le résultat exact est proche d'un point milieu [103]. Lefèvre [76] présente un algorithme rapide correctement arrondi pour des flottants multiprécision en base 2, basé sur des tableaux d'entiers. C'est l'algorithme utilisé par la fonction `mpfr_sum` de la bibliothèque GNU MPFR [1] de C.

Mais les filtres numériques dans les systèmes embarqués utilisent souvent la virgule fixe comme expliqué en section 1.3.2, notamment lors d'une implémentation matérielle (sur FPGA ou ASIC). Dans ce chapitre, on considère désormais exclusivement l'arithmétique en virgule fixe en base 2. De plus, dans le contexte de ces filtres, la longueur n des séquences d'entrées est petite (par exemple de l'ordre de 10). Les s_i sont des constantes connues à l'avance puisqu'elles correspondent aux coefficients du filtre. Les formats de toutes les entrées et le format désiré pour la sortie sont eux aussi connus à l'avance (c'est-à-dire au moment de l'implémentation de l'algorithme plutôt que de son exécution).

L'accumulateur de Kulisch mentionné précédemment s'adapte tout à fait au cas de la virgule fixe, même si le temps de calcul important peut être rédhibitoire dans des systèmes embarqués. De Dinechin *et al.* [37] présentent une construction matérielle automatique d'un algorithme de somme de produits par des constantes qui renvoie un arrondi fidèle du résultat exact. Les calculs intermédiaires sont effectués avec une précision légèrement supérieure à celle désirée pour le résultat : on parle de *bits de garde*. Les multiplications correctement arrondies par les constantes sont obtenues rapidement grâce à des tables de consultation (*lookup tables*) précalculées. Cette approche est conçue dans l'optique des filtres LTI FIR (section 2.2.3, définition 8). Elle est ensuite généralisée à des entrées de formats différents [111] pour pouvoir être appliquée à des filtres LTI IIR. Par ailleurs, Lopez [85] étudie la recherche par optimisation combinatoire d'un algorithme de somme de produits ayant de bonnes propriétés (temps de calcul, précision du résultat) pour des formats donnés. Il cherche à optimiser les formats des produits intermédiaires p_i (approximations respectives des $s_i t_i$), et la façon dont les p_i sont sommés à l'aide d'additions binaires, qu'il représente sous forme d'un arbre binaire (par exemple pour $n = 4$ les possibilités incluent $(p_0 + p_1) + (p_2 + p_3)$, $((p_0 + p_3) + p_1) + p_2$, etc.).

Ce chapitre présente la formalisation en Coq d'algorithmes de somme de produits pour la virgule fixe en base 2, fournissant une borne prouvée sur l'erreur en sortie de chacun. Dans un premier temps, dans les sections 5.1 à 5.3, j'ignore les problèmes liés aux *overflows*. La section 5.1 explique la virgule fixe sans *overflow* que je considère, puis décrit sa formalisation en Coq (qui s'appuie fortement sur la bibliothèque Flocq [20]). La section 5.2 détaille et formalise

1. www.mpfr.org

deux algorithmes de somme de produits mentionnés ci-dessus : l'accumulateur de Kulisch et l'algorithme fidèle à bits de garde (ou plutôt, une simplification de ce second algorithme n'exploitant pas le fait que les s_i soient des constantes). La section 5.3 propose et formalise un nouvel algorithme de somme de produits correctement arrondi pour un arrondi au plus proche. Enfin, la section 5.4 s'intéresse à la prise en compte des *overflows*. Elle présente ma formalisation en Coq de l'*overflow* modulaire, puis de l'algorithme fidèle à bits de garde en autorisant des *overflows* modulaires dans les calculs intermédiaires.

5.1 Virgule fixe ignorant le problème des *overflows*

Cette section présente l'étude et la formalisation de la virgule fixe en ignorant le problème des *overflows*. La section 5.1.1 introduit les définitions et propriétés mathématiques des formats de virgule fixe sans *overflow*. La section 5.1.2 présente les éléments de la bibliothèque Flocq que j'utilise dans ma formalisation, principalement à travers la surcouche décrite en section 5.1.3, spécialisée pour la virgule fixe en base 2 sans *overflow*.

5.1.1 Arithmétique en virgule fixe sans borne sur la mantisse

On a vu en section 1.3.2 qu'un format de virgule fixe $\mathbb{F}_{\ell,w}$ est caractérisé par un exposant implicite ℓ (qui est aussi le *lsb*) et une largeur w (donnant le nombre de bits utilisés pour représenter la mantisse entière en complément à deux). L'ensemble des nombres représentables est alors :

$$\mathbb{F}_{\ell,w} \triangleq \{m \times 2^\ell \mid m \in \mathbb{Z} \wedge -2^{w-1} \leq m \leq 2^{w-1}-1\} \qquad \text{(rappel de (1.8))}$$

On a aussi vu qu'on a un *overflow* dès qu'on excède les valeurs extrêmes de $\mathbb{F}_{\ell,w}$, qui dépendent surtout du *msb* égal à $\ell + w - 1$.

Dans un premier temps, dans les sections 5.1 à 5.3, on ignore les difficultés liées aux *overflows*. Pour cela, on oublie la borne sur la mantisse imposée par sa représentation sur une largeur w fixée. On travaille donc avec des formats de virgule fixe théoriques autorisant des mantisses entières arbitrairement grandes. Un tel format est caractérisé uniquement par un *lsb* ℓ. J'utilise la notation $\mathbb{F}_l$ à la fois pour le format et l'ensemble des nombres représentables dans ce format :

$$\mathbb{F}_\ell = \{m \times 2^\ell \mid m \in \mathbb{Z}\} \tag{5.2}$$

Il s'agit donc de tous les multiples entiers du facteur d'échelle implicite 2^ℓ. Un tel format n'a ni largeur, ni *msb* (on peut aussi considérer que les deux valent $+\infty$). C'est un format théorique où la mantisse m peut être n'importe quel entier de $\mathbb{Z}$ et on ne se préoccupe pas de sa représentation sur des bits. Il n'y a jamais d'*overflow* dans ce format puisqu'il contient des valeurs représentables arbitrairement grandes ou petites.

La section courante présente les propriétés arithmétiques de l'approche décrite ci-dessus, dont le but est d'ignorer le problème des *overflows* en virgule fixe.

Ces propriétés seront formalisées en Coq dans les sections 5.1.2 et 5.1.3 (respectivement par la bibliothèque Flocq et une surcouche que je lui ai ajoutée) puis utilisées tout au long des sections 5.2 et 5.3.

Arrondis vers $\mathbb{F}_\ell$. Les modes d'arrondis présentés en section 1.3.1 pour la virgule flottante s'appliquent aussi au format $\mathbb{F}_\ell$. Comme on sera amené à considérer des arrondis vers différents formats dans un même algorithme, on note le *lsb* du format en indice du symbole d'arrondi. On utilisera surtout l'arrondi vers le bas $\bigtriangledown_\ell : \mathbb{R} \to \mathbb{F}_\ell$ et l'arrondi au plus proche $\bigcirc_\ell^\tau$ pour une règle de *tie-breaking* donnée τ (règle de bris d'égalité comme expliqué en section 1.3.1). En effet, le premier est très facile à implémenter en complément à deux (simple troncature comme mentionné en section 1.3.2 et détaillé en section 5.4.1), et le second est le mode d'arrondi le plus précis (c'est-à-dire celui qui minimise l'erreur dans le pire cas).

Ces arrondis vérifient bien les propriétés de base attendues d'une fonction arrondi : ce sont des fonctions croissantes qui préservent les nombres représentables ($\forall x \in \mathbb{F}_\ell, \bigtriangledown_\ell(x) = x$).

Remarque : *ulp* en virgule fixe. Une notion utile pour analyser les erreurs dans les arithmétiques en précision finie de manière générale est l'*ulp* d'un nombre (*Unit in the Last Place*). Il s'agit de l'écart entre deux nombres représentables consécutifs (ainsi nommé car cela correspond à la différence entre deux nombres identiques partout sauf sur le dernier bit de la mantisse). Dans un format de virgule flottante, la définition de l'*ulp* d'un nombre est un peu longue car elle varie avec l'exposant, et il faut traiter soigneusement le cas des puissances de 2 positives ou négatives... Mais en virgule fixe, c'est beaucoup plus simple : l'*ulp* dans le format $\mathbb{F}_\ell$ est toujours 2^ℓ, puisque les éléments de $\mathbb{F}_\ell$ sont régulièrement espacés de 2^ℓ. Ainsi, lorsqu'on étudie les erreurs d'arrondi dans $\mathbb{F}_\ell$, on verra souvent apparaître directement 2^ℓ ou $2^{\ell-1}$ là où on aurait rencontré $ulp(x)$ ou $\frac{1}{2}ulp(x)$ en virgule flottante (comme par exemple dans les lemmes suivants).

Lemme 5.1 (Caractérisation et erreur de l'arrondi vers le bas).
Pour tous $\ell \in \mathbb{Z}$ et $x, y \in \mathbb{R}$, on a :

$$\bigtriangledown_\ell(x) = y \quad \iff \quad y \in \mathbb{F}_\ell \ \wedge \ y \leq x < y + 2^\ell \tag{5.3}$$

En particulier :

$$\left|\bigtriangledown_\ell(x) - x\right| < 2^\ell \tag{5.4}$$

Preuve. Par définition de l'arrondi vers le bas, et comme le prochain élément de $\mathbb{F}_\ell$ après un certain $y \in \mathbb{F}_\ell$ est $y + 2^\ell$, on obtient :

$$\begin{aligned} \bigtriangledown_\ell(x) = y \quad &\iff \quad y = \max\{a \in \mathbb{F}_\ell \mid a \leq x\} \\ &\iff \quad y \in \mathbb{F}_\ell \ \wedge \ y \leq x \ \wedge \ y + 2^\ell > x \end{aligned} \tag{5.5}$$

□

Lemme 5.2 (Caractérisation et erreur de l'arrondi vers le haut).
Pour tous $\ell \in \mathbb{Z}$ et $x, y \in \mathbb{R}$, on a :

$$\bigtriangleup_\ell(x) = y \quad \iff \quad y \in \mathbb{F}_\ell \ \wedge \ y - 2^\ell < x \leq y \tag{5.6}$$

En particulier :

$$\left|\triangle_\ell(x) - x\right| < 2^\ell \tag{5.7}$$

Preuve. Similaire à la preuve du lemme 5.1. □

Lemme 5.3 (Erreur d'un arrondi au plus proche). *Pour tous $\ell \in \mathbb{Z}$ et $x \in \mathbb{R}$, et pour n'importe quelle règle de tie-breaking τ, on a :*

$$|\bigcirc_\ell^\tau(x) - x| \leq 2^{\ell-1} \tag{5.8}$$

Preuve. C'est évidemment vrai si $x \in \mathbb{F}_\ell$ c'est-à-dire $\bigcirc_\ell^\tau(x) = x$. Sinon, les deux nombres représentables les plus proches de x sont espacés de 2^ℓ et x se trouve entre les deux. Cela signifie que les distances de x à chacun des deux sont deux nombres positifs dont la somme est 2^ℓ. La plus petite des deux est donc inférieure ou égale à $2^{\ell-1}$. □

Pour décrire le comportement de l'algorithme qui sera présenté en section 5.2.4, on aura besoin de la notion suivante d'*arrondi fidèle*.

Définition 5.4 (Arrondi fidèle). *Le réel y est un* arrondi fidèle *du réel x dans le format $\mathbb{F}_\ell$ si :*

$$y = \triangledown_\ell(x) \quad \vee \quad y = \triangle_\ell(x) \tag{5.9}$$

Autrement dit, y est un arrondi fidèle de x s'il s'agit de n'importe lequel des deux nombres représentables encadrant x (ou si $y = x$ lorsque x est déjà dans $\mathbb{F}_\ell$). Ceci est illustré par la figure 5.1, sur laquelle les traits verticaux représentent les éléments de $\mathbb{F}_\ell$ (régulièrement espacés de l'*ulp* 2^ℓ).

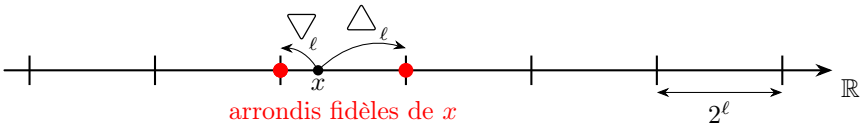


FIGURE 5.1 – Arrondi fidèle dans $\mathbb{F}_\ell$

Lemme 5.5 (Caractérisation d'un arrondi fidèle). *Le nombre $y \in \mathbb{F}_\ell$ est un* arrondi fidèle *du réel x dans le format $\mathbb{F}_\ell$ si et seulement si :*

$$|y - x| < 2^\ell \tag{5.10}$$

Preuve. C'est équivalent au lemme 5.1 lorsque $y \leq x$ ou au lemme 5.2 lorsque $y > x$. □

Notons que ce lemme est propre à la virgule fixe, au sens il n'a pas de contrepartie simple en virgule flottante [90] (à cause de la discontinuité de l'*ulp* au niveau des puissances de 2).

Lemme 5.6 (Inclusion des formats). *Si $\ell \leq \ell'$ alors $\mathbb{F}_{\ell'} \subset \mathbb{F}_\ell$.*

Preuve. Il suffit de se rappeler que $\mathbb{F}_\ell$ est par définition l'ensemble des nombres divisibles par 2^ℓ (c'est-à-dire s'écrivant sous la forme $k2^\ell$ avec $k \in \mathbb{Z}$). Comme 2^ℓ divise $2^{\ell'}$, tout multiple de $2^{\ell'}$ est aussi multiple de 2^ℓ d'où $\mathbb{F}_{\ell'} \subset \mathbb{F}_\ell$. □

Addition et multiplication. Les opérations basiques utilisées dans ce chapitre sont l'addition et la multiplication. On supposera qu'on dispose d'implémentations correctement arrondies pour l'arrondi vers le bas (ce qui est encore une fois facile en complément à deux). Comme on a vu en section 1.3.3, cela signifie que le résultat renvoyé est l'arrondi vers le bas du résultat exact en précision infinie. Je représente ces implémentations correctement arrondies par des notations binaires avec le symbole de l'opération à l'intérieur du symbole d'arrondi vers le bas, par exemple :

$$x \mathbin{\overline{\underline{+}}}_\ell y \triangleq \bigtriangledown_\ell(x+y) \qquad\qquad x \mathbin{\overline{\underline{\times}}}_\ell y \triangleq \bigtriangledown_\ell(x \times y) \tag{5.11}$$

Par ailleurs, on remarque que la somme (exacte, en précision infinie) de deux nombres de $\mathbb{F}_\ell$ est encore dans $\mathbb{F}_\ell$ (cela se voit bien en écrivant $m2^\ell + m'2^\ell = (m+m')2^\ell$). Une conséquence importante est que pour deux nombres x et y appartenant tous deux à $\mathbb{F}_\ell$, leur somme correctement arrondie (pour n'importe quel arrondi, en particulier l'arrondi vers le bas) est en fait exacte :

$$\forall x, y \in \mathbb{F}_\ell, \quad x \mathbin{\overline{\underline{+}}}_\ell y = \bigtriangledown_\ell(x+y) = x+y \qquad (\text{car } x+y \in \mathbb{F}_\ell) \tag{5.12}$$

Similairement, si $x \in \mathbb{F}_\ell$ et $y \in \mathbb{F}_{\ell'}$ alors $xy \in \mathbb{F}_{\ell+\ell'}$ (puisque $m2^\ell m'2^{\ell'} = mm'2^{\ell+\ell'}$) donc :

$$\forall x \in \mathbb{F}_\ell, \forall y \in \mathbb{F}_{\ell'}, \quad x \mathbin{\overline{\underline{\times}}}_{\ell+\ell'} y = xy \tag{5.13}$$

Lemmes pour la section 5.3. Voici deux lemmes qui serviront à prouver le lemme 5.15 en section 5.3.1 et le théorème 5.20 en section 5.3.2.

Lemme 5.7. *Soit $x, y \in \mathbb{R}$ et $\ell \in \mathbb{Z}$. Si $y \in \mathbb{F}_\ell$ alors :*

$$\bigtriangledown_\ell(x) + y = \bigtriangledown_\ell(x+y) \tag{5.14}$$

Preuve. Cette preuve repose sur la caractérisation l'arrondi vers le bas du lemme 5.1. On sait que $\bigtriangledown_\ell(x) \leq x < \bigtriangledown_\ell(x) + 2^\ell$ donc :

$$\bigtriangledown_\ell(x) + y \leq x + y < \bigtriangledown_\ell(x) + y + 2^\ell \tag{5.15}$$

De plus $\bigtriangledown_\ell(x) \in \mathbb{F}_\ell$ et $y \in \mathbb{F}_\ell$ donc $\bigtriangledown_\ell(x)+y \in \mathbb{F}_\ell$. On en déduit bien $\bigtriangledown_\ell(x+y) = \bigtriangledown_\ell(x) + y$. □

Lemme 5.8. *Soit $x \in \mathbb{R}$ et $\ell, \ell' \in \mathbb{Z}$. Si $\ell \leq \ell'$ alors :*

$$\bigtriangledown_{\ell'}(\bigtriangledown_\ell(x)) = \bigtriangledown_{\ell'}(x) \tag{5.16}$$

Preuve. Toujours d'après le lemme 5.1, on a :

$$\bigtriangledown_\ell(x) < \bigtriangledown_{\ell'}(\bigtriangledown_\ell(x)) + 2^{\ell'} \tag{5.17}$$

Or $\bigtriangledown_\ell(x) \in \mathbb{F}_\ell$ et $\bigtriangledown_{\ell'}(\bigtriangledown_\ell(x)) \in \mathbb{F}_\ell$ et $2^{\ell'} \in \mathbb{F}_\ell$ (d'après le lemme 5.6)
donc le nombre $\bigtriangledown_{\ell'}(\bigtriangledown_\ell(x)) + 2^{\ell'} - \bigtriangledown_\ell(x)$ appartient à $\mathbb{F}_\ell$ et est strictement positif, donc ce nombre est supérieur ou égal à 2^ℓ d'où :

$$\bigtriangledown_\ell(x) \leq \bigtriangledown_{\ell'}(\bigtriangledown_\ell(x)) + 2^{\ell'} - 2^\ell \tag{5.18}$$

On a alors :

$$\bigtriangledown_{\ell'}(\bigtriangledown_\ell(x)) \leq \bigtriangledown_\ell(x) \leq x < \bigtriangledown_\ell(x) + 2^\ell \leq \bigtriangledown_{\ell'}(\bigtriangledown_\ell(x)) + 2^{\ell'} - 2^\ell + 2^\ell \tag{5.19}$$

avec $\bigtriangledown_{\ell'}(\bigtriangledown_\ell(x)) \in \mathbb{F}_{\ell'}$ d'où $\bigtriangledown_{\ell'}(x) = \bigtriangledown_{\ell'}(\bigtriangledown_\ell(x))$.

□

Ces deux lemmes seront prouvés formellement en section 5.3.1.

5.1.2 Bibliothèque Flocq

Flocq [20] groupe en une seule formalisation très générale de nombreux formats de précision finie, notamment les formats usuels de virgule flottante et virgule fixe. Cette section présente les éléments de Flocq sur lesquels je m'appuie pour traiter la virgule fixe dans ma formalisation.

Cette bibliothèque utilise les réels `R` [87] et les entiers relatifs `Z` de la bibliothèque standard.

Prédicat pour décrire un format. Flocq représente un format sous la forme d'un prédicat `R → Prop` caractérisant l'ensemble des nombres représentables dans le format. Ainsi, les nombres du format sont manipulés comme des réels de type `R`, accompagnés si nécessaire de l'hypothèse qu'ils appartiennent au format. L'intérêt est qu'on peut alors directement appliquer tous les outils de Coq sur `R` à ces nombres. C'est cette approche qui a inspiré l'arithmétique générique que j'ai présentée en section 1.3.3.

Un élément essentiel de Flocq est `generic_format`, qui permet de construire les prédicats représentant de nombreux formats, notamment les formats usuels de virgule flottante ou fixe.

```
Definition generic_format (beta : radix) (fexp : Z → Z) : R → Prop.
```

Le format est caractérisé par une base β et une *fonction d'exposant* $\varphi : \mathbb{Z} \to \mathbb{Z}$ détaillée ci-dessous. Le type `radix`, utilisé pour la base, correspond aux entiers de `Z` qui sont supérieurs ou égaux à 2. Dans ce chapitre, je m'intéresse seulement à la base 2, qui est représentée par l'élément `radix2` de type `radix`.

Fonction d'exposant. La fonction d'exposant $\varphi : \mathbb{Z} \to \mathbb{Z}$ indique l'exposant à considérer pour chaque réel $r \in \mathbb{R}$. Son argument entier est la *magnitude* de r définie comme $\lfloor \log_\beta |r| \rfloor + 1$, c'est-à-dire l'unique [2] entier e tel que $\beta^{e-1} \leq |r| < \beta^e$. L'*exposant canonique* pour r est alors $\varphi(e)$. Le réel r est dans le format décrit par `generic_format` s'il est un multiple entier de $\beta^{\varphi(e)}$, c'est-à-dire s'il existe $k \in \mathbb{Z}$ tel que $r = k\beta^{\varphi(e)}$, autrement dit si :

$$r = \lfloor r\beta^{-\varphi(e)} \rfloor \, \beta^{\varphi(e)} \tag{5.20}$$

2. Dans le cas particulier $r = 0$, il n'existe pas d'entier e satisfaisant cette condition, et d'ailleurs $\log_\beta 0$ n'est pas défini dans $\mathbb{R}$. Ce n'est pas grave : on utilise alors n'importe quel entier e, et on obtient bien que 0 est dans le format puisque $0 = 0\beta^e$.

On remarque que pour être représentable à l'aide de `generic_format`, un format doit donc utiliser le même exposant sur toute une binade (ou β-ade en base quelconque). C'est bien le cas des formats classiques de virgule flottante et virgule fixe.

N'importe quelle fonction $\varphi : \mathbb{Z} \to \mathbb{Z}$ ne se prête pas à la définition d'un format de précision finie pertinent. Flocq définit donc un prédicat `Valid_exp : (Z → Z) → Prop` qui indique si une telle fonction est une fonction d'exposant adéquate. De nombreux lemmes et théorèmes de Flocq ont comme hypothèse que la fonction d'exposant considérée vérifie bien `Valid_exp`. Des exemples de tels théorèmes sont donnés plus bas.

Flocq fournit des fonctions φ permettant d'obtenir les prédicats de plusieurs formats usuels (en montrant bien sûr qu'elles vérifient `Valid_exp`). Par exemple, `FLX_exp prec` est la fonction d'exposant pour le format de virgule flottante de précision `prec : Z`, mais où l'exposant peut être n'importe quel entier (cela ignore donc les problèmes d'*overflow* et d'*underflow*). La fonction `FLT emin prec` représente le format de virgule flottante de précision `prec : Z` où les exposants sont au minimum `emin : Z`, mais n'ont pas de maximum (les *underflows* sont donc pris en compte mais pas les *overflows*).

Le format qui nous intéresse ici est la virgule fixe. L'exposant est le même pour tous les réels : il s'agit du *lsb* (*Least Significant Bit*) présenté en section 1.3.2). La fonction d'exposant définie par Flocq pour un *lsb* donné est donc la fonction constante égale à ce *lsb*.

```
Definition FIX_exp (lsb : Z) : Z → Z := fun (e : Z) ⇒ lsb.
Lemma FIX_exp_valid lsb : Valid_exp (FIX_exp lsb).
```

Le prédicat décrivant le format de virgule fixe en base 2 pour un certain *lsb* noté `lsb` est donc `generic_format radix2 (FIX_exp lsb) : R → Prop`.

Fonction d'arrondi. Flocq définit une fonction d'arrondi `round` qui envoie un nombre réel vers un nombre du format de précision finie considéré. Cette définition est à nouveau très générale : elle couvre de nombreux modes d'arrondis, dont tous les plus communs (vers le haut, vers le bas, vers zéro, au plus proche, etc.). Le nombre renvoyé est de type `R`, mais le théorème `generic_format_round` atteste qu'il est bien dans le format.

```
Definition round (beta : radix) (fexp : Z → Z) (rnd : R → Z)
                                                  : R → R.

Theorem generic_format_round (beta : radix) (fexp : Z → Z) :
  Valid_exp fexp →
  forall rnd : R → Z, Valid_rnd rnd →
  forall r : R, generic_format beta fexp (round beta fexp rnd r).
```

Le mode d'arrondi est décrit par l'argument `rnd`, qui indique quel arrondi vers $\mathbb{Z}$ on utilise pour obtenir la mantisse. Plus précisément, soit un réel r de magnitude e, dont l'exposant canonique est donc $\varphi(e)$. On veut arrondir r vers un nombre de la forme $m\beta^{\varphi(e)}$ où $m \in \mathbb{Z}$. Il suffit pour cela d'arrondir $r\beta^{-\varphi(e)}$ vers un entier. C'est là qu'on utilise la fonction `rnd` de type `R → Z`. L'arrondi de r est alors $\texttt{rnd}(r\beta^{-\varphi(e)})\beta^{\varphi(e)}$. Par exemple, pour décrire l'arrondi vers le bas, on instanciera `rnd` à la partie entière inférieure `Zfloor` ; pour l'arrondi vers le haut, on utilisera la partie entière supérieure `Zceil`.

Encore une fois, n'importe quelle fonction `rnd` ne produira pas un arrondi pertinent. Le prédicat `Valid_rnd : (R → Z) → Prop` caractérise les fonctions appropriées : elles doivent être croissantes et préserver les entiers (pour tout entier `z`, `rnd (IZR z) = z` où `IZR` est le plongement de `Z` dans `R`). Cela assure qu'on obtient une fonction d'arrondi $\mathbb{R} \to \mathbb{R}$ valide, comme défini en section 1.3. En particulier, on obtient le théorème suivant qui énonce que les nombres du format sont des points fixes de la fonction d'arrondi.

```
Theorem round_generic (beta : radix) (fexp : Z → Z)
                      (rnd : R → Z) :
  Valid_rnd rnd →
  forall x : R, generic_format beta fexp x →
  round beta fexp rnd x = x.
```

Arrondi au plus proche. On a vu qu'un arrondi au plus proche dépend d'une règle de *tie-breaking* pour les cas où le nombre à arrondir est un point milieu. La fonction `Znearest` à utiliser pour ce mode d'arrondi dépend donc d'un argument `choice` qui indique pour chaque point milieu s'il faut l'arrondir vers le haut ou vers le bas (plus précisément, il s'agit d'une fonction `Z → bool` à laquelle on donne en argument la mantisse du nombre fixe juste en Dessous du point milieu considéré, et qui renvoie `true` s'il faut arrondir ce point milieu vers le haut ou `false` si c'est vers le bas).

```
Definition Znearest (choice : Z → bool) (x : R) :=
  match Rcompare (x - IZR (Zfloor x)) (/ 2) with
  | Eq ⇒ if choice (Zfloor x) then Zceil x else Zfloor x
  | Lt ⇒ Zfloor x
  | Gt ⇒ Zceil x
  end.
```

Flocq définit les règles de *tie-breaking*, c'est-à-dire les fonctions `choice` à utiliser, pour l'arrondi au plus proche pair et l'arrondi au plus proche *away-from-zero* qui sont requis par la norme flottante IEEE-754 [65].

Borne sur l'erreur d'arrondi. Parmi les théorèmes particulièrement intéressants de Flocq figurent ceux qui bornent les erreurs d'arrondi. Voici ceux pour n'importe quel arrondi valide et pour un arrondi au plus proche avec n'importe quelle règle de *tie-breaking*. La borne est exprimée en fonction de l'*ulp*, qui est aussi défini par Flocq.

```
Theorem error_lt_ulp :
  forall (beta : radix) (fexp : Z → Z), Valid_exp fexp →
  forall rnd : R → Z, Valid_rnd rnd →
  forall x : R, x ≠ 0 →
  Rabs (round beta fexp rnd x - x) < ulp beta fexp x.

Theorem error_le_half_ulp :
  forall (beta : radix) (fexp : Z → Z), Valid_exp fexp →
  forall (choice : Z → bool) (x : R),
  (Rabs (round beta fexp (Znearest choice) x - x)
     ≤ / 2 * ulp beta fexp x)%R.
```

Comme on a vu en section 5.1.1, en virgule fixe en base 2, l'*ulp* de n'importe quel réel est toujours 2^{lsb}. Ce résultat est prouvé par Flocq pour n'importe quelle base (la fonction `bpow` appliquée à une base β et un entier e renvoie β^e).

```
Lemma ulp_FIX (beta : radix) (emin : Z) (x : R) :
  ulp beta (FIX_exp emin) x = bpow beta emin.
```

Grâce à ce lemme, on peut récrire les deux théorèmes précédents plus simplement dans le cas de la virgule fixe. Je le fais dans la surcouche présentée en section 5.1.3.

5.1.3 Surcouche pour la virgule fixe en base 2

Comme on vient de le voir, les définitions de Flocq ont beaucoup de paramètres pour être très générales. J'ai défini une formalisation simplifiée pour juste la virgule fixe en base 2. Comme expliqué en section 5.1.1, on ignore le problème des *overflows* en n'imposant aucune borne sur les mantisses.

Certaines définitions sont directement construites à partir de Flocq. Par exemple, `roundFIX` que je définis plus loin est juste `round` avec des paramètres particuliers. Beaucoup de preuves sont immédiates à partir du bon lemme de Flocq. Cette formalisation simplifiée n'est donc qu'une fine couche par-dessus Flocq. Son intérêt est d'alléger les énoncés avec moins de paramètres explicites. Elle permet aussi d'identifier les éléments à redéfinir lorsqu'on formalisera les *overflows* en section 5.4.2.

J'utilise le type `Z` pour les *lsbs* comme Flocq plutôt que `int` comme dans ma formalisation des filtres (ceci sera discuté en section 6.2.1).

Pour toute cette section, on se donne un paramètre `lsb` représentant le *lsb* du format considéré.

```
Variable lsb : Z.
```

Prédicat pour décrire un format. Commençons par définir le format $\mathbb{F}_{\texttt{lsb}}$. Je garde l'approche de Flocq où un format est défini par un prédicat, et les nombres du format sont simplement des réels accompagnés de l'hypothèse qu'ils appartiennent au format. Cependant, pour faciliter les preuves avec SSReflect [47], je ne considère pas un prédicat `R → Prop` comme dans Flocq, mais plutôt un prédicat `R → bool`. On a vu en section 5.1.1 que $\mathbb{F}_{\texttt{lsb}}$ est l'ensemble des nombres qui sont des multiples entiers de $2^{\texttt{lsb}}$. Je le caractérise donc par le prédicat `isFIX` suivant.

```
Definition isFIX (r : R) : bool := (twopow lsb |d r).
```

La notation `twopow` désigne la fonction $e \mapsto 2^e$, obtenue à partir de la fonction `bpow` de Flocq partiellement appliquée à la base 2. La notation `|d` représente la divisibilité entière dans $\mathbb{R}$. En effet, on caractérise les réels qui sont des entiers grâce à la fonction `up : R → Z` de la bibliothèque standard, qui renvoie un réel vers le plus petit entier qui dépasse strictement ce réel [3]. Ensuite, on définit qu'un réel divise un autre si leur quotient est un entier.

3. Il aurait été légèrement plus simple d'utiliser la partie entière inférieure `Zfloor` de Flocq. J'ai utilisé `up` pour ne pas avoir besoin d'importer Flocq dans le fichier `R_complements`.

```
Definition is_integer (r : R) : bool := IZR (up r) == r + 1.
Definition Rdivides (r1 r2 : R) : bool := is_integer (r2 * / r1).
Infix " |d " := Rdivides (at level 70).
```

Ces définitions ne sont pas essentielles pour la formalisation de la virgule fixe. On aurait pu définir `isFIX` autrement. Cependant, elles seront beaucoup utilisées pour les *overflows* en section 5.4. On peut donc aussi bien s'en servir pour écrire une définition simple de `isFIX` à valeurs dans `bool` plutôt que `Prop`.

On prouve ensuite que `isFIX` coïncide avec la définition de la virgule fixe obtenue à partir de `generic_format` de Flocq (sous la forme d'un lemme de réflection typique de SSReflect). Cela permettra d'utiliser de nombreux lemmes de Flocq.

```
Lemma isFIX_P (r : R) :
  reflect (generic_format radix2 (FIX_exp lsb) r) (isFIX r).
```

Arrondis et opérations. Pour un mode d'arrondi `rnd` comme dans Flocq, on définit l'arrondi vers $\mathbb{F}_{\texttt{lsb}}$ correspondant à partir de la fonction `round` vue en section 5.1.2. On définit aussi l'addition et la multiplication correctement arrondies pour cet arrondi. En effet, ce sont les deux opérations dont on aura besoin pour les algorithmes de somme de produits.

```
Definition roundFIX (rnd : R → Z) :=
  (round radix2 (FIX_exp lsb) rnd).
Definition binopFIX (rnd : R → Z) op (r1 r2 : R) :=
  roundFIX rnd (op r1 r2).
Definition addFIX (rnd : R → Z) := binopFIX rnd Rplus.
Definition mulFIX (rnd : R → Z) := binopFIX rnd Rmult.
```

On prouve quelques propriétés sur cet arrondi et ces opérations. Notamment, les lemmes d'erreur suivants sont obtenus à partir des théorèmes de Flocq donnés à la fin de la section 5.1.2, qui bornent l'erreur en fonction de l'*ulp* (toujours $2^{\texttt{lsb}}$ en virgule fixe).

```
Lemma roundFIX_error (rnd : R → Z) :
  Valid_rnd rnd →
  forall r : R, Rabs (roundFIX rnd r - r) < twopow lsb.
Lemma roundFIX_rN_error (choice : Z → bool) (r : R) :
  Rabs ( roundFIX (Znearest choice) r - r ) ≤ twopow (lsb - 1).
```

On définit aussi un prédicat `faithful` caractérisant un arrondi fidèle (définition 5.4). On prouve le lemme 5.5 : on a bien un arrondi fidèle quand l'erreur est strictement inférieure à un *ulp* (comme mentionné en section 5.1.1, ce lemme ne serait pas valable pour la virgule flottante). Ceci servira à prouver l'algorithme de la section 5.2.4.

```
Definition faithful (r x : R) : Prop :=
  x = roundFIX Zfloor r \/ x = roundFIX Zceil r.
Lemma diff_lt_faithful (r x : R) :
  isFIX x → Rabs (x - r) < twopow lsb → faithful r x.
```

5.2 Algorithmes classiques de somme de produits

Cette section présente des algorithmes préexistants de somme de produits. Elle ignore le problème des *overflows* en utilisant les formats de virgule fixe décrits en section 5.1.1. Les algorithmes présentés sont formalisés en utilisant la surcouche de Flocq pour la virgule fixe présentée en section 5.1.3.

La section 5.2.1 donne le contexte et les notations communs à tous les algorithmes des sections 5.2 et 5.3, avec les éléments de formalisation correspondants. L'accumulateur de Kulisch en section 5.2.2 utilise autant de bits qu'il faut pour rendre les opérations exactes, garantissant une grande précision du résultat au prix d'un temps de calcul élevé. La section 5.2.3 décrit une variante de cet accumulateur, qui s'appuie sur un tri des entrées selon leur *lsb* pour réduire le temps de calcul. L'algorithme présenté en section 5.2.4 utilise un nombre soigneusement choisi de bits de garde pour calculer très efficacement un arrondi fidèle (définition 5.4) du résultat exact.

5.2.1 Généralités et notations

Cette section présente les notations, principes et éléments de formalisation communs aux algorithmes des sections 5.2 et 5.3.

Entrées et sortie. Les entrées sont deux séquences de nombres $(s_i)_{0\leq i<n}$ et $(t_i)_{0\leq i<n}$ d'une même longueur notée n. On veut calculer la somme de leurs produits respectifs. On appelle *résultat modèle* et on note $mres$ le résultat exact obtenu en précision infinie :

$$mres \triangleq \sum_{i=0}^{n-1} s_i t_i \tag{5.21}$$

Les entrées sont des nombres en virgule fixe dans des formats qui peuvent varier d'une entrée à l'autre. On note ℓ_{s_i} et ℓ_{t_i} les *lsbs* respectifs de ces formats. On se donne aussi un *lsb final* ℓ_f : on souhaite que la sortie renvoyée par l'algorithme, notée `res`, soit dans le format $\mathbb{F}_{\ell_f}$.

Dans le contexte des sommes de produits utilisées dans les filtres numériques, on pourrait ajouter comme hypothèse que les s_i sont des constantes (coefficients a_i et b_i de la fonction de transfert dans le cas des formes directes, éléments des matrices d'une SIF...). Mais les algorithmes que je présente dans ce chapitre ne mettent pas à profit cette hypothèse donc je ne la considère pas.

Principe général et enjeu. Les algorithmes présentés dans ce chapitre ont tous pour principe général de multiplier chaque s_i par t_i, puis de sommer les produits obtenus à l'aide d'additions binaires. Les différences sont les formats et arrondis utilisés pour les calculs intermédiaires (n multiplications et $n-1$ additions binaires), et l'ordre dans lequel les produits sont ajoutés entre eux. En effet, cet ordre peut affecter le résultat puisque l'addition en virgule fixe n'est pas associative dans le cas général. Notons cependant qu'elle redevient associative lorsque l'on somme au même *lsb* que les termes afin d'assurer que toutes les additions sont exactes. Ceci permettra parfois de modifier l'ordre des additions pour les paralléliser facilement.

Les algorithmes de somme de produits sont évalués en fonction de deux critères principaux : leur précision (borne étroite sur l'erreur en sortie $|\texttt{res} - mres|$)

et leur efficacité (temps de calcul et énergie consommée, liés notamment au nombre de bits impliqués dans les calculs). Les deux varient avec les formats choisis pour les calculs intermédiaires, mais dans des directions opposées. En effet, utiliser des formats ayant des *lsbs* plus petits améliore la précision, mais augmente le nombre de bits manipulés ce qui diminue l'efficacité. L'enjeu est donc de maximiser les *lsbs* de ces formats et donc l'efficacité de l'algorithme, tout en s'assurant qu'on conserve une précision suffisante pour un contexte considéré.

J'illustrerai les algorithmes avec des figures représentant un exemple de positions de bits pour les produits intermédiaires, éventuellement d'autres résultats intermédiaires, et la sortie `res`. Par exemple, la figure 5.2 montre six produits $s_i t_i$ dans les formats respectifs $\mathbb{F}_{(\ell_{s_i}+\ell_{t_i})}$ (on a vu dans l'équation (5.13) que le produit exact $s_i t_i$ est bien représentable dans ce format) et la sortie qui est dans $\mathbb{F}_{\ell_f}$. Les bits de poids fort sont à gauche et ceux de poids faible à droite, donc les *lsbs* sont décroissants quand on va vers la droite. Je n'ai pas indiqué la position de la virgule car elle n'a pas d'importance pour les algorithmes que je considère ; ce qui compte est la position relative des *lsbs* : ici $\ell_f = \ell_{s_0}+\ell_{t_0}+6 = \ell_{s_1}+\ell_{t_1}+3$.

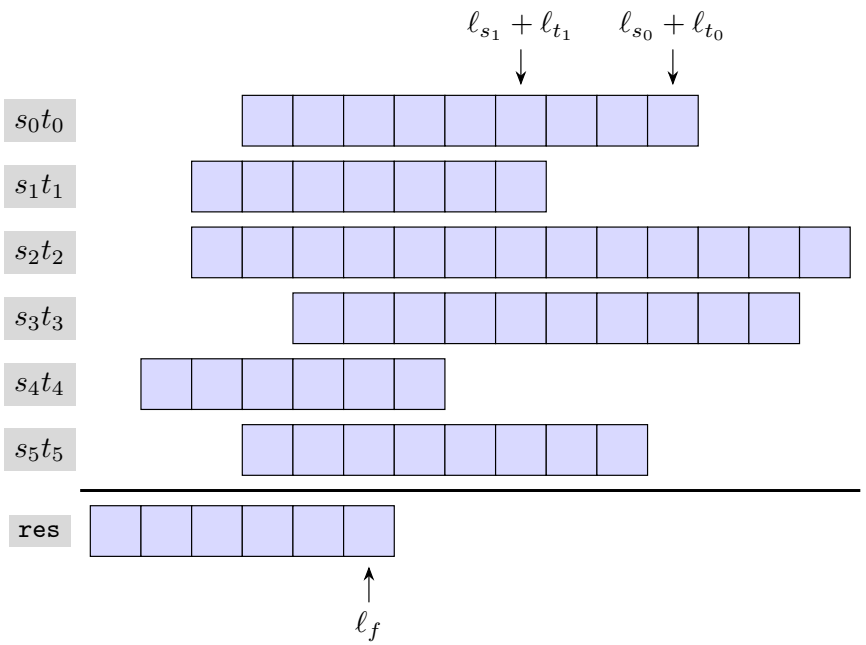


FIGURE 5.2 – Exemple : positions des bits des produits exacts et de la sortie

Précalculs comme le tri par *lsbs*. Les *lsbs* des entrées ℓ_{s_i} et ℓ_{t_i} et le *lsb* final ℓ_f sont connus à l'avance, c'est-à-dire au moment où l'algorithme est implémenté, contrairement aux valeurs des entrées elles-mêmes qui sont seulement reçues au moment de l'exécution. Ainsi, des opérations qui ne dépendent que de ces *lsbs* peuvent être précalculées, et n'affectent donc pas la complexité de l'algorithme lui-même. Par exemple, certains algorithmes supposent que les entrées sont triées de sorte que les *lsbs* assurant des produits exacts (c'est-à-dire

les sommes des *lsbs*) soient croissants : $\ell_{s_0} + \ell_{t_0} \leq \cdots \leq \ell_{s_{n-1}} + \ell_{t_{n-1}}$. Cette hypothèse peut être facilement assurée en considérant la bonne permutation de chaque suite d'entrées, permutation qui a été calculée à l'avance. En revanche, si on voulait trier les entrées par leurs propres valeurs, la complexité de l'algorithme prendrait en compte le coût de ce tri.

Formalisation. Voici les éléments communs aux formalisations en Coq des sections 5.2 et 5.3, notamment les variables et hypothèses à considérer.

On prend d'abord en paramètres les formats connus à l'avance : le *lsb* final ℓ_f ainsi que les *lsbs* des entrées, donnés sous la forme de deux séquences.

```
Variables (lf : Z) (ls lt : seq Z).
```

On se donne ensuite les deux séquences d'entrées (s_i) et (t_i). Elles sont à valeurs dans `R` : comme on a vu dans les sections 5.1.2 et 5.1.3, les nombres en virgule fixe sont représentés par des réels accompagnés quand nécessaire d'une hypothèse indiquant leur format. Par ailleurs, j'aurais pu ajouter comme hypothèse que les deux suites ont la même longueur. Mais pour alléger un tout petit peu le contexte, j'ai plutôt choisi de noter `n` le minimum de leurs tailles. Cela revient au même : si les suites ne font pas la même taille, les éléments excédentaires de la suite la plus longue seront ignorés.

```
Variables s t : seq R.
Notation n := (minn (size s) (size t)).
```

Pour indiquer que les *lsbs* des formats des entrées sont donnés par les suites `ls` et `lt`, on utilise la relation `are_lsb`. Celle-ci s'appuie sur `all2`, qui détermine si une relation donnée est vérifiée par les paires d'éléments respectifs de deux listes qui doivent avoir la même longueur, et sur `isFIX` défini dans la section 5.1.3.

```
Definition are_lsb (ls : seq Z) (s : seq R) : bool :=
  all2 isFIX ls s.
Hypothesis are_lsb_s : are_lsb ls s.
Hypothesis are_lsb_t : are_lsb lt t.
```

Par ailleurs, on définit `SOP_model`, qui correspond au résultat modèle *mres* en précision infinie.

```
Definition prod_seq_model : seq R := map2 Rmult s t.
Definition SOP_model : R := \sum_(r ← prod_seq_model) r.
```

Les théorèmes qu'on prouvera sur les sorties des différents algorithmes décriront la relation entre ces sorties et `SOP_model` (c'est-à-dire entre `res` et *mres*).

5.2.2 Accumulateur de Kulisch

L'accumulateur de Kulisch [75] est un des algorithmes de somme de produits les plus simples et naturels. Le principe est d'utiliser suffisamment de bits dans les calculs intermédiaires pour que ceux-ci soient tous exacts. Si on autorise l'algorithme à choisir le format de la sortie, il renvoie donc la somme de produits exacte *mres*. Mais comme on a vu en section 5.2.1, on va plutôt exiger que la sortie `res` soit dans le format donné $\mathbb{F}_{\ell_f}$. L'algorithme est alors correctement arrondi pour n'importe quelle fonction d'arrondi $\bigcirc_{\ell_f} : \mathbb{R} \to \mathbb{F}_{\ell_f}$ désirée (et

dont une implémentation est fournie). Autrement dit $\texttt{res} = \bigcirc_{\ell_f}(\mathit{mres})$. On choisit généralement un arrondi au plus proche (peu importe la règle de *tie-breaking* τ), auquel cas on obtient une borne sur l'erreur en sortie égale à un demi *ulp* c'est-à-dire $2^{\ell_f - 1}$. L'inconvénient principal de cet algorithme est que le nombre de bits impliqués dans les calculs est généralement très élevé, donc l'algorithme est lent et coûteux en énergie (surtout si les entrées ont des ordres de grandeur variés).

Description de l'algorithme. On utilise les notations de la section 5.2.1. On a vu que calculer le produit $s_i t_i$ dans le format $\mathbb{F}_{(\ell_{s_i} + \ell_{t_i})}$ assure un résultat exact. Puis comme une somme est représentable avec comme *lsb* le minimum des *lsbs* des termes (équation (5.12) et lemme 5.6), on ajoute les produits successivement en utilisant comme *lsb* :

$$\ell_{\min} \triangleq \min_{0 \le i < n} (\ell_{s_i} + \ell_{t_i}) \tag{5.22}$$

On calcule ainsi le résultat intermédiaire $q \in \mathbb{F}_{\ell_{\min}}$ suivant (l'ordre des additions n'a pas d'importance puisqu'on s'est arrangé pour qu'elles soient toutes exactes donc elles sont associatives) :

$$q \triangleq \bigtriangledown\!\!\!\!+_{\ell_{\min}, \, 0 \le i < n} \; (s_i \; \bigtriangledown\!\!\!\!\times_{\ell_{s_i} + \ell_{t_i}} \; t_i) \tag{5.23}$$

Enfin, ce résultat intermédiaire q est arrondi vers $\mathbb{F}_{\ell_f}$ en utilisant l'arrondi $\bigcirc_{\ell_f}$ pour lequel on souhaite avoir un algorithme correctement arrondi (souvent un arrondi au plus proche) :

$$\texttt{res} \triangleq \bigcirc_{\ell_f}(q) \tag{5.24}$$

Notons que n'importe quel arrondi peut être utilisé dans les opérations intermédiaires, puisque le *lsb* choisi pour chaque calcul assure que le résultat est exact de toute façon. L'arrondi vers le bas est utilisé par défaut car c'est le plus facile à implémenter en complément à deux (voir la section 5.4.1).

L'accumulateur de Kulisch est récapitulé par l'algorithme 11, et la succession d'opérations à effectuer est représentée sous forme de graphe sur la figure 5.3. La figure 5.4 illustre la position des bits des produits intermédiaires exacts, du résultat intermédiaire q et de la sortie `res` pour le même exemple jouet qu'en section 5.2.1.

Théorème 5.9. *L'accumulateur de Kulisch est correctement arrondi pour l'arrondi $\bigcirc_{\ell_f}$ choisi précédemment :*

$$\texttt{res} = \bigcirc_{\ell_f}(\mathit{mres}) \tag{5.25}$$

En particulier, si on l'arrondi choisi $\bigcirc_{\ell_f}$ est un arrondi au plus proche, on obtient comme borne d'erreur :

$$|\texttt{res} - \mathit{mres}| \le 2^{\ell_f - 1} \tag{5.26}$$

Preuve. On a vu que les *lsbs* des calculs intermédiaires ont été choisis pour que ces calculs soient exacts, donc :

$$q = \sum_{0 \le i < n} s_i t_i = \mathit{mres} \tag{5.27}$$

Algorithme 11 : Accumulateur de Kulisch

Donnés à l'avance : ℓ_f, n, ℓ_{s_0}, …, $\ell_{s_{n-1}}$, ℓ_{t_0}, …, $\ell_{t_{n-1}}$, un arrondi $\bigcirc_{\ell_f}$
Calculé à l'avance : $\ell_{\min} \triangleq \min_{0 \le i < n}(\ell_{s_i} + \ell_{t_i})$
Entrées : $s_0, \ldots, s_{n-1}, t_0, \ldots, t_{n-1}$
Sortie : `res`
Variable : v
$v \leftarrow s_0 \bigtriangledown_{\ell_{s_0}+\ell_{t_0}} t_0$
for $i \leftarrow 1$ **to** $n-1$ **do**
  $v \leftarrow v \bigtriangledown_{\ell_{\min}} (s_i \bigtriangledown_{\ell_{s_i}+\ell_{t_i}} t_i)$
end
`res` $\leftarrow \bigcirc_{\ell_f}(v)$

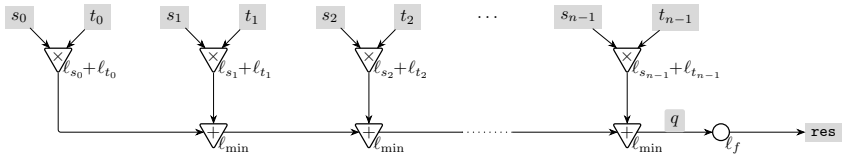


FIGURE 5.3 – Accumulateur de Kulisch : graphe des opérations

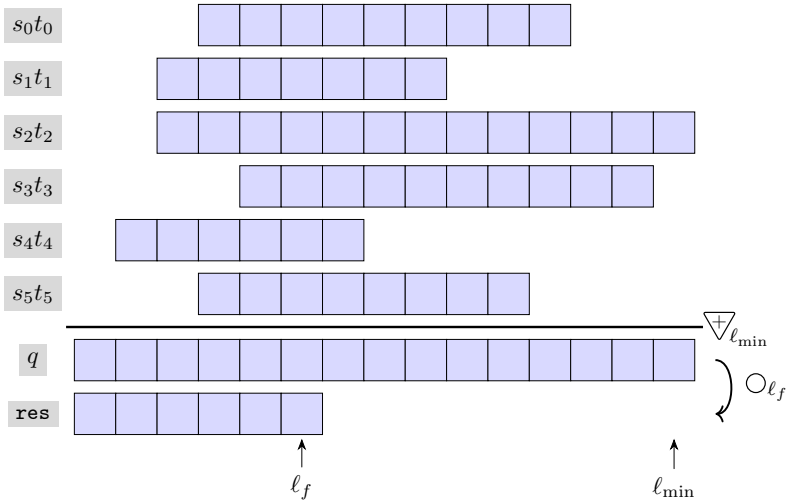


FIGURE 5.4 – Accumulateur de Kulisch : gestion des bits sur un exemple

On conclut trivialement grâce à l'équation (5.24). La borne d'erreur est ensuite un corollaire immédiat du lemme 5.3 bornant l'erreur d'un arrondi au plus proche. □

Un intérêt notable de l'accumulateur de Kulisch est que comme les additions sont toutes exactes, elles sont associatives et commutatives. En particulier, on peut les réordonner pour paralléliser leurs calculs sans affecter le résultat final `res`. Ou alors, on peut les ajouter en commençant par les *lsbs* les plus grands

pour réduire le nombre de bits manipulés dans les calculs : cette variante fera l'objet de la section 5.2.3.

Formalisation. Les variables `lf`, `ls`, `lt`, `s` et `t`, la notation `n` et les hypothèses `are_lsb_s` et `are_lsb_t` sont définies comme expliqué en section 5.2.1.

On se donne également deux modes d'arrondi, représentés par des fonctions $\mathbb{R} \to \mathbb{Z}$ comme expliqué dans les sections 5.1.2 et 5.1.3.

```
Variable rndf : R → Z.
Variable rndc : R → Z.
Hypothesis rndc_valid : Valid_rnd rndc.
```

Le second mode d'arrondi `rndc` sera utilisé pour toutes les opérations intermédiaires. Comme mentionné précédemment, ce sera généralement l'arrondi vers le bas, facile à implémenter en complément à deux. Mais n'importe quel arrondi vérifiant la condition `Valid_rnd` de Flocq (section 5.1.2) convient. Le premier mode d'arrondi `rndf` est celui désiré pour le résultat : celui qu'on note $\bigcirc_{\ell_f}$ tout au long de la section car ce sera souvent un arrondi au plus proche. Mais il peut bien s'agir de n'importe quel arrondi. On n'a même pas besoin de l'hypothèse qu'il vérifie `Valid_rnd`. En effet, on remarque que la preuve du théorème 5.9 montre que les valeurs auxquelles on applique cet arrondi sont déjà égales, donc on n'a besoin d'aucune propriété de l'arrondi lui-même.

Par ailleurs, on définit la suite des $\ell_{s_i} + \ell_{t_i}$, qui sont les *lsbs* respectifs des produits $s_i t_i$. On utilise pour cela la fonction `map2` qui permet d'appliquer un opérateur binaire aux éléments respectifs de deux listes. On se donne ensuite le *lsb* `lmin` à utiliser pour les calculs intermédiaires. Plutôt que de le définir comme le minimum des $\ell_{s_i} + \ell_{t_i}$, on se donne une hypothèse un peu plus générale : `lmin` doit être inférieur ou égal à ce minimum.

```
Notation lprods := (map2 Zplus ls lt).
Variable lmin : Z.
Hypothesis lmin_le : all (Z.leb lmin) lprods.
```

On définit ensuite la suite des produits. Chaque multiplication est faite avec le bon *lsb* grâce à `map3` qui applique une fonction à trois arguments aux éléments respectifs de trois listes. Puis on utilise l'opérateur `\big` de MathComp (section 1.2.5) pour ajouter les éléments de cette suite de produits avec pour *lsb* `lmin`. Enfin, on arrondit cette somme avec l'arrondi final `rndf` au *lsb* `lf`. On obtient ainsi `SOP_Kulisch` qui représente la sortie `res` de l'algorithme. On prouve alors que ce résultat est bien l'arrondi final du modèle *mres* représenté par `SOP_model` défini en section 5.2.1. La preuve Coq est facile est assez courte, comme suggéré par la preuve du théorème 5.9 ci-dessus.

```
Definition prods_Kulisch : seq R :=
  map3 (fun l ⇒ mulFIX l rndc) lprods s t.
Definition SOP_Kulisch : R :=
  roundFIX lf rndf
           (\big[addFIX lmin rndc / 0]_(x ← prods_Kulisch) x).
Theorem SOP_Kulisch_correct :
  SOP_Kulisch = roundFIX lf rnd (SOP_model s t).
```

5.2.3 Accumulateur de Kulisch par *lsb* décroissant

Cette section présente une variante de l'accumulateur de Kulisch décrit dans la section précédente. Le principe est de s'appuyer sur un tri des entrées selon leur *lsb* afin de réduire le nombre de bits impliqués dans les calculs (donc aussi le temps de calcul et l'énergie consommée).

Tri des entrées par *lsb* du produit. Comme expliqué en section 5.2.1, comme les *lsbs* sont connus à l'avance, on peut réordonner les entrées en fonctions de ces *lsbs* lors de l'implémentation de l'algorithme, sans affecter sa complexité à l'exécution. Dans cette section, on va donc supposer que les entrées sont triées de telle sorte que les *lsbs* $\ell_{s_i} + \ell_{t_i}$ permettant le calcul exact du produit $s_i t_i$ soient croissants :

$$\ell_{s_0} + \ell_{t_0} \leq \cdots \leq \ell_{s_{n-1}} + \ell_{t_{n-1}} \tag{5.28}$$

Pour alléger les notations, on pose :

$$\begin{aligned} \ell_i &\triangleq \ell_{s_i} + \ell_{t_i} \\ p_i &\triangleq s_i \mathbin{\overset{\times}{\nabla}_{\ell_i}} t_i = s_i t_i \in \mathbb{F}_{\ell_i} \end{aligned} \tag{5.29}$$

Le tri considéré assure donc qu'on a $\ell_0 \leq \cdots \leq \ell_{n-1}$.

Description et intérêt de l'algorithme. Comme en section 5.2.2, on calcule les produits exacts, pour lesquels on vient d'introduire la notation $p_i \in \mathbb{F}_{\ell_i}$. Mais cette fois-ci, on ajoute ces produits successivement en commençant par les plus grands indices, c'est-à-dire par *lsb* ℓ_i décroissant. Le *lsb* utilisé pour chaque addition est celui du nouveau produit qu'on est en train d'ajouter au total précédent. Cela garantit encore des additions exactes, puisque le nouveau *lsb* ℓ_i est bien inférieur aux *lsbs* de tous les produits déjà ajoutés grâce au tri considéré. On obtient alors :

$$q \triangleq ((p_{n-1} \mathbin{\overset{+}{\nabla}_{\ell_{n-2}}} p_{n-2}) \mathbin{\overset{+}{\nabla}_{\ell_{n-3}}} p_{n-3}) \cdots \mathbin{\overset{+}{\nabla}_{\ell_0}} p_0 \tag{5.30}$$

Puis comme dans la section précédente, on utilise l'arrondi choisi $\bigcirc_{\ell_f}$:

$$\texttt{res} \triangleq \bigcirc_{\ell_f}(q) \tag{5.31}$$

L'intérêt est que les *lsb* ℓ_i pour les additions sont plus grands que le *lsb* $\ell_{\min} \triangleq \min \ell_i$ précédemment utilisé dans toutes les additions. Cela signifie moins de bits manipulés dans les calculs, tout en gardant la même précision sur le résultat. L'inconvénient est que les additions doivent ici être effectuées en série donc l'algorithme est moins facilement parallélisable.

Algorithme de somme. On a vu que la seule particularité de l'algorithme ci-dessus est la façon dont les p_i sont sommés pour obtenir q. Le calcul des p_i et l'arrondi final de q vers `res` sont identiques à l'accumulateur de Kulisch basique. En particulier, comme les produits sont calculés exactement sans aucune originalité, on peut voir cette variante comme un algorithme de somme plutôt que de somme de produits. Les entrées sont alors les p_i de *lsbs* respectifs ℓ_i, et l'objectif est de calculer une bonne approximation de $\mathit{mres} = \sum_{0 \leq i < n} p_i$. Cet algorithme de somme est illustré par l'algorithme 12 et la figure 5.18. La figure 5.6

montre un exemple de positions des bits. Les entrées p_i sont bien les $s_i t_i$ de la figure 5.4 qui illustrait l'accumulateur de Kulisch basique, mais réordonnés pour avoir $\ell_0 \leq \cdots \leq \ell_5$. On les somme cette fois-ci en commençant par p_5 et p_4, et en finissant pas p_0. En plus des p_i, de q et de `res`, je représente en rouge les sommes partielles : on observe que leur *lsb* est bien le minimum des *lsbs* des entrées déjà sommées.

Algorithme 12 : Somme exacte par *lsb* décroissant

Donnés à l'avance : ℓ_f, n, $\ell_0 \leq \cdots \leq \ell_{n-1}$, un arrondi $\bigcirc_{\ell_f}$
Entrées : $p_0, \ldots, p_{n-1}$
Sortie : `res`
Variable : v
$v \leftarrow p_{n-1}$
for $i \leftarrow n-2$ **downto** 0 **do**
  $v \leftarrow v \mathbin{\overset{+}{\bigtriangledown}_{\ell_i}} p_i$
end
`res` $\leftarrow \bigcirc_{\ell_f}(v)$

Théorème 5.10. *Cet algorithme est correctement arrondi pour l'arrondi* $\bigcirc_{\ell_f}$ *choisi :*

$$\texttt{res} = \bigcirc_{\ell_f}(\mathit{mres}) \tag{5.32}$$

En particulier, lorsqu'on choisit un arrondi au plus proche, la borne d'erreur est :

$$|\texttt{res} - \mathit{mres}| \leq 2^{\ell_f - 1} \tag{5.33}$$

Preuve. Immédiate à partir du lemme 5.11 suivant (et du lemme 5.3 bornant l'erreur d'un arrondi au plus proche). □

Lemme 5.11. *Le calcul de q est est en fait exact :*

$$q \triangleq ((p_{n-1} \mathbin{\overset{+}{\bigtriangledown}_{\ell_{n-2}}} p_{n-2}) \mathbin{\overset{+}{\bigtriangledown}_{\ell_{n-3}}} p_{n-3}) \cdots \mathbin{\overset{+}{\bigtriangledown}_{\ell_0}} p_0 = \sum_{0 \leq i < n} p_i \triangleq \mathit{mres} \tag{5.34}$$

Preuve. Comme $\ell_{n-2} \leq \ell_{n-1}$, on a $p_{n-1} \in \mathbb{F}_{\ell_{n-1}} \subset \mathbb{F}_{\ell_{n-2}}$ et $p_{n-2} \in \mathbb{F}_{\ell_{n-2}}$. L'addition suivante est donc exacte : $p_{n-1} \mathbin{\overset{+}{\bigtriangledown}_{\ell_{n-2}}} p_{n-2} = p_{n-1} + p_{n-2}$. Et bien sûr, le résultat d'un arrondi au *lsb* ℓ_{n-2} est dans $\mathbb{F}_{\ell_{n-2}}$. Avec les mêmes arguments, on prouve facilement par récurrence descendante :

$$\begin{aligned} &\forall i \in \{n-2, n-3, \ldots, 1, 0\}, \\ &\quad ((p_{n-1} \mathbin{\overset{+}{\bigtriangledown}_{\ell_{n-2}}} p_{n-2}) \mathbin{\overset{+}{\bigtriangledown}_{\ell_{n-3}}} p_{n-3}) \cdots \mathbin{\overset{+}{\bigtriangledown}_{\ell_i}} p_i = \sum_{i \leq j < n} p_j \in \mathbb{F}_{\ell_j} \end{aligned} \tag{5.35}$$

□

Formalisation. La partie intéressante à prouver est le lemme 5.11, puisque le reste est identique à l'accumulateur de Kulisch classique de la section 5.2.2.

On se donne la liste `ps` des produits à ajouter, avec la liste `ls` de leurs *lsbs* respectifs. On suppose aussi que ces *lsbs* sont triés selon `Z.leb`, qui est la relation booléenne représentant $\leq$ sur $\mathbb{Z}$.

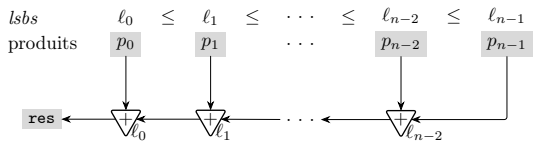


FIGURE 5.5 – Accumulateur par *lsb* décroissant : graphe des opérations

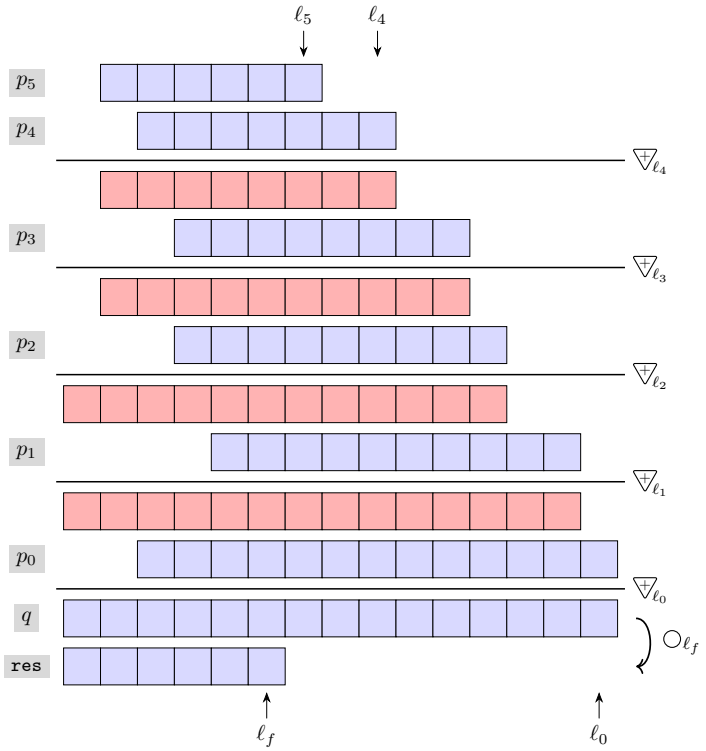


FIGURE 5.6 – Accumulateur par *lsb* décroissant : gestion des bits sur un exemple

```
Variable ls : seq Z.
Variable ps : seq R.
Hypothesis are_lsb_ls_ps : are_lsb ls ps.
Hypothesis ls_srt : sorted Z.leb ls.
Notation n := (size ps).
```

On définit récursivement les sommes partielles du calcul de q (équation (5.30)) :

$$\begin{cases} v_0 \triangleq p_{n-1} \\ v_i \triangleq v_{i-1} \boxplus_{\ell_{n-1-i}} p_{n-1-i} \quad \text{pour } 1 \leq i \leq n-1 \end{cases} \tag{5.36}$$

La notation `s ' i` représente l'élément d'indice `i` d'une liste `s`, pour `0 ≤ i < size s`. On rappelle que `m.+1` et `m.-1` sont respectivement le successeur et le prédécesseur de `m` (avec `0.-1 = 0` pour rester dans `nat`).

```
Fixpoint vi (i : nat) :=
  match i with
    | 0 ⇒ ps ' n.-1
    | S j ⇒ addFIX (ls ' (n.-1 - j.+1)%nat) Zfloor
               (vi j) (ps ' (n.-1 - j.+1)%nat)
  end.
```

On a dit qu'on s'intéresse seulement au lemme 5.11 qui concerne le résultat intermédiaire q, simplement obtenu en considérant v_{n-1}.

```
Definition summation_decreasinglsb : R := vi n.-1.
```

On prouve enfin le lemme 5.11. La preuve récursive ne pose pas de difficulté.

```
Theorem summation_decreasinglsb_exact :
  summation_decreasinglsb = \sum_(x ← ps) x.
```

5.2.4 Algorithme fidèle avec des bits de garde

Cette section présente une version allégée d'un algorithme de somme de produits conçu spécifiquement pour calculer des filtres numériques [111]. Il est moins précis que l'accumulateur de Kulisch : il garantit seulement que la sortie `res` est un arrondi fidèle (section 5.1.1, définition 5.4) du résultat modèle *mres*. Son intérêt est qu'il est plus efficace : en général, les calculs font intervenir beaucoup moins de bits qu'un accumulateur de Kulisch. De plus, l'algorithme original met à profit le fait que les s_i soient des constantes connues à l'avance dans le cas d'un filtre numérique, mais cette optimisation n'est pas étudiée ici. La version que je présente ne suppose donc rien de particulier sur les s_i.

L'idée est d'utiliser un *lsb de calcul* ℓ_c bien choisi pour toutes les opérations intermédiaires (à la fois multiplications et additions binaires). On veut que ce *lsb* soit le plus grand possible pour diminuer le nombre de bits à manipulé, mais aussi suffisamment petit pour assurer l'obtention d'un arrondi fidèle. On verra dans la preuve du théorème 5.12 que la valeur qui accomplit ceci est :

$$\ell_c \triangleq \ell_f - 1 - \lceil \log_2 n \rceil \tag{5.37}$$

(On utilise toujours les notations de la section 5.2.1 et on rappelle que n est la longueur des suites d'entrées (s_i) et (t_i).) Les bits entre ℓ_c inclus et ℓ_f exclu sont appelés les *bits de garde*, d'où le nom d'*algorithme fidèle avec des bits de garde*.

Tous les calculs intermédiaires, à savoir les multiplications deux à deux et les additions successives des produits obtenus, sont effectuées dans le format $\mathbb{F}_{\ell_c}$:

$$q \triangleq \bigtriangledown_{\ell_c,\, 0 \le i < n} \left(s_i \bigtriangledown_{\ell_c} t_i\right) \tag{5.38}$$

Les produits approchés sont dans $\mathbb{F}_{\ell_c}$ donc les sommes arrondies vers $\mathbb{F}_{\ell_c}$ sont exactes. Cela signifie aussi qu'elles sont associatives et commutatives donc l'ordre n'a pas d'importance, ce qui nous autorise à employer une notation de somme similaire au $\sum$ usuel. Encore une fois, n'importe quel arrondi valide peut être utilisé pour ces opérations, mais l'arrondi vers la bas est choisi par défaut car il est facile à implémenter en complément à deux.

La sortie est ensuite obtenue en appliquant un arrondi au plus proche vers $\mathbb{F}_{\ell_f}$ à ce résultat intermédiaire :

$$\texttt{res} \triangleq \bigcirc^{\tau}_{\ell_f}(q) \tag{5.39}$$

La règle de *tie-breaking* τ n'a pas d'importance ici : n'importe laquelle convient. On peut par exemple utiliser l'arrondi au plus proche pair usuel (règle *even*).

L'algorithme est récapitulé dans l'algorithme 13. La succession d'opérations à effectuer est représentée sous forme de graphe sur la figure 5.7. La figure 5.8 illustre les bits impliqués sur un exemple. On voit que les bits à droite de ℓ_c des produits exacts $s_i t_i$ ne sont pas utilisés.

Algorithme 13 : Algorithme fidèle avec des bits de garde

Donnés à l'avance : ℓ_f, n (et n'importe quelle règle de *tie-breaking* τ)
Calculé à l'avance : $\ell_c \triangleq \ell_f - 1 - \lceil \log_2 n \rceil$
Entrées : $s_0, \ldots, s_{n-1}, t_0, \ldots, t_{n-1}$
Sortie : `res`
Variable : v
$v \leftarrow s_0 \bigtriangledown_{\ell_c} t_0$
for $i \leftarrow 1$ **to** $n-1$ **do**
  $v \leftarrow v \bigtriangledown_{\ell_c} (s_i \bigtriangledown_{\ell_c} t_i)$
end
`res` $\leftarrow \bigcirc^{\tau}_{\ell_f}(v)$

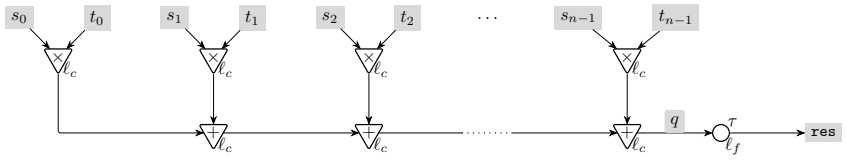


FIGURE 5.7 – Algorithme fidèle : graphe des opérations

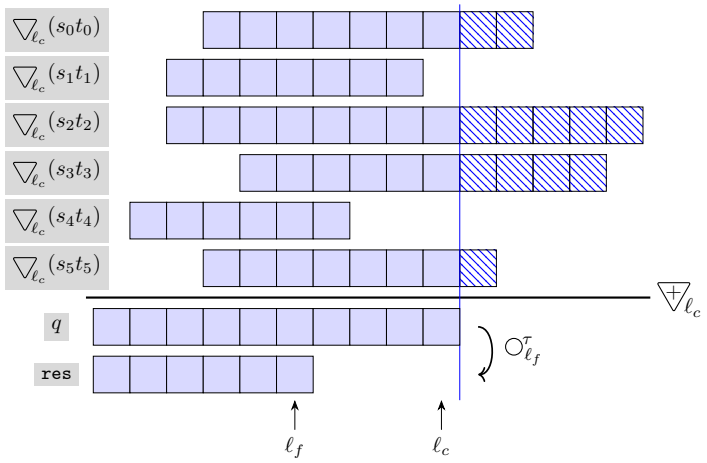


FIGURE 5.8 – Algorithme fidèle : gestion des bits sur un exemple

Théorème 5.12. *La sortie* `res` *de l'algorithme avec des bits de garde décrit ci-dessus est un arrondi fidèle dans* $\mathbb{F}_{\ell_f}$ *du résultat modèle* $mres$.

Cela signifie que la borne d'erreur de cet algorithme est :

$$|\texttt{res} - mres| < 2^{\ell_f} \tag{5.40}$$

Preuve. L'erreur sur un arrondi vers le bas est strictement inférieure à un *ulp* (lemme 5.1) donc :

$$\forall i, \quad \left| s_i \mathbin{\overline{\otimes}_{\ell_c}} t_i - s_i t_i \right| < 2^{\ell_c} \tag{5.41}$$

Par ailleurs, on a vu que la somme arrondie vers $\mathbb{F}_{\ell_c}$ de deux nombres déjà dans $\mathbb{F}_{\ell_c}$ est exacte (équation (5.12)) donc :

$$q \triangleq \overline{\sum}_{\substack{\ell_c \\ 0 \le i < n}} (s_i \mathbin{\overline{\otimes}_{\ell_c}} t_i) = \sum_{0 \le i < n} (s_i \mathbin{\overline{\otimes}_{\ell_c}} t_i) \tag{5.42}$$

On a alors :

$$|q - mres| = \left| \sum_{0 \le i < n} (s_i \mathbin{\overline{\otimes}_{\ell_c}} t_i) - s_i t_i \right| \le \sum_{0 \le i < n} \left| (s_i \mathbin{\overline{\otimes}_{\ell_c}} t_i) - s_i t_i \right| < n\, 2^{\ell_c} \tag{5.43}$$

Enfin, l'erreur sur un arrondi au plus proche est au plus un demi *ulp* (lemme 5.3). On a donc :

$$|\texttt{res} - mres| \le \left| \bigcirc_{\ell_f}(q) - q \right| + |q - mres| < 2^{\ell_f - 1} + n\, 2^{\ell_c} \tag{5.44}$$

Or d'après le lemme 5.5, `res` est un arrondi fidèle de $mres$ dans $\mathbb{F}_{\ell_f}$ si et seulement si :

$$|\texttt{res} - mres| < 2^{\ell_f} \tag{5.45}$$

Par transitivité, il suffit donc de prouver :

$$\begin{aligned} 2^{\ell_f - 1} + n\, 2^{\ell_c} \le 2^{\ell_f} \quad & i.e. \quad n \le 2^{\ell_f - 1 - \ell_c} \\ & i.e. \quad \ell_c \le \ell_f - 1 - \log_2 n \\ & i.e. \quad \ell_c \le \lfloor \ell_f - 1 - \log_2 n \rfloor = \ell_f - 1 - \lceil \log_2 n \rceil \end{aligned} \tag{5.46}$$

Ce qui est vrai puisqu'on a justement défini ℓ_c comme :

$$\ell_c \triangleq \ell_f - 1 - \lceil \log_2 n \rceil \qquad \text{(rappel de (5.37))}$$

On a ainsi prouvé que `res` est bien un arrondi fidèle de $mres$ dans $\mathbb{F}_{\ell_f}$. De plus, on remarque que ℓ_c est le plus grand *lsb* pour lequel cette preuve fonctionne.

□

Notons que cet algorithme ne dépend pas explicitement des formats des entrées : les ℓ_{s_i} et ℓ_{t_i} n'apparaissent pas.

Comme les additions des produits approchés sont exactes, l'ordre dans lequel elles sont effectuées n'affecte pas le résultat. On peut donc facilement les paralléliser, comme pour l'accumulateur de Kulisch en section 5.2.2.

On peut aussi accélérer légèrement l'algorithme en utilisant l'accumulateur par *lsb* décroissant de la section 5.2.3 uniquement pour les produits représentables sur un *lsb* plus grand que ℓ_c, c'est-à-dire les $s_i t_i$ tels que $\ell_{s_i} + \ell_{t_i} \ge \ell_c$.

En effet, les *lsbs* de calcul de cet accumulateur sont ceux des produits exacts, donc tant qu'ils sont plus grands que ℓ_c, cela diminue le nombre de bits impliqués. Pour les i tels que $\ell_{s_i} + \ell_{t_i} < \ell_c$, on continue à utiliser l'algorithme à bits de garde. Comme les deux algorithmes effectuent des additions exactes, on peut ajouter les sommes obtenues par chacun et on obtient la même somme intermédiaire q que précédemment, qu'on peut ensuite arrondir au plus proche vers $\mathbb{F}_{\ell_f}$.

Variante : multiplications arrondies au plus proche. On a dit que l'algorithme fonctionne, c'est-à-dire renvoie un arrondi fidèle, quel que soit l'arrondi valide utilisé dans les opérations intermédiaires. Mais utiliser un arrondi plus précis, par exemple un arrondi au plus proche, permet potentiellement d'utiliser un *lsb* de calcul ℓ_c un peu plus grand que la valeur donnée par l'équation (5.37). On obtient des calculs faisant intervenir moins de bits, au prix d'un arrondi plus difficile à calculer que l'arrondi vers le bas.

Le théorème suivant donne la nouvelle expression pour ℓ_c lorsque les multiplications sont arrondies au plus proche. Notons que les additions sont exactes de toute façon, donc on ne gagnerait rien à exiger un arrondi particulier pour celles-ci.

Théorème 5.13. *La sortie de l'algorithme avec des bits de garde est encore un arrondi fidèle du résultat modèle si l'arrondi utilisé pour les multiplications est un arrondi au plus proche (avec n'importe quelle règle de tie-breaking) et le lsb de calcul est :*

$$\ell_c \triangleq \ell_f - 1 - \lfloor \log_2 n \rfloor \tag{5.47}$$

Preuve. La preuve est similaire à celle du théorème 5.12. L'erreur sur chaque multiplication n'est plus majorée strictement par 2^{ℓ_c}, mais largement par $2^{\ell_c - 1}$ vu qu'on utilise un arrondi au plus proche. Les produits sont à nouveau dans $\mathbb{F}_{\ell_c}$ donc les additions successives sont exactes. L'arrondi final au plus proche n'a pas changé. Au lieu de l'équation (5.44), on obtient donc :

$$|\texttt{res} - \mathit{mres}| \leq 2^{\ell_f - 1} + n\, 2^{\ell_c - 1} \tag{5.48}$$

Montrer que `res` est un arrondi fidèle de mres dans $\mathbb{F}_{\ell_f}$ revient encore à prouver :

$$|\texttt{res} - \mathit{mres}| < 2^{\ell_f} \tag{5.49}$$

Par transitivité, il suffit donc de prouver :

$$\begin{aligned} 2^{\ell_f - 1} + n\, 2^{\ell_c - 1} < 2^{\ell_f} \quad & i.e. \quad n < 2^{\ell_f - \ell_c} \\ & i.e. \quad \ell_c < \ell_f - \log_2 n \\ & i.e. \quad \ell_c \leq \lceil \ell_f - \log_2 n \rceil - 1 = \ell_f - 1 - \lfloor \log_2 n \rfloor \end{aligned} \tag{5.50}$$

Ceci est justement assuré par la définition de ℓ_c dans l'équation (5.47). □

En comparant les équations (5.37) et (5.47), on constate que les *lsbs* de calcul ne diffèrent que quand n est une puissance de 2, auquel cas leur différence est 1. Le nombre d'entrées n est connu à l'avance, donc peut informer le choix de l'algorithme. La variante ci-dessus n'est donc envisageable que si n est une puissance de 2. Même dans ce cas, ce ne sera généralement pas rentable d'utiliser

un arrondi au plus proche, plus coûteux que l'arrondi vers le bas, pour alléger les opérations d'un bit seulement.

Formalisation. La formalisation couvre à la fois l'algorithme présenté initialement où les multiplications sont arrondies vers le bas, et la variante où elles sont arrondies au plus proche.

On prend à nouveau en paramètres le *lsb* final souhaité `lf` et les suites d'entrées. On se donne également un *lsb* de calcul `lc` quelconque pour l'instant. La valeur que doit prendre ce *lsb* de calcul sera une hypothèse des théorèmes plus bas. Cela permettra notamment d'avoir une valeur différente pour la variante où les multiplications sont arrondies au plus proche. En revanche, on a vu que l'algorithme ne dépend pas des *lsbs* des entrées, qui n'apparaissent donc pas dans la formalisation.

```
Variable lf : Z.
Variables s t : seq R.
Notation n := (minn (size s) (size t)).
Variable lc : Z.
```

On définit ensuite l'algorithme en fonctions d'arrondis donnés pour les multiplications, les additions et l'arrondi final. La sortie `res` de l'algorithme est représentée par `SOP_guard`.

```
Definition prods_guard (rndmul : R → Z) : seq R :=
  map2 (mulFIX lc rndmul) s t.
Definition SOP_guard_aux (rndmul rndadd : R → Z) : R :=
  \big[addFIX lc rndadd / 0]_(x ← prods_guard rndmul) x.
Definition SOP_guard (rndmul rndadd rndf : R → Z) : R :=
  roundFIX lf rndf (SOP_guard_aux rndmul rndadd).
```

On prouve alors le théorème 5.12. On rappelle que l'arrondi final doit être au plus proche avec n'importe quelle règle de *tie-breaking* `choicef`. Les arrondis pour les opérations n'ont pas d'importance tant qu'ils sont valides. On suppose seulement que le *lsb* de calcul est inférieur ou égal à la valeur donnée par l'équation (5.37), vu que cela suffit pour prouver le théorème. La fonction `log2_ceil` envoie l'entier $n \in \mathbb{N}$ vers $\lceil \log_2 n \rceil \in \mathbb{Z}$. Enfin, le prédicat `faithful` défini en section 5.1.3 caractérise un arrondi fidèle. La preuve suit celle du théorème 5.12 avec de nombreuses étapes supplémentaires mais sans grande difficulté.

```
Theorem SOP_faithful (rndmul rndadd : R → Z)
                     (choicef : Z → bool) :
  Valid_rnd rndmul → Valid_rnd rndadd →
  (lc ≤ lf - 1 - log2_ceil n)%Z →
  faithful lf SOP_model (SOP rndmul rndadd (Znearest choicef)).
```

On prouve similairement la variante du théorème 5.13 où les multiplications sont arrondies au plus proche.

```
Theorem SOP_rN_faithful (choicemul : Z → bool) (rndadd : R → Z)
                        (choicef : Z → bool) :
  Valid_rnd rndadd →
  (lc ≤ lf - 1 - log2_floor n)%Z →
  faithful lf SOP_model (SOP (Znearest choicemul) rndadd
           (Znearest choicef)).
```

5.3 Algorithme correctement arrondi au plus proche

Motivation. On a vu que l'accumulateur de Kulisch en section 5.2.2 est correctement arrondi au plus proche (pour la règle de *tie-breaking* de notre choix), mais peu efficace (beaucoup de bits manipulés, donc temps de calcul et consommation d'énergie élevés). Sa variante en section 5.2.3 est plus efficace et offre la même précision. L'algorithme à bits de garde de la section 5.2.4 est souvent beaucoup plus efficace dans le contexte d'un filtre numérique. Cependant, il ne garantit qu'un arrondi fidèle.

Comme on a vu en section 1.3.3, un avantage d'un algorithme correctement arrondi est qu'il est déterministe, au sens où la sortie pour des entrées donnée peut prendre une unique valeur. Au contraire, pour un arrondi fidèle, il y a deux possibilités distinctes $\bigtriangledown_{\ell_f}(\mathit{mres})$ et $\bigtriangleup_{\ell_f}(\mathit{mres})$ dès que $\mathit{mres} \notin \mathbb{F}_{\ell_f}$. De plus, un algorithme correctement arrondi au plus proche dans $\mathbb{F}_{\ell_f}$ a une borne d'erreur en sortie de 2^{ℓ_f-1}, contre 2^{ℓ_f} pour un arrondi fidèle. C'est d'ailleurs la meilleure borne dans le pire cas qu'on puisse avoir pour un résultat dans $\mathbb{F}_{\ell_f}$ vu qu'un point milieu est à distance au moins 2^{ℓ_f-1} de tout élément de $\mathbb{F}_{\ell_f}$.

Cette section présente un algorithme original correctement arrondi au plus proche vers $\mathbb{F}_{\ell_f}$ et plus efficace que l'accumulateur de Kulisch et sa variante. Comme en section 5.2.3, il s'agit en fait d'un algorithme de somme, qui peut être utilisé comme algorithme de somme de produits en calculant les produits dans des formats garantissant qu'ils sont exacts :

$$\begin{aligned} \ell_i &\triangleq \ell_{s_i} + \ell_{t_i} \\ p_i &\triangleq s_i \bigtriangledown_{\ell_i} t_i = s_i t_i \in \mathbb{F}_{\ell_i} \end{aligned} \qquad \text{(rappel de (5.29))}$$

avec les notations de la section 5.2.1.

Pour énoncer le nouvel algorithme comme un algorithme de somme, on considère dans cette section que les entrées sont directement les $(p_i)_{0\le i<n}$ dont on connaît à l'avance les *lsbs* respectifs ℓ_i. La sortie idéale dont on veut calculer un arrondi au plus proche est donc :

$$\mathit{mres} \triangleq \sum_{0\le i<n} p_i \tag{5.51}$$

Le *lsb* désiré pour la sortie `res` est toujours noté ℓ_f. On considère parfois aussi une règle de *tie-breaking* désirée τ pour l'arrondi au plus proche selon lequel l'algorithme doit être correctement arrondi. La sortie sera alors notée $\texttt{res}_\tau$ et on devra prouver que $\texttt{res}_\tau = \bigcirc^{\tau}_{\ell_f}(\mathit{mres})$.

Comme expliqué en section 5.2.1 et utilisé en section 5.2.3, on peut supposer les entrées triées selon leur *lsb*. Ici aussi, on suppose qu'on a :

$$\ell_0 \le \cdots \le \ell_{n-1} \tag{5.52}$$

Principe de l'algorithme et plan de la section. Le principe du nouvel algorithme de somme est de sommer séparément les entrées p_i à petit *lsb* $\ell_i < \ell_f$ et celles à grand *lsb* $\ell_i \ge \ell_f$.

Pour celles à grand *lsb*, on utilise simplement l'accumulation par *lsb* décroissant de la section 5.2.3, dont les sommes partielles sont toutes exactes. Les *lsbs*

utilisés pour calculer les additions successives sont alors ℓ_{n-2}, ℓ_{n-3}, ℓ_{n-4}, etc. Mais ici ces *lsbs* sont grands, c'est-à-dire supérieurs ou égaux à ℓ_f. Cela signifie d'une part que tous les bits considérés sont de poids plus fort que le bit en ℓ_f, donc sont importants pour un résultat final dans $\mathbb{F}_{\ell_f}$. D'autre part, on a vu que pour manipuler moins de bits, on veut entre autres utiliser des *lsbs* de calcul les plus grands possibles. Or ceux utilisés ici sont déjà relativement grands. Pour ces deux raisons, l'accumulation exacte par *lsb* décroissant est particulièrement adaptée pour les entrées de grand *lsb*.

Pour sommer les termes à petit *lsb*, on utilise un algorithme original dont les *lsbs* de calculs sont légèrement plus grands que dans une accumulation par *lsb* décroissant. Les sommes partielles ne sont pas exactes, mais ont une précision suffisante pour assurer l'obtention d'un résultat final correctement arrondi au plus proche. Cet algorithme est décrit en section 5.3.1.

L'algorithme complet pour des *lsbs* de toutes tailles (indiquant comment additionner les sommes obtenues séparément pour les deux groupes d'entrées) est présenté en section 5.3.2.

Enfin, la section 5.3.3 présente un autre algorithme original de somme [17], lui aussi correctement arrondi au plus proche (pour la règle de *tie-breaking* τ demandée). Cet algorithme est basé sur l'utilisation de l'arrondi impair. Cependant, il est rendu obsolète par celui de la section 5.3.2, qui est un peu plus efficace pour la même précision.

5.3.1 Somme des termes à petit *lsb*

Cette section présente l'algorithme utilisé pour sommer les entrées p_i à petit *lsb* : $\ell_i < \ell_f$. Pour décrire cet algorithme, on suppose ici que toutes les entrées sont à petit *lsb*, c'est-à-dire que les *lsbs* vérifient :

$$\ell_0 \leq \cdots \leq \ell_{n-1} < \ell_f \tag{5.53}$$

Les termes p_i sont sommés par *lsb* croissant. Chaque p_i est ajouté à la somme partielle précédente en utilisant l'addition $\overset{+}{\triangledown}_{\ell_{i+1}}$ arrondie vers le bas au *lsb* ℓ_{i+1} du terme suivant. Le premier terme p_0 est simplement arrondi vers le bas vers $\mathbb{F}_{\ell_1}$ (ce qui revient au cas général si on considère que la somme partielle initiale vaut 0 : en effet $\triangledown_{\ell_1}(p_0) = 0 \overset{+}{\triangledown}_{\ell_1} p_0$). Quant au dernier produit p_{n-1}, comme il n'y a pas de terme suivant, le *lsb* utilisé pour son ajout est toujours $\ell_f - 1$. On pose la convention suivante pour éviter de devoir écrire un cas particulier pour p_{n-1} :

$$\ell_n \triangleq \ell_f - 1 \tag{5.54}$$

Notons que les ℓ_i restent croissants puisque $\ell_{n-1} < \ell_f$ par hypothèse, c'est-à-dire $\ell_{n-1} \leq \ell_n$.

Autrement dit, les sommes partielles calculées sont :

$$\begin{cases} v_0 \triangleq \triangledown_{\ell_1}(p_0) \\ v_i \triangleq v_{i-1} \overset{+}{\triangledown}_{\ell_{i+1}} p_i \quad \text{pour } 1 \leq i \leq n-1 \end{cases} \tag{5.55}$$

Le résultat intermédiaire q similaire à ceux des autres algorithmes est alors défini par :

$$q \triangleq v_{n-1} \tag{5.56}$$

Enfin, on arrondit q au plus proche vers $\mathbb{F}_{\ell_f}$:

$$\texttt{res} \triangleq \bigcirc_{\ell_f}^{\textit{ties-up}}(q) \tag{5.57}$$

Notons que contrairement aux algorithmes de la section 5.2, on ne peut pas utiliser n'importe quelle règle de *tie-breaking* ici. On verra dans la preuve du théorème 5.14 qu'on a besoin de la règle *ties-up* spécifiquement. Celle-ci consiste à systématiquement arrondir les points milieu vers le haut. L'arrondi au plus proche utilisant cette règle est parfois dit *upwards*, et apparaît dans la norme de SystemC [64] sous le nom de `SC_RND`. On verra en section 5.4.1 que cette règle est assez facile à implémenter en complément à deux : c'est le pendant pour cette représentation de la règle *away-from-zero* pour la virgule flottante. Par ailleurs, on a vu en section 5.1.2 que du point de vue de la bibliothèque Flocq [20], une règle de *tie-breaking* est une fonction `choice : Z → bool` qui indique pour chaque point milieu de quel côté l'arrondir. Dans ce formalisme, la règle *ties-up* est simplement la fonction constante $z \mapsto \textit{true}$.

L'algorithme décrit ci-dessus est récapitulé par l'algorithme 14 et la figure 5.9. La figure 5.10 montre la gestion des bits sur le même exemple jouet qu'en section 5.2, avec les sommes partielles en rouge.

Algorithme 14 : Somme de termes à petit *lsb*

Donnés à l'avance : n, $\ell_0 \leq \cdots \leq \ell_{n-1} < \ell_f$
Entrées : $p_0, \ldots, p_{n-1}$
Sortie : `res`
Variables : v
$v \leftarrow 0$
for $i \leftarrow 0$ **to** $n-1$ **do**
  $v \leftarrow v \mathbin{\overset{+}{\bigtriangledown}_{\ell_{i+1}}} p_i$ `// par convention` $\ell_n \triangleq \ell_f - 1$
end
`res` $\leftarrow \bigcirc_{\ell_f}^{\textit{ties-up}}(v)$

Théorème 5.14. *Cet algorithme est correctement arrondi pour l'arrondi au plus proche utilisant la règle ties-up :*

$$\texttt{res} = \bigcirc_{\ell_f}^{\textit{ties-up}}(\mathit{mres}) \tag{5.58}$$

En particulier, on a la borne d'erreur :

$$|\texttt{res} - \mathit{mres}| \leq 2^{\ell_f - 1} \tag{5.59}$$

La preuve de ce théorème reposera sur les deux lemmes suivants. Ils seront d'ailleurs réutilisés séparément pour prouver l'algorithme complet (pour des *lsbs* de toutes tailles) de la section 5.3.2.

Lemme 5.15. *Le résultat intermédiaire q est l'arrondi vers le bas dans $\mathbb{F}_{\ell-1}$ de la somme modèle :*

$$q = \bigtriangledown_{\ell_f - 1}(\mathit{mres}) \tag{5.60}$$

Preuve. Montrons par récurrence sur $i \in \{0, 1, \ldots, n-1\}$ que :

$$v_i = \bigtriangledown_{\ell_{i+1}}\Big(\sum_{0 \leq j \leq i} p_j\Big) \tag{5.61}$$

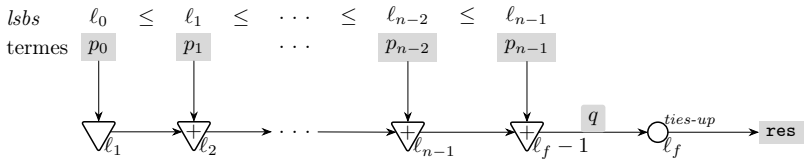


FIGURE 5.9 – Somme de termes à petit *lsb* : graphe des opérations

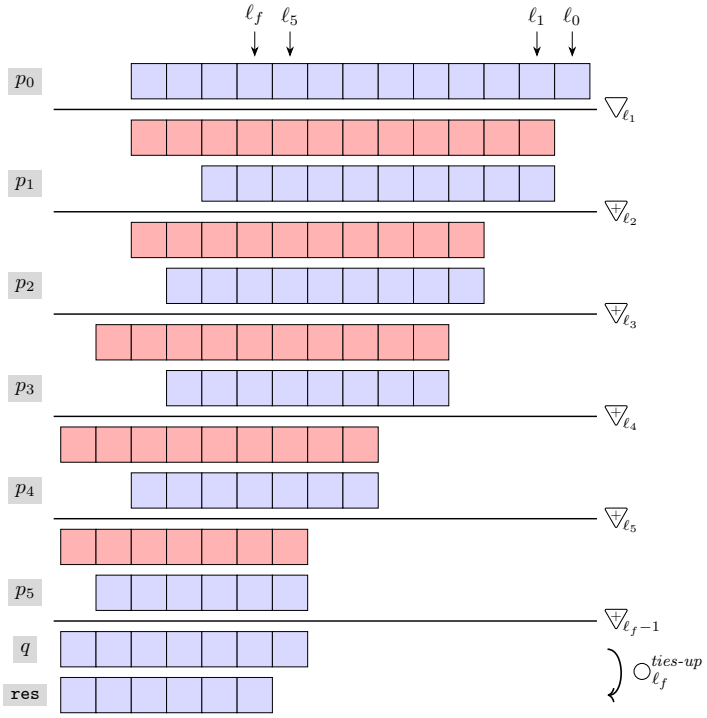


FIGURE 5.10 – Somme de termes à petit *lsb* : gestion des bits sur un exemple

— Pour $i = 0$, on a bien $v_0 \triangleq \bigtriangledown_{\ell_1}(p_0)$.
— Soit $0 \leq i \leq n-2$ vérifiant l'hypothèse de récurrence. Alors :

$$\begin{aligned} v_{i+1} \triangleq v_i \mathbin{\dot{+}}_{\ell_{i+2}} p_{i+1} &= \bigtriangledown_{\ell_{i+2}}\Big(\bigtriangledown_{\ell_{i+1}}\Big(\sum_{0\leq j\leq i} p_j \Big) + p_{i+1} \Big) \\ &= \bigtriangledown_{\ell_{i+2}}\Big(\bigtriangledown_{\ell_{i+1}}\Big(\sum_{0\leq j\leq i} p_j + p_{i+1}\Big) \Big) \qquad \text{(lemme 5.7)} \\ &= \bigtriangledown_{\ell_{i+2}}\Big(\sum_{0\leq j\leq i} p_j + p_{i+1} \Big) \qquad \text{(lemme 5.8)} \end{aligned} \tag{5.62}$$

Puis en considérant $i = n-1$, on obtient :

$$q \triangleq v_{n-1} = \bigtriangledown_{\ell_n}\Big(\sum_{0\leq j\leq n-1} p_j \Big) \triangleq \bigtriangledown_{\ell_f - 1}(\mathit{mres}) \tag{5.63}$$

□

Lemme 5.16. *Pour tous $x \in \mathbb{R}$ et $\ell \in \mathbb{Z}$, on a :*

$$\bigcirc_\ell^{\textit{ties-up}}(\bigtriangledown_{\ell-1}(x)) = \bigcirc_\ell^{\textit{ties-up}}(x) \tag{5.64}$$

Preuve. Comme $\bigtriangledown_{\ell-1}(x)$ est dans $\mathbb{F}_{\ell-1}$, on a soit $\bigtriangledown_{\ell-1}(x) \in \mathbb{F}_\ell$, soit $\bigtriangledown_{\ell-1}(x)$ est un point milieu dans $\mathbb{F}_\ell$. Distinguons ces deux cas.

— Si $\bigtriangledown_{\ell-1}(x) \in \mathbb{F}_\ell$ alors $\bigcirc_\ell^{\textit{ties-up}}(\bigtriangledown_{\ell-1}(x)) = \bigtriangledown_{\ell-1}(x)$. De plus, on sait que $\bigtriangledown_{\ell-1}(x) \leq x < \bigtriangledown_{\ell-1}(x)+2^{\ell-1}$ donc l'arrondi au plus proche de x vers $\mathbb{F}_\ell$ est $\bigtriangledown_{\ell-1}(x)$ pour n'importe quelle règle de *tie-breaking*. En particulier $\bigtriangledown_{\ell-1}(x) = \bigcirc_\ell^{\textit{ties-up}}(x)$. On conclut par transitivité.

— Si $\bigtriangledown_{\ell-1}(x)$ est un point milieu dans $\mathbb{F}_\ell$, alors son arrondi au plus proche vers $\mathbb{F}_\ell$ est soit $\bigtriangledown_{\ell-1}(x) - 2^{\ell-1}$, soit $\bigtriangledown_{\ell-1}(x) + 2^{\ell-1}$ selon la règle de *tie-breaking* considérée. Ici on utilise *ties-up* qui arrondit tous les points milieu vers le haut donc $\bigcirc_\ell^{\textit{ties-up}}(\bigtriangledown_{\ell-1}(x)) = \bigtriangledown_{\ell-1}(x) + 2^{\ell-1}$. Par ailleurs, on a $\bigtriangledown_{\ell-1}(x) \leq x < \bigtriangledown_{\ell-1}(x) + 2^{\ell-1}$ c'est-à-dire $(\bigtriangledown_{\ell-1}(x) + 2^{\ell-1}) - 2^{\ell-1} \leq x < \bigtriangledown_{\ell-1}(x) + 2^{\ell-1}$ avec $\bigtriangledown_{\ell-1}(x) + 2^{\ell-1} \in \mathbb{F}_\ell$. On en déduit $\bigcirc_\ell^{\textit{ties-up}}(x) = \bigtriangledown_{\ell-1}(x) + 2^{\ell-1}$. On conclut à nouveau par transitivité. □

Preuve du théorème 5.14. En combinant les lemmes 5.15 et 5.16 on a bien :

$$\texttt{res} \triangleq \bigcirc_{\ell_f}^{\textit{ties-up}}(q) = \bigcirc_{\ell_f}^{\textit{ties-up}}(\bigtriangledown_{\ell_f-1}(\mathit{mres})) = \bigcirc_{\ell_f}^{\textit{ties-up}}(\mathit{mres}) \tag{5.65}$$

Puis la borne d'erreur sur un arrondi au plus proche est à nouveau donnée par le lemme 5.3. □

On remarque qu'on a eu besoin de l'hypothèse sur le tri des *lsbs* pour utiliser le lemme 5.8 dans la preuve du lemme 5.15. Cela inclut $\ell_{n-1} \leq \ell_n \triangleq \ell_f - 1$, et effectuer la dernière addition vers $\mathbb{F}_{\ell_f-1}$ permet ensuite d'appliquer le lemme 5.16. C'est pour cela que cet algorithme de somme s'applique à des termes ayant tous un petit *lsb* (strictement inférieur à ℓ_f).

On remarque aussi que les preuves s'appuient explicitement sur le fait que les arrondis sont vers le bas. On ne peut pas remplacer ces arrondis par n'importe quel arrondi valide dans cet algorithme, contrairement à ceux des sections 5.2.2 à 5.2.4.

Une autre différence avec l'accumulateur de Kulisch de la section 5.2.2 et sa variante de la section 5.2.3 est que l'algorithme est ici correctement arrondi pour un arrondi au plus proche pour la règle imposée *ties-up*, plutôt que pour n'importe quelle règle τ demandée. Ce n'est pas un problème dans la mesure où ce qui nous intéresse surtout est la borne d'erreur en sortie, qui est $|\texttt{res} - \mathit{mres}| \leq 2^{\ell-1} = \frac{1}{2}\mathit{ulp}$ pour tous les arrondis au plus proche indépendamment de leur règle de *tie-breaking*.

Formalisation. On prouve d'abord les lemmes auxiliaires indépendants dont on aura besoin : les lemmes 5.7, 5.8 et 5.16.

```
Notation roundFIXdown l := (roundFIX l Zfloor).
Notation roundFIXnearest_tiesup l :=
  (roundFIX l (Znearest (fun⇒ true))).

Lemma Rplus_rd_isFIX l x y :
  isFIX l y → (roundFIXdown l x) + y = roundFIXdown l (x + y).
Lemma rd_rd_lsble l l' x :
  (l ≤ l')%Z →
  roundFIXdown l' (roundFIXdown l x) = roundFIXdown l' x.
Lemma rn_rdpred l x :
  roundFIXnearest_tiesup l (roundFIXdown (l-1) x)
  = roundFIXnearest_tiesup l x.
```

Comme en section 5.2.3, on se donne la liste `ps` des termes à ajouter et la liste `ls` de leurs *lsbs* respectifs. On suppose que les ℓ_i sont croissants et tous strictement inférieurs au *lsb* final `lf`.

```
Variable lf : Z.
Variable ls : seq Z.
Variable ps : seq R.

Hypothesis are_lsb_ls_ps : are_lsb ls ps.
Hypothesis ls_srt : sorted Z.leb ls.
Hypothesis all_small : all (fun l ⇒ Z.ltb l lf) ls.

Notation n := (size ps).
```

Pour pouvoir utiliser la convention $\ell_n \triangleq \ell_f - 1$, on définit `getl` comme une application partielle de `nth` qui renvoie le n-ième élément d'une liste, à qui on donne ici la valeur $\ell_f - 1$ à renvoyer par défaut si l'indice demandé est trop grand.

```
Definition getl := nth (lf - 1)%Z ls.
```

On définit ensuite les $\boldsymbol{v}_i$ de l'équation (5.55), puis le résultat intermédiaire q, et enfin la sortie `res` appelée `summation_smalllsb`.

```
Fixpoint vi (i : nat) :=
match i with
  | O ⇒ roundFIX (getl 1) Zfloor (ps ' 0)
  | S j ⇒ addFIX (getl j.+2) Zfloor (vi j) (ps ' j.+1)
end.
Definition q := (vi n.-1).
Definition summation_smalllsb : R := roundFIXnearest_tiesup lf q.
```

On prouve facilement le lemme 5.15 par récurrence grâce aux lemmes 5.7 et 5.8, et enfin le théorème 5.14 à partir des lemmes 5.15 et 5.16.

```
Notation mres := (\sum_(i ← ps) i).
Lemma q_rd_mres : q = roundFIXdown (lf - 1) mres.
Theorem summation_smalllsb_correct :
  summation_smalllsb = roundFIXnearest_tiesup lf mres.
```

Variante : utilisation du *sticky bit* pour pouvoir choisir n'importe quelle règle de *tie-breaking*. On a vu que l'algorithme précédent impose la règle *ties-up*, ce qui n'est généralement pas un problème. Mais si on tient tout de même à obtenir un arrondi correct pour l'arrondi au plus proche basé sur n'importe quelle règle de *tie-breaking* τ désirée, on peut utiliser la variante suivante de l'algorithme.

On va s'appuyer sur la notion de *sticky bit*, aussi appelé *inexact flag* [90]. Il vaut 1 (ou *true*) si un calcul a nécessité un arrondi d'un nombre vers un format auquel il n'appartenait pas, ou 0 si tous les arrondis étaient triviaux donc le résultat est exact. Le *sticky bit* doit être fourni par les opérations de la norme flottante IEEE-754 [65]. Il se calcule facilement en matériel à l'aide d'un *ou* logique des bits de poids faible supprimés. Ici, on s'intéressera surtout au *sticky bit* d'additions binaires arrondies vers le bas. Dans les preuves, on utilisera que le *sticky bit* associé à l'opération $x \mathbin{\overset{+}{\nabla_\ell}} y$, noté *sticky*, vérifie :

$$sticky \iff x \mathbin{\overset{+}{\nabla_\ell}} y \neq x + y \tag{5.66}$$

On calcule les sommes partielles v_i comme précédemment (équation (5.55)), mais on note aussi $sticky_i$ le *sticky bit* correspondant au calcul de chaque v_i :

$$\begin{cases} sticky_0 \iff (\nabla_{\ell_1}(p_0) \neq p_0) \\ sticky_i \iff (v_{i-1} \mathbin{\overset{+}{\nabla_{\ell_{i+1}}}} p_i \neq v_{i-1} + p_i) \quad \text{pour } 1 \leq i \leq n-1 \end{cases} \tag{5.67}$$

On détermine le *ou* logique de ces *sticky bits*, qu'on note $sticky_q$:

$$sticky_q \triangleq \bigvee_{0 \leq i < n} sticky_i \tag{5.68}$$

La règle de *tie-breaking* qu'on utilise pour arrondir $q \triangleq v_{n-1}$ vers $\mathbb{F}_{\ell_f}$ dépend alors de $sticky_q$. Si $sticky_q$ est vrai, on utilise *ties-up* comme avant, sinon on utilise la règle τ exigée (pour laquelle l'algorithme devra être correctement arrondi). On note ici $\texttt{res}_\tau$ la sortie de l'algorithme correspondant à cette règle :

$$\texttt{res}_\tau \triangleq \begin{cases} \bigcirc_{\ell_f}^{ties\text{-}up}(q) & \text{si } sticky_q = 1 \\ \bigcirc_{\ell_f}^{\tau}(q) & \text{si } sticky_q = 0 \end{cases} \tag{5.69}$$

L'algorithme utilisant les *sticky bits* est récapitulé par l'algorithme 15 et la figure 5.11. Pour condenser l'écriture sur cette figure et dans les preuves plus loin, on notera :

$$(sticky_q \vee \tau) \triangleq (\text{si } sticky_q \text{ alors } ties\text{-}up \text{ sinon } \tau) \tag{5.70}$$

Cette notation vient de ce que la règle *ties-up* est représentée dans Flocq comme la fonction identiquement vraie, comme expliqué plus haut. L'intérêt est qu'on peut alors écrire :

$$\texttt{res}_\tau \triangleq \bigcirc_{\ell_f}^{sticky_q \vee \tau}(q) \tag{5.71}$$

On remarque d'ailleurs qu'on a toujours $sticky_q \vee ties\text{-}up = ties\text{-}up$. Lorsqu'on choisit $\tau = ties\text{-}up$, les *sticky bits* n'ont aucune influence sur la sortie de l'algorithme. Ainsi, la variante présentée ici peut être vue comme un cas général de l'algorithme précédent, dont la sortie `res` est égale à $\texttt{res}_{ties\text{-}up}$.

Algorithme 15 : Petits *lsbs* : *sticky bit* pour garantir la règle τ désirée

Donnés à l'avance : n, $\ell_0 \leq \cdots \leq \ell_{n-1} < \ell_f$, règle de *tie-breaking* τ
Entrées : $p_0, \ldots, p_{n-1}$
Sortie : $\texttt{res}_\tau$
Variables : v, *sticky*, *sticky_tmp*
$(v, sticky) \leftarrow \bigtriangledown_{\ell_1}(p_0)$
for $i \leftarrow 1$ **to** $n-1$ **do**
  $(v, sticky\_tmp) \leftarrow v \ \boxplus_{\ell_{i+1}} \ p_i$ `// par convention` $\ell_n \triangleq \ell_f - 1$
  $sticky \leftarrow sticky \vee sticky\_tmp$
end
if *sticky* **then**
  $\texttt{res}_\tau \leftarrow \bigcirc_{\ell_f}^{ties\text{-}up}(v)$
else
  $\texttt{res}_\tau \leftarrow \bigcirc_{\ell_f}^{\tau}(v)$
end

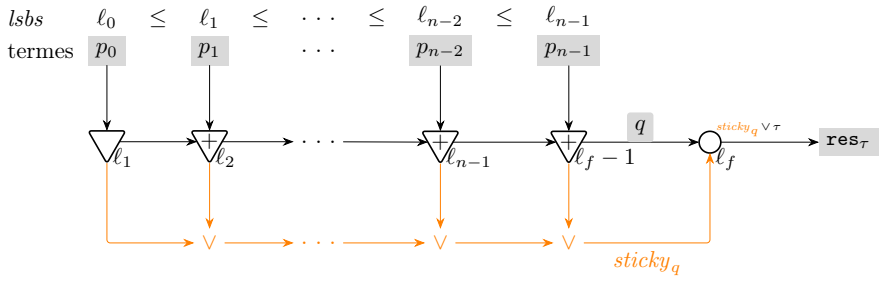


FIGURE 5.11 – Petits *lsbs* : *sticky bit* pour garantir la règle τ désirée

Théorème 5.17. *Cet algorithme est correctement arrondi au plus proche pour la règle de tie-breaking τ demandée :*

$$\texttt{res}_\tau = \bigcirc_{\ell_f}^{\tau}(mres) \tag{5.72}$$

À nouveau, on prouve d'abord deux lemmes auxiliaires, qui sont liés aux deux lemmes utilisés pour prouver le théorème 5.14. Le lemme 5.18 complète le lemme 5.15 lorsqu'on s'intéresse aux *sticky bits*. Puis le lemme 5.19 est un cas général du lemme 5.16 : il fait intervenir n'importe quelle règle τ au lieu de spécifiquement *ties-up*.

Lemme 5.18. *Le bit $sticky_q$ indique si l'arrondi $\bigtriangledown_{\ell_f-1}(mres)$ est inexact :*

$$sticky_q \iff \bigtriangledown_{\ell_f-1}(mres) \neq mres \tag{5.73}$$

Preuve. Montrons par récurrence sur $i \in \{0, 1, \ldots, n-1\}$ que :

$$\bigvee_{0 \leq j \leq i} sticky_j \iff (v_i \neq \sum_{0 \leq j \leq i} p_j) \tag{5.74}$$

— Pour $i = 0$, on a bien $sticky_0 \iff (\bigtriangledown_{\ell_1}(p_0) \neq p_0)$ par construction.

— Soit $0 \leq i \leq n-2$ vérifiant l'hypothèse de récurrence. On a d'une part :

$$v_{i+1} \triangleq v_i \mathbin{\overset{+}{\nabla}_{\ell_{i+2}}} p_{i+1} \leq v_i + p_{i+1} \tag{5.75}$$

D'autre part, on a vu dans la preuve du lemme 5.15 que $v_i = \nabla_{\ell_{i+1}}(\sum_{0\leq j\leq i} p_j)$. On a donc :

$$v_i + p_{i+1} = \nabla_{\ell_{i+1}}(\sum_{0\leq j\leq i} p_j) + p_{i+1} \leq \sum_{0\leq j\leq i+1} p_j \tag{5.76}$$

On en déduit que l'égalité $v_{i+1} = \sum_{0\leq j\leq i+1} p_j$ est vraie si et seulement si les deux inégalités ci-dessus sont en fait des égalités :

$$\begin{aligned} v_{i+1} = \sum_{0\leq j\leq i+1} p_j &\iff v_i \mathbin{\overset{+}{\nabla}_{\ell_{i+2}}} p_{i+1} = v_i + p_{i+1} \ \wedge\ v_i + p_{i+1} = \sum_{0\leq j\leq i+1} p_j \\ &\iff v_i \mathbin{\overset{+}{\nabla}_{\ell_{i+2}}} p_{i+1} = v_i + p_{i+1} \ \wedge\ v_i = \sum_{0\leq j\leq i} p_j \end{aligned} \tag{5.77}$$

Par construction de $sticky_{i+1}$ et par hypothèse de récurrence :

$$v_{i+1} = \sum_{0\leq j\leq i+1} p_j \iff \neg sticky_{i+1} \ \wedge\ \neg(\bigvee_{0\leq j\leq i} sticky_j) \tag{5.78}$$

c'est-à-dire :

$$v_{i+1} \neq \sum_{0\leq j\leq i+1} p_j \iff sticky_{i+1} \vee \bigvee_{0\leq j\leq i} sticky_j \iff \bigvee_{0\leq j\leq i+1} sticky_j \tag{5.79}$$

Puis en considérant $i = n-1$ et grâce au lemme 5.15 :

$$sticky_q \iff \bigvee_{0\leq j\leq n-1} sticky_j \iff q \neq \sum_{0\leq j\leq n-1} p_j \iff \nabla_{\ell_f - 1}(mres) \neq mres \tag{5.80}$$

□

Lemme 5.19. *Soit $x \in \mathbb{R}$, $\ell \in \mathbb{Z}$, et τ une règle de tie-breaking. Soit sticky le bit indiquant si l'arrondi $\nabla_{\ell-1}(x)$ est inexact :*

$$sticky \iff \nabla_{\ell-1}(x) \neq x \tag{5.81}$$

Alors :

$$\bigcirc_\ell^{sticky \vee \tau}(\nabla_{\ell-1}(x)) = \bigcirc_\ell^\tau(x) \tag{5.82}$$

Preuve. Ce lemme est une version plus générale du lemme 5.16 (qui correspondait au cas $\tau = ties\text{-}up$ vu que $sticky \vee ties\text{-}up = ties\text{-}up$). Leurs preuves se ressemblent donc beaucoup.

On a encore $\nabla_{\ell-1}(x) \in \mathbb{F}_{\ell-1}$ donc soit $\nabla_{\ell-1}(x) \in \mathbb{F}_\ell$, soit $\nabla_{\ell-1}(x)$ est un point milieu dans $\mathbb{F}_\ell$. Distinguons à nouveau ces deux cas.

— Le cas $\bigtriangledown_{\ell-1}(x) \in \mathbb{F}_\ell$ est identique au lemme 5.16 vu qu'il ne dépend pas de la règle de *tie-breaking* : on a $\bigtriangledown_{\ell-1}(x) \leq x < \bigtriangledown_{\ell-1}(x) + 2^{\ell-1}$ avec $\bigtriangledown_{\ell-1}(x) \in \mathbb{F}_\ell$ donc n'importe quel arrondi au plus proche de x vers $\mathbb{F}_\ell$ est $\bigtriangledown_{\ell-1}(x)$. En particulier $\bigcirc_\ell^\tau(x) = \bigtriangledown_{\ell-1}(x)$. Or $\bigtriangledown_{\ell-1}(x) \in \mathbb{F}_\ell$ signifie aussi que $\bigcirc_\ell^{sticky \vee \tau}(\bigtriangledown_{\ell-1}(x)) = \bigtriangledown_{\ell-1}(x) = \bigcirc_\ell^\tau(x)$.

— Si $\bigtriangledown_{\ell-1}(x)$ est un point milieu dans $\mathbb{F}_\ell$, alors son arrondi au plus proche vers $\mathbb{F}_\ell$ est soit $\bigtriangledown_{\ell-1}(x) - 2^{\ell-1}$, soit $\bigtriangledown_{\ell-1}(x) + 2^{\ell-1}$ selon la règle de *tie-breaking* considérée, ici $sticky \vee \tau$. Cela signifie aussi que $\bigtriangledown_{\ell-1}(x) + 2^{\ell-1} \in \mathbb{F}_\ell$. Distinguons deux cas selon la valeur de $sticky$.

  — Si $sticky = 1$ alors par sa définition $\bigtriangledown_{\ell-1}(x) \neq x$. On a donc $\bigtriangledown_{\ell-1}(x) < x < \bigtriangledown_{\ell-1}(x) + 2^{\ell-1}$ avec $\bigtriangledown_{\ell-1}(x) + 2^{\ell-1} \in \mathbb{F}_\ell$ donc $\bigcirc_\ell^\tau(x) = \bigtriangledown_{\ell-1}(x) + 2^{\ell-1}$. Par ailleurs, on a alors $sticky \vee \tau = ties\text{-}up$ donc cette règle arrondit le point milieu $\bigtriangledown_{\ell-1}(x)$ vers le haut d'où $\bigcirc_\ell^{sticky \vee \tau}(\bigtriangledown_{\ell-1}(x)) = \bigtriangledown_{\ell-1}(x) + 2^{\ell-1} = \bigcirc_\ell^\tau(x)$.

  — Si $sticky = 0$ alors on a $\bigtriangledown_{\ell-1}(x) = x$ par définition, et aussi $sticky \vee \tau = \tau$, d'où $\bigcirc_\ell^{sticky \vee \tau}(\bigtriangledown_{\ell-1}(x)) = \bigcirc_\ell^\tau(x)$.

□

Preuve du théorème 5.17. En combinant les lemmes 5.15, 5.18 et 5.19 on a bien :

$$\texttt{res}_\tau \triangleq \bigcirc_{\ell_f}^{sticky_q \vee \tau}(q) = \bigcirc_{\ell_f}^{sticky_q \vee \tau}(\bigtriangledown_{\ell_f-1}(mres)) = \bigcirc_{\ell_f}^{\tau}(mres) \tag{5.83}$$

□

Formalisation de la variante. On prouve d'abord le lemme 5.19 auxiliaire.

```
Lemma rn_rdpred_sticky l tau x :
  let sticky := (roundFIXdown (l-1) x != x) in
  roundFIX l (Znearest (fun z ⇒ sticky || tau z))
             (roundFIXdown (l-1) x)
  = roundFIX l (Znearest tau) x.
```

On garde le même contexte que pour la formalisation précédente de l'algorithme initial avec la règle *ties-up* et on réutilise les définitions de `vi` et `q`. On définit $sticky_q$ de l'équation (5.68) (attention, ici `stickyi i` n'est pas $sticky_i$ de l'équation (5.67) mais le *ou* des $sticky_j$ pour $j \leq i$). On se donne ensuite la règle désirée τ et on construit la sortie $\texttt{res}_\tau$.

```
Fixpoint stickyi (i : nat) :=
match i with
  | O ⇒ vi O != (ps ' O)
  | S j ⇒ stickyi j || ( vi j.+1 != (vi j + (ps ' j.+1))%R )
end.
Definition stickyq := (stickyi n.-1).

Variable tau : Z → bool.
Definition summation_smalllsb_sticky : R :=
  roundFIX lf (Znearest (fun z ⇒ stickyq || tau z)) q.
```

On prouve facilement le lemme 5.18 puis le théorème 5.17 en réutilisant le lemme 5.15 de l'algorithme sans *sticky bit* et grâce au lemme 5.19 auxiliaire.

```
Lemma stickyq_mres : stickyq = (roundFIXdown (lf-1) mres != mres).

Theorem summation_smalllsb_sticky_correct :
  summation_smalllsb_sticky = roundFIX lf (Znearest tau) mres.
```

5.3.2 Algorithme complet

On s'intéresse maintenant au cas général, où les termes peuvent avoir des *lsbs* de n'importe quelle taille (mais déjà triés). Le principe est d'appliquer l'algorithme de la section 5.3.1 aux termes de petit *lsb* (c'est-à-dire les p_i tels que $\ell_i < \ell_f$) et l'accumulation de Kulisch par *lsb* décroissant de la section 5.2.3 aux autres ($\ell_i \geq \ell_f$). On sépare ces deux groupes de termes en posant m tel que :

$$\ell_{m-1} < \ell_f \leq \ell_m \tag{5.84}$$

On s'intéresse en fait au résultat intermédiaire de chacun des deux algorithmes mentionnés, juste avant l'arrondi au plus proche final. Pour les termes $p_0, \ldots, p_{m-1}$ à petit *lsb*, on définit donc, similairement aux équations (5.55) et (5.56) :

$$\begin{aligned} v_0 &\triangleq \bigtriangledown_{\ell_1}(p_0) \\ v_i &\triangleq v_{i-1} \mathbin{\overset{+}{\bigtriangledown}}_{\ell_{i+1}} p_i \quad \text{pour } 1 \leq i \leq m-2 \\ q_{\text{petit}} &\triangleq v_{m-1} \triangleq v_{m-2} \mathbin{\overset{+}{\bigtriangledown}}_{\ell_f - 1} p_{m-1} \end{aligned} \tag{5.85}$$

Pour les termes à grand *lsb*, l'équation (5.30) devient :

$$q_{\text{grand}} \triangleq ((p_{n-1} \mathbin{\overset{+}{\bigtriangledown}}_{\ell_{n-2}} p_{n-2}) \mathbin{\overset{+}{\bigtriangledown}}_{\ell_{n-3}} p_{n-3}) \cdots \mathbin{\overset{+}{\bigtriangledown}}_{\ell_m} p_m \tag{5.86}$$

On additionne alors ces deux résultats intermédiaires en arrondissant au plus proche vers $\mathbb{F}_{\ell_f}$ avec la règle de *tie-breaking ties-up* comme en section 5.3.1 :

$$\texttt{res} \triangleq q_{\text{petit}} \oplus_{\ell_f}^{\textit{ties-up}} q_{\text{grand}} \tag{5.87}$$

Cet algorithme est récapitulé par l'algorithme 16 et illustré par la figure 5.12. Je ne donne pas ici de schéma de bits manipulés car il s'agit simplement de juxtaposer la figure 5.10 pour les petits *lsbs* et la figure 5.6 pour les grands *lsbs*.

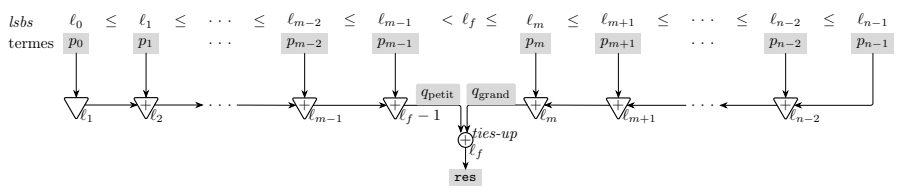


FIGURE 5.12 – Somme correctement arrondie au plus proche (règle *ties-up*)

Algorithme 16 : Somme correctement arrondie au plus proche (*ties-up*)

Donnés à l'avance : ℓ_f, n, $\ell_0 \leq \cdots \leq \ell_{n-1}$
Calculé à l'avance : m tel que $\ell_{m-1} < \ell_f \leq \ell_m$
Entrées : $p_0, \ldots, p_{n-1}$
Sortie : `res`
Variables : v_{petit}, v_{grand}
$v_{\text{petit}} \leftarrow \bigtriangledown_{\ell_1}(p_0)$
for $i \leftarrow 1$ **to** $m-2$ **do**
  $v_{\text{petit}} \leftarrow v_{\text{petit}} \ \overset{+}{\bigtriangledown}_{\ell_{i+1}} p_i$
end
$v_{\text{petit}} \leftarrow v_{\text{petit}} \ \overset{+}{\bigtriangledown}_{\ell_f-1} p_{m-1}$
$v_{\text{grand}} \leftarrow p_{n-1}$
for $i \leftarrow n-2$ **downto** m **do**
  $v_{\text{grand}} \leftarrow v_{\text{grand}} \ \overset{+}{\bigtriangledown}_{\ell_i} p_i$
end
`res` $\leftarrow v_{\text{petit}} \oplus_{\ell_f}^{\textit{ties-up}} v_{\text{grand}}$

Théorème 5.20. *L'algorithme complet est correctement arrondi pour l'arrondi au plus proche utilisant la règle ties-up :*

$$\texttt{res} = \bigcirc_{\ell_f}^{\textit{ties-up}}(\mathit{mres}) \tag{5.88}$$

En particulier, on a encore la borne d'erreur :

$$|\texttt{res} - \mathit{mres}| \leq 2^{\ell_f - 1} \tag{5.89}$$

Preuve. D'après les lemmes 5.11 et 5.15 on a :

$$q_{\text{grand}} = \sum_{m \leq i < n} p_i \qquad \text{et} \qquad q_{\text{petit}} = \bigtriangledown_{\ell_f-1}\Big(\sum_{0 \leq i < m} p_i\Big) \tag{5.90}$$

De plus par construction $q_{\text{grand}} \in \mathbb{F}_{\ell_m} \subset \mathbb{F}_{\ell_f - 1}$ (car $\ell_f \leq \ell_m$) donc :

$$\begin{aligned} q_{\text{petit}} + q_{\text{grand}} &= \bigtriangledown_{\ell_f-1}\Big(\sum_{0 \leq i < m} p_i\Big) + q_{\text{grand}} \\ &= \bigtriangledown_{\ell_f-1}\Big(\sum_{0 \leq i < m} p_i + q_{\text{grand}}\Big) \qquad \text{(lemme 5.7)} \\ &= \bigtriangledown_{\ell_f-1}\Big(\sum_{0 \leq i < n} p_i\Big) = \bigtriangledown_{\ell_f-1}(\mathit{mres}) \end{aligned} \tag{5.91}$$

Puis grâce au lemme 5.16 :

$$\texttt{res} \triangleq \bigcirc_{\ell_f}^{\textit{ties-up}}(q_{\text{petit}} + q_{\text{grand}}) = \bigcirc_{\ell_f}^{\textit{ties-up}}(\bigtriangledown_{\ell_f-1}(\mathit{mres})) = \bigcirc_{\ell_f}^{\textit{ties-up}}(\mathit{mres}) \tag{5.92}$$

Et comme d'habitude, on utilise le lemme 5.3 pour la borne d'erreur. □

Ainsi, on a bien un algorithme correctement arrondi pour un arrondi au plus proche, qui fonctionne quels que soient les *lsbs* des termes à sommer. L'arrondi

au plus proche en question utilise nécessairement la règle *ties-up*, mais comme expliqué en section 5.3.1, ce n'est pas un problème puisque la borne d'erreur est 2^{ℓ_f-1} pour n'importe quelle règle de *tie-breaking*. On peut tout de même choisir une règle τ particulière en utilisant des *sticky bits* comme en section 5.3.1. Ceci sera détaillé plus bas.

Comme pour l'accumulateur décroissant de la section 5.2.3, on perd ici la possibilité de complètement réordonner les additions pour facilement les paralléliser, qui était offerte par l'accumulateur de Kulisch basique en section 5.2.2 et l'algorithme à bits de gardes de la section 5.2.4. On peut tout de même paralléliser les calculs de q_{petit} et q_{grand}.

L'algorithme présenté ici a la même précision (borne d'erreur sur la sortie) et est plus efficace (moins de bits manipulés) que l'accumulateur de Kulisch de la section 5.2.2, mais aussi que sa variante par *lsb* décroissant de la section 5.2.3. En effet, il fait les mêmes additions que cette dernière pour les termes à grand *lsb*. Mais pour les termes à petit *lsb*, la succession de *lsbs* utilisés pour les additions binaires était $\ell_{m-2}, \ell_{m-3}, \ldots, \ell_1, \ell_0$ dans l'accumulation par *lsb* décroissant, tandis qu'ici il s'agit de $\ell_2, \ell_3, \ldots, \ell_{m-1}, \ell_f - 1$. Or des *lsbs* de calcul plus grands signifient moins de bits manipulés. De plus, sommer par *lsb* croissant plutôt que décroissant a un intérêt supplémentaire quand on prend en compte les *msbs* et largeurs de formats $\mathbb{F}_{\ell,w}$. Ceux-ci ont été ignorés dans les sections 5.1 à 5.3, et on ne rentre pas dans les détails ici. On remarque seulement que les formats des termes en entrée ont souvent des largeurs assez similaires, ce qui signifie que les *msbs* sont ordonnés à peu près comme les *lsbs*. Sommer par *lsb* croissant peut alors permettre d'utiliser des *msbs* plus petits dans les calculs intermédiaires, ce qui diminue à nouveau le nombre de bits manipulés.

L'algorithme présent peut être plus ou moins efficace que l'algorithme à bits de garde de la section 5.2.4, selon les *lsbs* des entrées. Dans le contexte des filtres numériques, il est souvent moins efficace. Cependant, il est plus précis puisqu'il fournit un arrondi correct au plus proche au lieu de seulement un arrondi fidèle.

Formalisation. On se donne le même contexte qu'en section 5.3.1 (sauf qu'on ne suppose évidemment pas que les *lsbs* de `ls` sont plus petits que `lf`).

```
Variable lf : Z.
Variable ls : seq Z.
Variable ps : seq R.

Hypothesis are_lsb_ls_ps : are_lsb ls ps.
Hypothesis ls_srt : sorted Z.leb ls.

Notation n := (size ps).
```

Grâce à la fonction `find` de MathComp, on définit `m` comme l'indice du premier élément de `ls` vérifiant le prédicat `Z.leb lf` (c'est-à-dire tel que ℓ_f est inférieur ou égal à cette élément). On définit alors les listes `ps_small` et `ls_small` correspondant aux `m` premiers éléments de `ps` et `ls`, et les listes `ps_large` et `ls_large` obtenues en enlevant les `m` premiers éléments de `ps` et `ls`.

```
Notation m := (find (Z.leb lf) ls).
Notation ps_small := (take m ps).
Notation ls_small := (take m ls).
Notation ps_large := (drop m ps).
Notation ls_large := (drop m ls).
```

On définit ensuite les sommes partielles q_{petit} et q_{grand}. On a placé la formalisation pour les petits *lsbs* dans un module nommé `Small_lsb` afin d'éviter de potentiels conflits de noms. On accède donc au `q` défini en section 5.3.1 en écrivant `Small_lsb.q`. D'autre part, on utilise `summation_decreasinglsb` de la section 5.2.3 (qui calculait seulement les additions exactes sans arrondi au plus proche final). On définit enfin `summation_nearest` en ajoutant ces deux résultats partiels à l'aide d'un arrondi au plus proche *ties-up*.

```
Notation q_small := (Small_lsb.q lf ls_small ps_small).
Notation q_large := (summation_decreasinglsb ls_large ps_large).
Definition summation_nearest :=
  addFIX lf (Znearest (fun⇒ true)) q_small q_large.
```

La preuve de l'algorithme suit la preuve du théorème 5.20. Elle est facile en réutilisant les lemmes de la section 5.3.1. Notons qu'on prouve d'abord la valeur de $q_{\text{petit}} + q_{\text{grand}}$ dans un lemme séparé car on en aura à nouveau besoin plus loin.

```
Lemma add_q_mres : q_small + q_large = roundFIXdown (lf - 1) mres.

Theorem summation_nearest_correct :
  summation_nearest = roundFIX lf (Znearest (fun⇒ true)) mres.
```

Variante : choisir la règle τ grâce aux *sticky bits*. Comme en section 5.3.1, si on souhaite tout de même obtenir un arrondi au plus proche correct pour une règle τ particulière, c'est possible en s'appuyant sur des *sticky bits*. En effet, il suffit d'appliquer aux termes de petits *lsbs* la variante utilisant des *sticky bits* de la section 5.3.1. On définit alors, similairement aux équations (5.67), (5.68) et (5.71) :

$$\begin{aligned}
sticky_0 &\iff (\bigtriangledown_{\ell_1}(p_0) \neq p_0) \\
sticky_i &\iff (v_{i-1} \mathbin{\overset{\top}{\bigtriangledown}}_{\ell_{i+1}} p_i \neq v_{i-1} + p_i) \qquad \text{pour } 1 \leq i \leq m-2 \\
sticky_{m-1} &\iff (v_{m-2} \mathbin{\overset{\top}{\bigtriangledown}}_{\ell_f - 1} p_{m-1} \neq v_{m-2} + p_{m-1}) \\
sticky_q &\triangleq \bigvee_{0 \leq i < m} sticky_i \\
\texttt{res}_\tau &\triangleq q_{\text{petit}} \oplus_{\ell_f}^{sticky_q \vee \tau} q_{\text{grand}}
\end{aligned} \tag{5.93}$$

Je ne récris pas l'algorithme complet, qui se déduit facilement de l'algorithme 16 précédent et de l'algorithme 15 pour la gestion des *sticky bits* et des règles de *tie-breaking*. Je donne néanmoins un graphe des opérations en figure 5.13, bien que lui aussi s'obtienne facilement à partir des graphes précédents.

Théorème 5.21. *Cette variante est correctement arrondie au plus proche pour la règle de tie-breaking τ demandée :*

$$\texttt{res}_\tau = \bigcirc_{\ell_f}^{\tau}(mres) \tag{5.94}$$

Preuve. Dans la preuve du théorème 5.20, on a vu que :

$$q_{\text{petit}} + q_{\text{grand}} = \bigtriangledown_{\ell_f - 1}(mres) \tag{5.95}$$

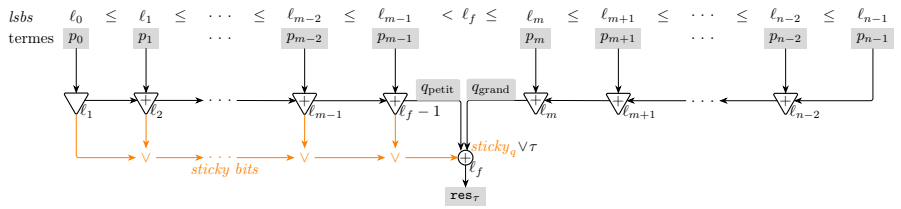


FIGURE 5.13 – Somme correctement arrondie au plus proche pour la règle τ

mais aussi que :

$$\bigtriangledown_{\ell_f-1}(mres) = \bigtriangledown_{\ell_f-1}(\sum_{0\le i<m} p_i) + q_{\text{grand}} = \bigtriangledown_{\ell_f-1}(\sum_{0\le i<m} p_i) + \sum_{m\le i<n} p_i \quad (5.96)$$

On a donc (en utilisant le lemme 5.18 pour la dernière équivalence) :

$$\begin{aligned} \bigtriangledown_{\ell_f-1}(mres) \neq mres &\iff \bigtriangledown_{\ell_f-1}(\sum_{0\le i<m} p_i) + \sum_{m\le i<n} p_i \neq \sum_{0\le i<n} p_i \\ &\iff \bigtriangledown_{\ell_f-1}(\sum_{0\le i<m} p_i) \neq \sum_{0\le i<m} p_i \\ &\iff sticky_q \end{aligned} \quad (5.97)$$

Enfin, grâce au lemme 5.19 :

$$\texttt{res}_\tau \triangleq \bigcirc_{\ell_f}^{sticky_q \vee \tau}(q_{\text{petit}} + q_{\text{grand}}) = \bigcirc_{\ell_f}^{sticky_q \vee \tau}(\bigtriangledown_{\ell_f-1}(mres)) = \bigcirc_{\ell_f}^{\tau}(mres) \quad (5.98)$$

□

Formalisation de la variante. On garde le même contexte que pour la version qui imposait la règle *ties-up*. On récupère le *sticky bit* associé aux calculs pour les termes de petit *lsb*. On se donne aussi la règle de *tie-breaking* désirée τ et on construit la sortie $\texttt{res}_\tau$.

```
Notation stickyq_small := (Small_lsb.stickyq lf ls_small ps_small).

Variable tau : Z → bool.

Definition summation_nearest_sticky : R :=
  addFIX lf (Znearest (fun z ⇒ stickyq_small || tau z))
         q_small q_large.
```

On prouve que le *sticky bit* obtenu vérifie bien l'équation (5.97). Puis on prouve très facilement que l'algorithme est correctement arrondi en réutilisant le lemme `add_q_mres` (prouvé précédemment lorsqu'on utilisait la règle *ties-up*), ainsi que le lemme 5.19 (`rn_rdpred_sticky`) de la section 5.3.1.

```
Lemma stickyq_small_mres :
  stickyq_small = (roundFIXdown (lf-1) mres != mres).

Theorem summation_nearest_sticky_correct :
  summation_nearest_sticky = roundFIX lf (Znearest tau) mres.
```

5.3.3 Un autre algorithme basée sur des arrondis impairs

Cette section présente un autre algorithme original de somme de produits (ou plutôt de somme, qu'on peut appliquer à des produits calculés exactement), correctement arrondi au plus proche pour n'importe quelle règle de *tie-breaking* τ souhaitée. Il a été développé avant, et est moins efficace que celui de la section section 5.3.2. Je ne le décris donc que brièvement ici. Une présentation plus détaillée a été publiée [17], incluant notamment une preuve de correction.

Cet algorithme est basé sur l'utilisation de l'*arrondi impair*, noté $\square$. Si un nombre n'est pas dans le format considéré, son arrondi impair est le plus proche nombre du format ayant une mantisse impaire :

$$\forall \ell \in \mathbb{Z}, \forall x \in \mathbb{R}, \quad \square_\ell(x) \triangleq \begin{cases} x & \text{si } x \in \mathbb{F}_\ell \\ \bigtriangledown_\ell(x) & \text{si la mantisse } \bigtriangledown_\ell(x)\, 2^{-\ell} \text{ est impaire} \\ \bigtriangleup_\ell(x) & \text{sinon} \end{cases} \tag{5.99}$$

L'intérêt de cet arrondi est qu'il se comporte bien lors d'arrondis successifs vers des formats de différents *lsbs*. Il ne fait pas partie du standard IEEE-1666 de SystemC pour la virgule fixe [62], mais il n'est pas difficile à implémenter. En matériel, il s'agit simplement d'arrondir vers le bas puis d'effectuer un *ou* logique entre le *sticky bit* et le dernier bit de la mantisse. Une implémentation logicielle efficace est également disponible [19].

On aura besoin que les ℓ_i soient triés *strictement* : $\ell_i < \ell_{i+1}$. Pour passer du tri large ($\ell_i \leq \ell_{i+1}$) supposé au début de la section 5.3 à un tri strict, on effectue un pré-traitement : on groupe les produits ayant le même *lsb* à l'aide d'une addition à ce *lsb*, qu'on sait être exacte. Ce pré-traitement est décrit par l'algorithme 17. Il modifie les p_i et ℓ_i, et décroît potentiellement le nombre total n de terme. Cependant, il garde la somme exacte $mres \triangleq \sum_{0 \leq i < n} p_i$ inchangée, et conserve la propriété que ℓ_i est toujours le *lsb* de p_i. Et à l'issue de ce pré-traitement, on a bien :

$$\ell_0 < \cdots < \ell_{n-1} \tag{5.100}$$

Algorithme 17 : Pré-traitement pour obtenir des *lsbs* strictement triés

```
while on trouve un i tel que ℓ_i = ℓ_{i+1} do
    p_i ← p_i ▽+_{ℓ_i} p_{i+1}
    for j ← i + 1 to n − 2 do
        p_j ← p_{j+1}
        ℓ_j ← ℓ_{j+1}
    end
    n ← n − 1
end
```

Ensuite, comme en section 5.3.2, on sépare les termes en deux groupes selon leur *lsb*. Mais cette fois-ci, la séparation se situe entre $\ell_f - 2$ et $\ell_f - 1$, au lieu d'entre $\ell_f - 1$ et ℓ_f. En effet, on pose m tel que :

$$\ell_{m-1} \leq \ell_f - 2 < \ell_m \tag{5.101}$$

Pour les termes à grand *lsb*, on utilise à nouveau l'accumulateur par *lsb* décroissant de la section 5.2.3. Le traitement des termes à petit *lsb* ressemble à

la section 5.3.1 ; cependant, l'ajout de chaque p_i au résultat précédent se fait avec un arrondi impair plutôt que vers le bas, et au *lsb* $\ell_{i+1}-1$ plutôt que ℓ_{i+1}. Et pour le dernier produit p_{m-1}, on utilise le *lsb* ℓ_f-2. On utilise aussi le *lsb* ℓ_f-2 pour ajouter les deux sommes obtenues pour les petits *lsbs* et les grands *lsbs* respectivement, avant d'arrondir au plus proche vers $\mathbb{F}_{\ell_f}$ avec la règle τ demandée. Tout cela est décrit par l'algorithme 18 et illustré par la figure 5.14.

Algorithme 18 : Algorithme utilisant des arrondis impairs

Donnés à l'avance : ℓ_f, n, $\ell_0 < \cdots < \ell_{n-1}$, τ
Calculé à l'avance : m tel que $\ell_{m-1} \le \ell_f - 2 < \ell_m$
Entrées : $p_0, \ldots, p_{n-1}$
Sortie : `res`
Variables : v_{petit}, v_{grand}
$v_{\text{petit}} \leftarrow \square_{\ell_1-1}(p_0)$
for $i \leftarrow 1$ **to** $m-2$ **do**
  $v_{\text{petit}} \leftarrow v_{\text{petit}} \boxplus_{\ell_{i+1}-1} p_i$
end
$v_{\text{petit}} \leftarrow v_{\text{petit}} \boxplus_{\ell_f-2} p_{m-1}$
$v_{\text{grand}} \leftarrow p_{n-1}$
for $i \leftarrow n-2$ **downto** m **do**
  $v_{\text{grand}} \leftarrow v_{\text{grand}} \bigtriangledown_{\ell_i} p_i$
end
`res` $\leftarrow \bigcirc^{\tau}_{\ell_f}(v_{\text{petit}} \bigtriangledown_{\ell_f-2} v_{\text{grand}})$

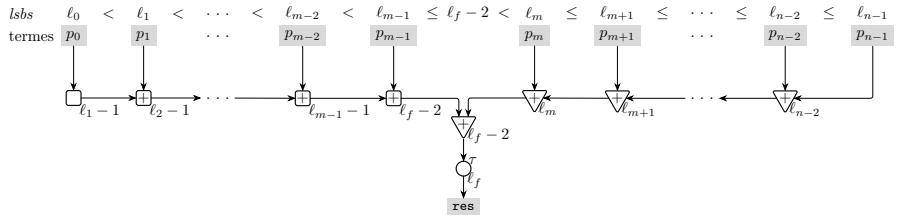


FIGURE 5.14 – Algorithme utilisant des arrondis impairs

Théorème 5.22. *La sortie de l'algorithme utilisant des arrondis impairs est l'arrondi au plus proche vers $\mathbb{F}_{\ell_f}$ de la somme exacte, pour la règle de tie-breaking τ désirée :*

$$\texttt{res} = \bigcirc^{\tau}_{\ell_f}(\mathit{mres}) \tag{5.102}$$

Ce théorème est prouvé sur papier [17], ainsi que formellement en Coq (théorème `summation_odd_correct`). On a donc encore un algorithme correctement arrondi au plus proche pour la règle τ demandée. Cependant, celui de la section 5.3.2 est légèrement plus efficace (surtout quand accepte un arrondi correct au plus proche selon *ties-up* plutôt qu'une règle τ au choix). En effet, l'arrondi vers le bas est un peu plus rapide à calculer que l'arrondi impair en virgule fixe en complément à deux (surtout quand on n'a pas besoin de connaître le *sticky bit*), et les *lsbs* des additions de termes à petit *lsb* sont plus grands de 1 (ℓ_{i+1} en section 5.3.2 au lieu de $\ell_{i+1}-1$ ici pour l'ajout de p_i).

5.4 Prise en compte des *overflows* modulaires

Cette section s'intéresse à la prise en compte des *overflows*, qui avaient été ignorés dans les sections 5.1 à 5.3. On considère surtout l'*overflow* modulaire : on verra qu'il est facile à implémenter en complément à deux (il suffit d'ignorer les bits de poids trop fort) et particulièrement adapté pour calculer des sommes (les sommes partielles peuvent déborder sans changer la qualité du résultat final tant que celui-ci ne déborde pas). La section 5.4.1 détaille les propriétés de l'*overflow* modulaire et des arrondis et opérations associés. La section 5.4.2 présente ma formalisation des *overflows* en virgule fixe, et plus particulièrement de l'*overflow* modulaire. Cette formalisation est ensuite utilisée en section 5.4.3 pour prouver le comportement de l'algorithme fidèle à bits de garde de la section 5.2.4 en tenant compte des *overflows*.

5.4.1 Virgule fixe en complément à deux et overflow modulaire

Cette section présente l'*overflow* modulaire en virgule fixe et ses conséquences sur les arrondis et opérations.

Représentation des entiers en complément à deux. J'ai affirmé en section 1.3.2 que l'*overflow* modulaire est naturel en virgule fixe lorsque la mantisse est représentée en complément à deux. Pour expliquer cela, intéressons-nous plus en détails à la représentation en complément à deux des nombres entiers. Rappelons que dans cette représentation, l'entier m associé à w bits $m_{w-1}, \ldots, m_0$ est :

$$m = -m_{w-1}2^{w-1} + \sum_{i=0}^{w-2} m_i 2^i \qquad \text{(rappel de (1.7))}$$

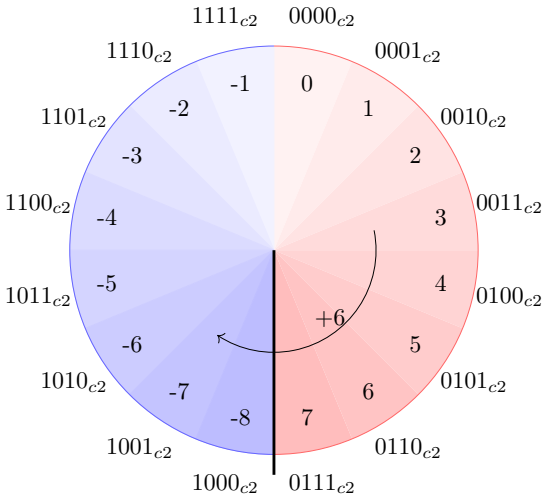


FIGURE 5.15 – Représentation des entiers en complément à deux sur $w = 4$ bits

La figure 5.15 illustre les nombres représentés en complément à deux sur $w = 4$ bits. Il s'agit des nombres entre $-2^{w-1} = -8$ et $2^{w-1} - 1 = 7$ inclus. La

seule différence avec la représentation binaire non signée habituelle des entiers positifs est le poids négatif devant le bit m_{w-1} de poids le plus fort. Cela signifie que lorsque $m_{w-1} = 0$, il n'y a aucune différence, comme on peut le voir pour les nombres de 0 à 7. Mais quand $m_{w-1} = 1$, on obtient un nombre négatif, dont la différence avec le nombre en binaire usuel non signé est 2^w. C'est pour cela que cette représentation est adaptée à des opérations modulaires, modulo 2^w. Par exemple, en représentation binaire positive usuelle, $3 + 6 = 9$ se traduit par $11_2 + 110_2 = 1001_2$. En complément à deux, comme 3 et 6 sont entre 0 et 7, leurs représentations sur 4 bits sont encore 0011_{c2} et 0110_{c2}. Leur somme est calculée de la même manière bits à bits, mais le résultat est interprété en complément à deux : on obtient $1001_{c2} = -7 \equiv 9$ [16]. On a bien observé un *overflow* modulaire car 9 n'est pas représentable en complément à deux sur 4 bits.

L'intervalle $[-2^{w-1}, 2^{w-1}[$ sera appelé la *période principale*. L'*overflow* modulaire consiste simplement à envoyer un nombre vers son représentant modulo 2^w dans cette période principale.

Pour diminuer la largeur w, il suffit de supprimer les bits sur la gauche jusqu'à avoir le nombre de bits voulu, ce qui peut provoquer un *overflow* modulaire. Par exemple, si on veut représenter le nombre à 6 bits $101010_{c2} = -22$ sur seulement 4 bits, on enlève les deux premiers bits et on obtient $1010_{c2} = -6 \equiv -22$ [16]. Lorsqu'on effectue une opération dont le résultat doit être sur w bits, on ne calcule pas du tout les bits à gauche de celui d'indice $w - 1$.

Si on souhaite au contraire augmenter la largeur w, on remplit les nouveaux bits à gauche avec des copies du bit de signe. Si le nombre est positif, on le complète donc par des zéros comme dans la représentation binaire positive usuelle : on a $0011_{c2} = 3$ sur 4 bits, et on a encore $000011_{c2} = 3$ sur 6 bits. En revanche, quand le nombre est négatif, le compléter par des zéros modifierait sa valeur : $1010_{c2} = -6$ mais $001010_{c2} = 10$. On remarque d'ailleurs que $10 \equiv -6$ [16], mais on n'a pas de raison de faire apparaître un modulo alors que -6 est représentable sur 6 bits. En complétant par des uns au lieu de zéros, on obtient bien $111010_{c2} = -6$.

***Overflow* modulaire en virgule fixe.** Dans tout le reste de la section, on se donne un *lsb* ℓ et une largeur $w \geq 2$ (pour éviter des cas particuliers dégénérés). Rappelons qu'un nombre du format $\mathbb{F}_{\ell,w}$ est représenté par une mantisse m en complément à deux sur w bits, qu'il faut ensuite multiplier par le facteur d'échelle implicite 2^ℓ :

$$\mathbb{F}_{\ell,w} \triangleq \{m \times 2^\ell \mid m \in \mathbb{Z} \wedge -2^{w-1} \leq m \leq 2^{w-1} - 1\} \qquad \text{(rappel de (1.8))}$$

Pour passer du format $\mathbb{F}_\ell$ ignorant les *overflows* au format $\mathbb{F}_{\ell,w}$, il suffit d'appliquer un modulo 2^w à la mantisse entière comme ci-dessus. En effet, par définition, un nombre $x \in \mathbb{F}_\ell$ s'écrit sous la forme $x = m2^\ell$ avec $m \in \mathbb{Z}$. On envoie donc m vers son représentant m' modulo 2^w dans $[-2^{w-1}, 2^{w-1}[$, et on obtient bien $x' = m'2^\ell \in \mathbb{F}_{\ell,w}$. Cela revient à choisir directement x' comme le représentant de x modulo $2^{w+\ell}$ dans $[-2^{w+\ell-1}, 2^{w+\ell-1}[$.

L'intervalle $[-2^{w+\ell-1}, 2^{w+\ell-1}[$ sera donc appelé la *période principale du format* $\mathbb{F}_{\ell,w}$. De plus, on notera $cmod_p : \mathbb{R} \to \mathbb{R}$ la fonction qui à x associe son représentant modulo p dans l'intervalle $[-\frac{p}{2}, \frac{p}{2}[$ (le c vient de *central* car l'intervalle d'arrivée est centré en 0). On utilisera donc $p = 2^{w+\ell}$ dans le cadre du

format $\mathbb{F}_{\ell,w}$. Quand on sera clairement en train de considérer un tel format, on omettra souvent l'indice : on écrira simplement *cmod* pour $cmod_{2^{w+\ell}}$.

***Overflow* et arrondi.** Lorsqu'on doit représenter un réel x quelconque dans $\mathbb{F}_{\ell,w}$, par exemple un coefficient de filtre ou le résultat d'un calcul, on doit à la fois effectuer un arrondi (c'est-à-dire appliquer une fonction $\bigcirc_\ell : \mathbb{R} \to \mathbb{F}_\ell$ au cas où x ne serait pas un multiple entier de 2^ℓ) et utiliser la fonction *cmod* définie ci-dessus (en cas d'*overflow*). La question qui se pose alors est dans quel ordre appliquer ces deux fonctions.

On a vu que les calculs en complément à deux sont directement effectués sans jamais prendre en considération les bits à gauche du *msb*. L'*overflow* modulaire est donc appliqué en permanence au cours du calcul. Au contraire, on doit parfois s'intéresser à des bits à droite du *lsb* pour pouvoir arrondir correctement une opération, donc on voit plutôt l'arrondi comme une étape ajoutée à la fin. Il est donc plus naturel d'appliquer le modulo avant l'arrondi : on considère $\bigcirc_\ell(cmod(x))$. Cependant, on peut avoir un problème : *cmod* peut renvoyer un nombre dans l'intervalle $]2^{w+\ell-1} - 2^\ell, 2^{w+\ell-1}[$, qui peut ensuite être arrondi vers $2^{w+\ell-1}$ (si par exemple $\bigcirc_\ell$ est un arrondi vers le haut) qui n'est pas dans la période principale. On a donc besoin d'appliquer à nouveau *cmod* après l'arrondi : $cmod_{2^{w+\ell}}(\bigcirc_\ell(cmod_{2^{w+\ell}}(x))) \in \mathbb{F}_{\ell,w}$. Ceci est illustré par la figure 5.16. On obtient alors la définition suivante.

Définition 5.23 (Arrondi vers $\mathbb{F}_{\ell,w}$). *La fonction d'arrondi $\bigcirc_{\ell,w} : \mathbb{R} \to \mathbb{F}_{\ell,w}$ correspondant à un arrondi sans overflow $\bigcirc_\ell : \mathbb{R} \to \mathbb{F}_\ell$ est définie par :*

$$\forall x \in \mathbb{R}, \quad \bigcirc_{\ell,w}(x) \triangleq cmod_{2^{w+\ell}}(\bigcirc_\ell(cmod_{2^{w+\ell}}(x))) \tag{5.103}$$

On utilise naturellement les notations $\bigtriangledown_{\ell,w}$ ou $\bigtriangleup_{\ell,w}$ ou $\bigcirc^\tau_{\ell,w}$ pour les fonctions d'arrondi vers $\mathbb{F}_{\ell,w}$ correspondant à $\bigtriangledown_\ell$ ou $\bigtriangleup_\ell$ ou $\bigcirc^\tau_\ell$.

Arrondi quasi périodique. J'ai expliqué pourquoi la fonction *cmod* apparaît deux fois dans l'équation (5.103), mais ce n'est pas très pratique. Je vais donc montrer qu'on peut enlever l'application initiale de *cmod* à x sans modifier le résultat si $\bigcirc_\ell$ vérifie la propriété suivante.

Définition 5.24 (Fonction quasi périodique). *Une fonction $f : \mathbb{R} \to \mathbb{R}$ est* quasi périodique pour l'addition*, de période $p > 0$, si :*

$$\forall x \in \mathbb{R}, \quad f(x + p) = f(x) + p \tag{5.104}$$

De manière générale, le terme *quasi périodique* peut être utilisé pour n'importe quelle propriété qui ressemble à la périodicité. Notons que cela n'a rien à voir avec la quasi périodicité d'un signal en traitement du signal. Dans ce manuscrit, *quasi périodique* désignera toujours la propriété définie ci-dessus.

On prouve maintenant un premier lemme auxiliaire, qui sera souvent appliqué aux fonctions d'arrondi quasi périodiques. Puis on l'utilise pour montrer que comme affirmé précédemment, un arrondi quasi périodique permet d'omettre un appel à *cmod* dans la définition 5.23. Enfin, le lemme 5.27 donne une condition suffisante pour déterminer l'arrondi vers $\mathbb{F}_{\ell,w}$ d'un nombre.

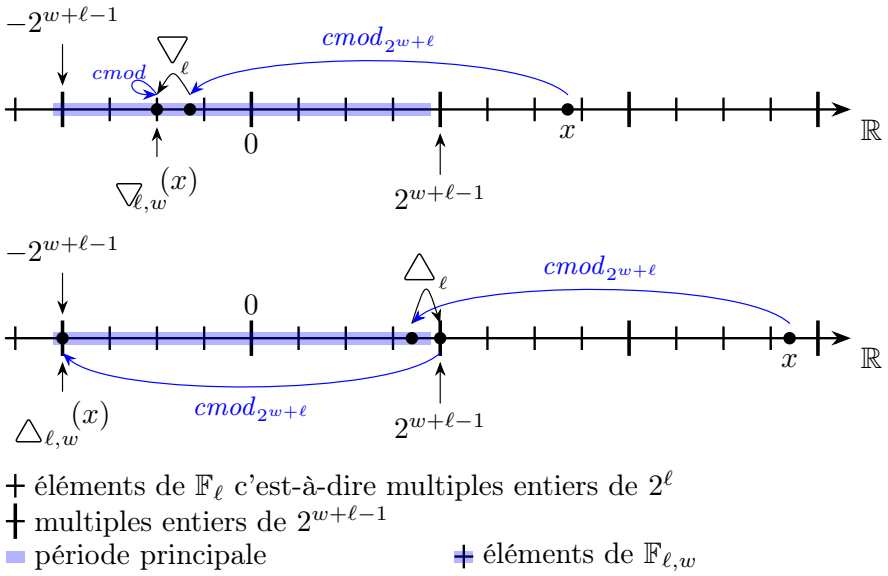


FIGURE 5.16 – Arrondir vers $\mathbb{F}_{\ell,w}$: calculs respectifs de $\nabla_{\ell,w}(x)$ et de $\triangle_{\ell,w}(x)$

Lemme 5.25. *Si f est quasi périodique de période p, alors :*

$$\forall x, y \in \mathbb{R}, \quad x \equiv y \ [p] \quad \Longrightarrow \quad f(x) \equiv f(y) \ [p] \tag{5.105}$$

Preuve. On montre facilement par récurrence sur $n \in \mathbb{N}$ que $\forall x \in \mathbb{R}, f(x + np) = f(x) + np$. Puis en remarquant que $f(x - np) = f(x) - np$, on obtient :

$$\forall k \in \mathbb{Z}, \forall x \in \mathbb{R}, \quad f(x + kp) = f(x) + kp \tag{5.106}$$

On conclut en rappelant que $x \equiv y \ [p]$ signifie $\exists k \in \mathbb{Z}, x = y + kp$ par définition. □

Lemme 5.26. *Si $\bigcirc_\ell$ est quasi périodique de période $2^{w+\ell}$, alors :*

$$\forall x \in \mathbb{R}, \quad \bigcirc_{\ell,w}(x) = cmod_{2^{w+\ell}}(\bigcirc_\ell(x)) \tag{5.107}$$

Preuve. On applique le lemme 5.25 à $\bigcirc_\ell$ qui est quasi périodique de période $2^{w+\ell}$, en remarquant que $cmod(x) \equiv x \ [2^{w+\ell}]$ par définition de $cmod$:

$$\forall x \in \mathbb{R}, \quad \bigcirc_\ell(cmod(x)) \equiv \bigcirc_\ell(x) \ [2^{w+\ell}] \tag{5.108}$$

Ces deux nombres ont donc le même représentant modulo $2^{w+\ell}$ dans la période principale :

$$\forall x \in \mathbb{R}, \quad \bigcirc_{\ell,w}(x) \triangleq cmod(\bigcirc_\ell(cmod(x))) = cmod(\bigcirc_\ell(x)) \tag{5.109}$$

□

Lemme 5.27. *Si $\bigcirc_\ell$ est quasi périodique de période $2^{w+\ell}$, alors :*

$$\forall x, y \in \mathbb{R}, \quad y \in \mathbb{F}_{\ell,w} \ \wedge \ y \equiv \bigcirc_\ell(x) \ [2^{w+\ell}] \quad \Longrightarrow \quad y = \bigcirc_{\ell,w}(x) \tag{5.110}$$

Preuve. Si $y \in \mathbb{F}_{\ell,w}$ alors en particulier y est dans la période principale de $\mathbb{F}_{\ell,w}$. Si de plus y est congru à $\bigcirc_\ell(x)$ modulo $2^{w+\ell}$, alors $y = cmod(\bigcirc_\ell(x))$ par définition de *cmod*. On conclut grâce au lemme 5.26. □

On s'intéressera donc désormais seulement à des arrondis quasi périodiques. Parmi les arrondis usuels, l'arrondi vers le bas $\bigtriangledown_\ell$ et l'arrondi vers le haut $\bigtriangleup_\ell$ sont quasi périodiques (de période $2^{w+\ell}$ pour n'importe quel $w \geq 2$). En revanche, l'arrondi vers 0 ne l'est pas [4], ce qui explique qu'il soit rarement utilisé en virgule fixe. Un arrondi au plus proche est quasi périodique si et seulement si sa règle de *tie-breaking* τ est périodique. C'est bien le cas de la règle *ties-to-even*, donc l'arrondi au plus proche pair $\bigcirc_\ell^{even}$ est quasi périodique. La règle *ties-up* utilisée dans la section 5.3 est aussi périodique, mais la règle *away-from-zero* ne l'est pas.

Arrondi vers le bas et règle *ties-up* en matériel. Notons qu'en plus d'être quasi périodique, l'arrondi vers le bas est particulièrement facile à implémenter en complément à deux. En effet, il revient simplement à supprimer les bits de poids plus faible que le *lsb* (aussi bien pour les nombres positifs que négatifs, grâce à la nature cyclique de cette représentation). En cela, il est similaire à l'arrondi vers 0 pour les nombres flottants (ou de manière plus générale pour les représentations signe / valeur absolue). De même, la règle de *tie-breaking ties-up* utilisée en section 5.3 revient toujours à incrémenter le dernier bit. Elle est donc le pendant pour la virgule fixe en complément à deux de la règle *away-from-zero* pour la virgule flottante.

En section 5.4.3, pour décrire le comportement de l'algorithme à bits de garde de la section 5.2.4 lorsqu'il risque d'y avoir des *overflows*, on utilisera la notion suivante.

Définition 5.28 (Arrondi fidèle modulaire). *Le réel y est un* arrondi fidèle modulaire *du réel x dans le format $\mathbb{F}_{\ell,w}$ si :*

$$y = \bigtriangledown_{\ell,w}(x) \quad \vee \quad y = \bigtriangleup_{\ell,w}(x) \tag{5.111}$$

On retrouve la même définition que pour un arrondi fidèle (section 5.1.1), mais avec des arrondis vers $\mathbb{F}_{\ell,w}$ au lieu de $\mathbb{F}_\ell$. Ceci est illustré par la figure 5.17. Notons que comme les arrondis vers le bas et vers le haut sont tous deux quasi périodiques, le lemme 5.26 nous permet d'omettre l'application initiale de *cmod* dans la définition 5.23.

On utilisera la condition suffisante ci-dessous pour montrer qu'on a un arrondi fidèle modulaire.

Lemme 5.29. *Soit $x \in \mathbb{R}$ et $y \in \mathbb{F}_{\ell,w}$. Si :*

$$\exists y' \in \mathbb{R}, \quad y \equiv y' \ [2^{w+\ell}] \ \wedge \ |y' - x| < 2^\ell \tag{5.112}$$

alors y est un arrondi fidèle modulaire de x dans $\mathbb{F}_{\ell,w}$.

4. Le problème est qu'un nombre positif et un nombre négatif peuvent être congrus modulo la période alors qu'ils s'arrondissent dans des sens différents. Par exemple si $\ell = 0$ et $w = 3$ donc la période est $2^{3+0} = 8$, on a $5{,}7 = -2{,}3 + 8$ mais leurs arrondis respectifs vers 0 dans $\mathbb{F}_0 = \mathbb{Z}$ sont $5 \neq -2 + 8$.

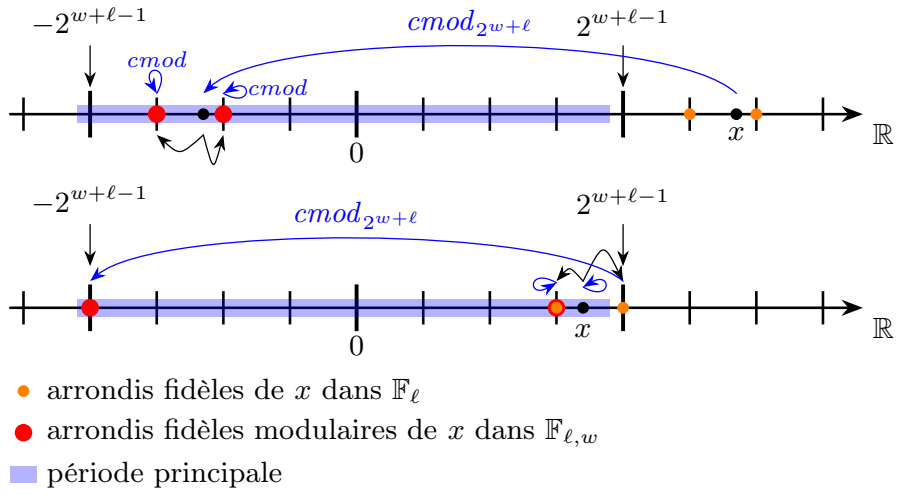


FIGURE 5.17 – Arrondis fidèles modulaires de x

Preuve. On rappelle que $w \geq 0$ donc 2^ℓ divise $2^{w+\ell}$ donc $y \equiv y' \ [2^\ell]$. Or $y \in \mathbb{F}_{\ell,w}$ donc $y \equiv 0 \ [2^\ell]$ d'où $y' \equiv 0 \ [2^\ell]$ c'est-à-dire $y' \in \mathbb{F}_\ell$. Comme $|y' - x| < 2^\ell$, on en déduit grâce au lemme 5.5 que y' est un arrondi fidèle de x dans $\mathbb{F}_\ell$. Si $y' = \bigtriangledown_\ell(x)$ alors $y \equiv \bigtriangledown_\ell(x) \ [2^{w+\ell}]$ avec $y \in \mathbb{F}_{\ell,w}$ donc $y = \bigtriangledown_{\ell,w}(x)$ d'après le lemme 5.27. De même, si $y' = \bigtriangleup_\ell(x)$ alors $y = \bigtriangleup_{\ell,w}(x)$. Dans tous les cas, y est bien un arrondi fidèle modulaire de x dans $\mathbb{F}_{\ell,w}$. □

Une propriété classique sur l'*overflow* modulaire est la suivante : si le résultat d'une somme obtenue en ignorant les problèmes d'*overflow* (c'est-à-dire calculée dans $\mathbb{F}_\ell$) est dans la période principale de $\mathbb{F}_{\ell,w}$, alors on obtient le même résultat en calculant cette somme dans $\mathbb{F}_{\ell,w}$ même si les calculs intermédiaires peuvent contenir des *overflows*. Plus précisément :

Lemme 5.30 (Règle de Jackson). *Pour tous $x_0, \ldots, x_{n-1} \in \mathbb{R}$, si :*

$$((x_0 \mathbin{\bigtriangledown\!\!\!+_\ell} x_1) \mathbin{\bigtriangledown\!\!\!+_\ell} x_2) \cdots \mathbin{\bigtriangledown\!\!\!+_\ell} x_{n-1} \in [-2^{w+\ell-1}, 2^{w+\ell-1}[\tag{5.113}$$

alors :

$$((x_0 \mathbin{\bigtriangledown\!\!\!+_{\ell,w}} x_1) \mathbin{\bigtriangledown\!\!\!+_{\ell,w}} x_2) \cdots \mathbin{\bigtriangledown\!\!\!+_{\ell,w}} x_{n-1} = ((x_0 \mathbin{\bigtriangledown\!\!\!+_\ell} x_1) \mathbin{\bigtriangledown\!\!\!+_\ell} x_2) \cdots \mathbin{\bigtriangledown\!\!\!+_\ell} x_{n-1} \tag{5.114}$$

Preuve. Montrons par récurrence sur $i \in \{0, \ldots, n-1\}$ que :

$$((x_0 \mathbin{\bigtriangledown\!\!\!+_{\ell,w}} x_1) \mathbin{\bigtriangledown\!\!\!+_{\ell,w}} x_2) \cdots \mathbin{\bigtriangledown\!\!\!+_{\ell,w}} x_i \equiv ((x_0 \mathbin{\bigtriangledown\!\!\!+_\ell} x_1) \mathbin{\bigtriangledown\!\!\!+_\ell} x_2) \cdots \mathbin{\bigtriangledown\!\!\!+_\ell} x_i \ [2^{w+\ell}] \tag{5.115}$$

L'initialisation est simplement $x_0 \equiv x_0 \ [2^{w+\ell}]$. Pour l'hérédité, considérons un $i \leq n-2$ et posons $A \triangleq ((x_0 \mathbin{\bigtriangledown\!\!\!+_{\ell,w}} x_1) \mathbin{\bigtriangledown\!\!\!+_{\ell,w}} x_2) \cdots \mathbin{\bigtriangledown\!\!\!+_{\ell,w}} x_i$
et $B \triangleq ((x_0 \mathbin{\bigtriangledown\!\!\!+_\ell} x_1) \mathbin{\bigtriangledown\!\!\!+_\ell} x_2) \cdots \mathbin{\bigtriangledown\!\!\!+_\ell} x_i$. On a $A \equiv B \ [2^{w+\ell}]$ par hypothèse de récurrence, donc $A + x_{i+1} \equiv B + x_{i+1} \ [2^{w+\ell}]$. Par le lemme 5.25 vu que $\bigtriangledown_\ell$ est quasi périodique, puis par définition de *cmod* :

$$\bigtriangledown_\ell(B + x_{i+1}) \equiv \bigtriangledown_\ell(A + x_{i+1}) \equiv cmod(\bigtriangledown_\ell(A + x_{i+1})) \ [2^{w+\ell}] \tag{5.116}$$

Grâce au lemme 5.26, on obtient bien $\bigtriangledown_\ell(B + x_{i+1}) \equiv \bigtriangledown_{\ell,w}(A + x_{i+1})\ [2^{w+\ell}]$.

La preuve par récurrence est terminée et on en déduit :

$$((x_0 \mathbin{\overline{\bigtriangledown}}_{\ell,w} x_1) \mathbin{\overline{\bigtriangledown}}_{\ell,w} x_2) \cdots \mathbin{\overline{\bigtriangledown}}_{\ell,w} x_{n-1} \equiv ((x_0 \mathbin{\overline{\bigtriangledown}}_{\ell} x_1) \mathbin{\overline{\bigtriangledown}}_{\ell} x_2) \cdots \mathbin{\overline{\bigtriangledown}}_{\ell} x_{n-1}\ [2^{w+\ell}] \tag{5.117}$$

De plus $((x_0 \mathbin{\overline{\bigtriangledown}}_{\ell} x_1) \mathbin{\overline{\bigtriangledown}}_{\ell} x_2) \cdots \mathbin{\overline{\bigtriangledown}}_{\ell} x_{n-1}$ est dans la période principale de $\mathbb{F}_{\ell,w}$ par hypothèse, et $((x_0 \mathbin{\overline{\bigtriangledown}}_{\ell,w} x_1) \mathbin{\overline{\bigtriangledown}}_{\ell,w} x_2) \cdots \mathbin{\overline{\bigtriangledown}}_{\ell,w} x_{n-1}$ appartient à $\mathbb{F}_{\ell,w}$ donc est aussi dans cette période principale. Ces deux nombres sont donc égaux. □

Cette propriété est connue en traitement du signal sous le nom de *règle de Jackson* [68], mais ce n'est ni plus ni moins qu'une conséquence de l'arithmétique modulaire. Je ne m'en servirai pas directement, mais des raisonnements apparaissant dans sa preuve (montrer la congruence des deux sommes par récurrence, et remarquer que les résultats sont égaux car ils sont à la fois congrus modulo $2^{w+\ell}$ et tous deux dans la période principale) seront réutilisés pour prouver les théorèmes 5.31 et 5.32 en section 5.4.3.

5.4.2 Formalisation de l'overflow modulaire

Cette section présente ma formalisation en Coq de l'*overflow* modulaire pour la virgule fixe. Je formalise d'abord des généralités sur les *overflows* en virgule fixe. Mais je me concentre rapidement sur l'*overflow* modulaire, et plus particulièrement sur les propriétés dont j'aurai besoin en section 5.4.3 pour prouver le comportement de l'algorithme à bits de garde en présence d'éventuels *overflows* modulaires.

Période principale. Je définis d'abord un prédicat `In_cpp` caractérisant la période principale $[-\frac{p}{2}, \frac{p}{2}[$ en fonction de la période p (la lettre `c` du fait que cette période principale est *centrée* en 0).

```
Definition In_cpp (p r : R) : Prop := - (p / 2) ≤ r < p / 2.
```

Je définis ensuite la fonction $cmod_p$ définie en section 5.4.1, qui à un réel associe son représentant modulo p dans cette période principale. La lettre `r` indique que c'est une fonction sur les réels, alors que d'habitude on s'attend plutôt à rencontrer une telle fonction sur les entiers. Je prouve que cette fonction est bien à valeurs dans la période principale.

```
Definition cmodr (p r : R) :=
  let k := Znearest (fun _ ⇒ true) (r/p) in
  r - IZR k * p.
Lemma In_cpp_cmodr (p : R) :
  0 < p → forall r : R, In_cpp p (cmodr p r).
```

Dans le cadre de la virgule fixe, la période principale est toujours une puissance de 2. On utilisera donc plutôt le prédicat `In_cppe` suivant (où `e` vient d'*exposant*) qui prend en argument l'exposant `ep` de la période, et applique `In_cpp` à $2^{\texttt{ep}}$. On a vu en section 5.4.1 que pour le format $\mathbb{F}_{\ell,w}$, l'exposant à utiliser sera $\texttt{ep} = w + \ell = msb + 1$.

```
Definition In_cppe (ep : Z) : R → Prop := In_cpp (twopow ep).
```

Fonction d'*overflow*. On dira qu'une fonction $\mathbb{R} \to \mathbb{R}$ est une *fonction d'overflow* si son image est incluse dans la période principale, et qu'elle est égale à l'identité sur cette période principale.

```
Definition Is_overf (ep : Z) (f : R → R) : Prop :=
  (forall r : R, In_cppe ep (f r)) ∧
  (forall r : R, In_cppe ep r → f r = r).
```

Par exemple, l'*overflow* saturé est représenté par la fonction qui est constante à $-2^{\mathtt{ep}-1}$ avant la période principale, l'identité sur la période principale, et constante à $2^{\mathtt{ep}-1} - 2^{\ell}$ après la période principale. On prouve qu'il s'agit bien d'une fonction d'*overflow*.

```
Definition overfSAT (ep lsb : Z) r :=
  if Rlt_le_dec r (- twopow (ep - 1))
  then - twopow (ep - 1)
  else
    if Rlt_le_dec r (twopow (ep - 1))
    then r
    else twopow (ep - 1) - twopow lsb.
Lemma overfSAT_Is_overf (ep lsb : Z) :
  (lsb ≤ ep - 1)%Z → Is_overf ep (overfSAT ep lsb).
```

L'*overflow* modulaire qui nous intéresse est représenté par la fonction $cmod_{2^{w+\ell}}$.

```
Definition overfWRAP (ep : Z) : R → R := cmodr (twopow ep).
Lemma overfWRAP_Is_overf (ep : Z) : Is_overf ep (overfWRAP ep).
```

J'aurais pu écrire d'autres définitions et propriétés sur les fonctions d'*overflow* de manière générale. Mais je n'en ai pas besoin pour l'algorithme de la section 5.4.3. Je n'utilise d'ailleurs pas l'*overflow* saturé, qui est seulement donné comme un exemple supplémentaire parmi plusieurs *overflows* définis pas la norme de SystemC [64].

Format $\mathbb{F}_{\ell,w}$. On se donne désormais un `lsb` et une largeur `w`. Cette dernière est de type `width`, c'est-à-dire un entier qui doit être supérieur ou égal à 2 (pour éviter des cas particuliers dégénérés). Il s'agit d'un type enregistrement (`Record`) comme pour les signaux en section 2.1.1. On utilise à nouveau la syntaxe `:>` pour ajouter une coercion, c'est-à-dire que Coq peut implicitement convertir un élément de type `width` en élément de type `Z`. Dans la suite, on considérera donc `w` comme un entier. On a vu que l'exposant de la période pour le format $\mathbb{F}_{\mathtt{lsb},\mathtt{w}}$ est $\mathtt{ep} = \mathtt{w} + \mathtt{lsb}$.

```
Record width :=
  { width_val :> Z ; width_prop : (2 ≤ width_val)%Z }.
Variables (lsb : Z) (w : width).
Notation ep := (w + lsb)%Z. (* = msb + 1 *)
```

On définit alors un prédicat caractérisant le format $\mathbb{F}_{\mathtt{lsb},\mathtt{w}}$, qui correspond à l'intersection de $\mathbb{F}_{\mathtt{lsb}}$ avec la période principale (ce prédicat était plus simple à définir dans `Prop` que dans `bool` à cause des inégalités sur `Z` dans `In_cppe`). Notons que cette définition est encore indépendante du mode d'*overflow* utilisé.

```
Definition isFIXo (x : R) : Prop := (isFIX lsb x ∧ In_cppe ep x).
```

***Overflow* modulaire.** On s'intéresse maintenant à la formalisation des arrondis et opérations en virgule fixe en présence d'*overflows* modulaires.

On a vu en section 5.4.1 qu'un arrondi vers $\mathbb{F}_{\ell,w}$ s'obtient à partir d'un arrondi $\bigcirc_\ell$ vers $\mathbb{F}_\ell$ (formalisé en section 5.1.3) en appliquant la fonction *cmod* à la fois avant et après $\bigcirc_\ell$. Ici, on note `oW` la fonction d'*overflow* représentant l'*overflow* modulaire, définie précédemment à partir de *cmod*.

```
Notation oW := (overfWRAP ep).
Definition roundFOW (rnd : R → Z) (r : R) : R :=
  oW (roundFIX lsb rnd (oW r)).
```

On définit ensuite comme en section 5.1.3 les opérations élémentaires dont on aura besoin dans les sommes de produits.

```
Definition binopFOW (rnd : R → Z) op (r1 r2 : R) : R :=
  roundFOW rnd (op r1 r2).
Definition addFOW (rnd : R → Z) : R → R → R :=
  binopFOW rnd Rplus.
Definition mulFOW (rnd : R → Z) : R → R → R :=
  binopFOW rnd Rmult.
```

On définit aussi la notion de quasi périodicité de la définition 5.24, pour une fonction $\mathbb{R} \to \mathbb{R}$ comme dans cette définition, mais aussi pour une fonction $\mathbb{R} \to \mathbb{Z}$ (grâce au plongement `IZR : Z → R`). En effet, la fonction d'arrondi `roundFIX lsb rnd` est quasi périodique si le mode d'arrondi `rnd : Z → R` (introduit en section 5.1.2 avec la bibliothèque Flocq) est quasi périodique. On verra plus loin un exemple de lemme exigeant en hypothèse que le mode d'arrondi `rnd` soit quasi périodique.

```
Definition quasiperR p (f : R → R) : Prop :=
  forall r, f (r + p) = f r + p.
Definition quasiperZ m (f : R → Z) : Prop :=
  forall r, f (r + IZR m) = (f r + m)%Z.
```

De nombreuses propriétés sur les arrondis et opérations sont prouvées en raisonnant modulo la période $2^{\mathtt{ep}}$. On définit pour cela la congruence modulo un réel : il s'agit du prédicat `eqmodr` (où `|d` est la divisibilité entière dans $\mathbb{R}$, définie en section 5.1.3). Le lemme `eq_In_cpp_eqmodr` sera souvent utilisé : si deux nombres appartenant à la période principale sont congrus modulo la période, alors ils sont égaux.

```
Definition eqmodr (p x y : R) : bool := p |d x - y.
Lemma eq_In_cpp_eqmodr (p r1 r2 : R) :
  In_cpp p r1 → In_cpp p r2 → eqmodr p r1 r2 → r1 = r2.
```

Par exemple, on se servira beaucoup du lemme `eqmodr_roundFIXFOW` : les arrondis vers $\mathbb{F}_{\mathtt{lsb}}$ et $\mathbb{F}_{\mathtt{lsb,w}}$ sont congrus modulo 2^{e_p} (à condition que le mode d'arrondi `rnd` soit quasi périodique). On remarque qu'il s'agit d'un corollaire du lemme 5.26.

```
Lemma eqmodr_roundFIXFOW (lsb : Z) (w : width) (rnd : R → Z) :
  quasiperZ (2 ^ w) rnd →
  forall r : R,
  eqmodr (twopow ep) (roundFIX lsb rnd r) (roundFOW lsb w rnd r).
```

On prouve aussi la règle de Jackson (lemme 5.30).

```
Lemma Jackson_rule_seqR (rnd : R → Z) :
       quasiperZ (2 ^ w) rnd →
       forall s : seq R,
In_cppe ep (\big[addFIX lsb rnd / 0]_(x ← s) x) →
    \big[addFOW / 0]_(x ← s) x
  = \big[addFIX lsb rnd / 0]_(x ← s) x.
```

Comme mentionné en section 5.4.1, ce lemme ne sera pas utilisé. Je l'ai tout de même prouvé en Coq car c'était un bon entraînement aux techniques utilisées pour raisonner sur des *overflows* modulaires.

Par ailleurs, on définit la notion d'arrondi fidèle modulaire (définition 5.28) et on prouve une condition suffisante (lemme 5.29) qui sera utilisée en section 5.4.3.

```
Definition faithfulFOW (x y : R) : Prop :=
  y = roundFOW Zfloor x \/ y = roundFOW Zceil x.
Lemma diff_lt_faithfulFOW (x y : R) :
  isFIXo y →
  (exists y', eqmodr (twopow ep) y y' ∧
              Rabs (y' - x) < twopow lsb) →
  faithfulFOW x y.
```

5.4.3 Algorithme fidèle à bits de garde avec des *overflows* modulaires

Cette section s'intéresse à l'algorithme fidèle à bits de garde de la section 5.2.4 en prenant cette fois-ci en compte des *overflows* modulaires potentiels, c'est-à-dire en calculant dans des formats de la forme $\mathbb{F}_{\ell,w}$. On utilise les mêmes notations que dans la section 5.2.4. Mais en plus du *lsb* final ℓ_f, on connaît à l'avance une *largeur finale* désirée w_f : la sortie doit maintenant être dans le format $\mathbb{F}_{\ell_f,w_f}$. Et en plus du *lsb* de calcul ℓ_c :

$$\ell_c \triangleq \ell_f - 1 - \lceil \log_2 n \rceil \qquad \text{(rappel de (5.37))}$$

on choisit une *largeur de calcul* w_c :

$$w_c \triangleq w_f + \ell_f - \ell_c \tag{5.118}$$

Cette largeur a été choisie de sorte que les *msbs* pour les calculs et pour la sortie soient identiques :

$$w_c + \ell_c - 1 = w_f + \ell_f - 1 \tag{5.119}$$

Les calculs sont les mêmes qu'en section 5.2.4 mais dans le format $\mathbb{F}_{\ell_c,w_c}$, avec un arrondi final vers $\mathbb{F}_{\ell_f,w_f}$:

$$q \triangleq \bigtriangledown^{+}_{\substack{\ell_c,w_c \\ 0\le i<n}} (s_i \bigtriangledown^{\times}_{\ell_c,w_c} t_i) \tag{5.120}$$

$$\texttt{res} \triangleq \bigcirc^{\tau}_{\ell_f,w_f}(q) \tag{5.121}$$

Encore une fois, on peut utiliser différents modes d'arrondi pour les calculs intermédiaires. Cependant, les fonctions d'arrondi vers $\mathbb{F}_{\ell_c}$ correspondantes

doivent être quasi périodiques (définition 5.24) de période $2^{w_c+\ell_c}$. On rappelle que les arrondis vers le bas et vers le haut conviennent, ainsi que les arrondis au plus proche si leur règle de *tie-breaking* est périodique (ce qui est le cas de *ties-to-even* et *ties-up* mais pas de *away-from-zero*). L'arrondi final au plus proche $\bigcirc^{\tau}_{\ell_f}$ doit lui aussi être quasi-périodique de période $2^{w_f+\ell_f}$ donc τ doit être périodique.

L'algorithme est récapitulé dans l'algorithme 19. La figure 5.18 illustre les bits impliqués sur le même exemple jouet que pour les autres algorithmes du chapitre.

Algorithme 19 : Algorithme à bits de garde avec des *overflows* potentiels

Donnés à l'avance : ℓ_f, w_f, n (et une règle τ périodique)
Calculés à l'avance : $\ell_c \triangleq \ell_f - 1 - \lceil \log_2 n \rceil$, $w_c \triangleq w_f + \ell_f - \ell_c$
Entrées : $s_0, \ldots, s_{n-1}, t_0, \ldots, t_{n-1}$
Sortie : `res`
Variable : v
$v \leftarrow s_0 \mathbin{\overline{\times}}_{\ell_c,w_c} t_0$
for $i \leftarrow 1$ **to** $n-1$ **do**
  $v \leftarrow v \mathbin{\overline{+}}_{\ell_c,w_c} (s_i \mathbin{\overline{\times}}_{\ell_c,w_c} t_i)$
end
`res` $\leftarrow \bigcirc^{\tau}_{\ell_f,w_f}(v)$

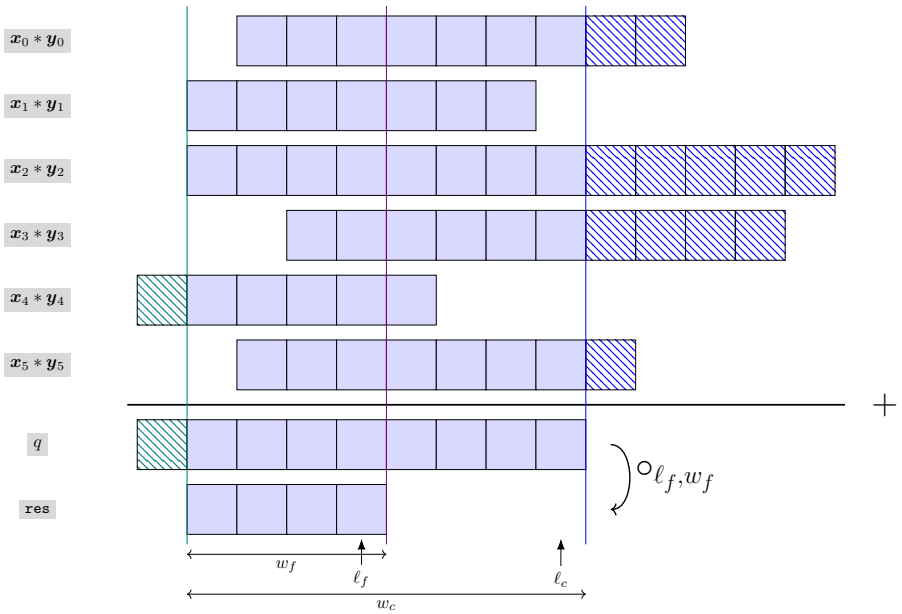


FIGURE 5.18 – Algorithme à bits de garde avec *overflow* : exemple

Théorème 5.31. *La sortie* `res` *est un arrondi fidèle modulaire (définition 5.28) du modèle mres dans* $\mathbb{F}_{\ell_f,w_f}$.

Preuve. On sait que $\texttt{res} \in \mathbb{F}_{\ell_f,w_f}$ puisqu'il est obtenu par arrondi vers ce format. D'après le lemme 5.29, il suffit de montrer qu'il existe $y' \in \mathbb{R}$ tel que $\texttt{res} \equiv y' \ [2^{\ell_f+w_f}]$ et $|y' - \mathit{mres}| < 2^{\ell_f}$.

Un arrondi au plus proche est fidèle, donc on sait que :

$$\texttt{res} = \bigtriangledown_{\ell_f,w_f}(q) \quad \vee \quad \texttt{res} = \bigtriangleup_{\ell_f,w_f}(q) \tag{5.122}$$

Notons τ' la règle de *tie-breaking* qui consiste à toujours arrondir vers le bas si $\texttt{res} = \bigtriangledown_{\ell_f,w_f}(q)$, et à toujours arrondir vers le haut sinon.

Par ailleurs, on note q' et $\texttt{res}'$ le résultat intermédiaire et la sortie de l'algorithme sans *overflow* de la section 5.2.4, où les arrondis sont effectués vers $\mathbb{F}_{\ell_c}$ ou $\mathbb{F}_{\ell_f}$.

On pose alors :

$$y' \triangleq \bigcirc^{\tau'}_{\ell_f,w_f}(q') \tag{5.123}$$

Comme l'arrondi vers le bas est quasi périodique, à partir du lemme 5.26 on obtient :

$$\forall i, \qquad s_i \ \bigtriangledown_{\ell_c,w_c} t_i \equiv s_i \ \bigtriangledown_{\ell_c} t_i \ [2^{w_c+\ell_c}] \tag{5.124}$$

On prouve ensuite (par récurrence comme dans la preuve du lemme 5.30) :

$$\bigoplus_{\substack{\ell_c,w_c \\ 0\le i<n}} (s_i \ \bigotimes_{\ell_c,w_c} t_i) \equiv \bigoplus_{\substack{\ell_c \\ 0\le i<n}} (s_i \ \bigotimes_{\ell_c} t_i) \ [2^{w_c+\ell_c}] \tag{5.125}$$

On reconnaît les définitions de q et q'. De plus $w_c + \ell_c = w_f + \ell_f$ par définition de w_c. On a donc :

$$q \equiv q' \ [2^{w_f+\ell_f}] \tag{5.126}$$

D'après le lemme 5.25, on a alors :

$$\bigtriangledown_{\ell_f,w_f}(q) \equiv \bigtriangledown_{\ell_f,w_f}(q') \ [2^{w_f+\ell_f}] \ \wedge \ \bigtriangleup_{\ell_f,w_f}(q) \equiv \bigtriangleup_{\ell_f,w_f}(q') \ [2^{w_f+\ell_f}] \tag{5.127}$$

De plus, comme 2^{ℓ_f} divise $2^{w_f+\ell_f}$, les nombres q et q' sont aussi congrus modulo 2^{ℓ_f}. Ils sont donc arrondis au plus proche vers $\mathbb{F}_{\ell_f,w_f}$ dans la même direction (vers le haut ou vers le bas) si ce ne sont pas des points milieu. Si ce sont des points milieu, cela dépend des règles de *tie-breaking* utilisées. Or on a justement défini τ' pour arrondir vers le bas si $\bigcirc^{\tau}_{\ell_f,w_f}(q) = \bigtriangledown_{\ell_f,w_f}(q)$ et sinon vers le haut. On en déduit que $\bigcirc^{\tau}_{\ell_f,w_f}(q)$ et $\bigcirc^{\tau'}_{\ell_f,w_f}(q')$ sont soit tous deux des arrondis vers le bas, soit tous deux des arrondis vers le haut. Grâce à l'équation (5.127), on a alors :

$$\bigcirc^{\tau}_{\ell_f,w_f}(q) \equiv \bigcirc^{\tau'}_{\ell_f,w_f}(q') \ [2^{w_f+\ell_f}] \tag{5.128}$$

c'est-à-dire :

$$\texttt{res} \equiv y' \ [2^{w_f+\ell_f}] \tag{5.129}$$

comme voulu. Or $|y' - \mathit{mres}| < 2^{\ell_f}$ d'après le théorème 5.12 (en utilisant τ' au lieu de τ pour l'arrondi final, mais on a vu que l'algorithme de la section 5.2.4 fonctionne pour n'importe quelle règle de *tie-breaking*). En utilisant le lemme 5.29, $\texttt{res}$ est donc bien un arrondi fidèle modulaire de mres. □

De plus, si la largeur w_f de la sortie est suffisamment grande pour que le modèle *mres* puisse être représenté dans $\mathbb{F}_{\ell_f,w_f}$ sans *overflow*, alors on peut montrer que la sortie de l'algorithme est un arrondi fidèle du modèle (et pas seulement un arrondi fidèle modulaire). Cela signifie que malgré d'éventuels *overflows* au cours des calculs, on obtient la même sortie que s'il n'y avait jamais d'*overflow*.

Théorème 5.32. *Si* $w_f \geq \lceil \log_2(|mres| + 2^{\ell_f}) \rceil - \ell_f + 1$ *alors la sortie* `res` *est un arrondi fidèle du modèle mres dans* $\mathbb{F}_{\ell_f}$.

Preuve. On a le même contexte que le théorème 5.31, avec une hypothèse supplémentaire sur w_f. On peut donc définir y' comme dans la preuve précédente, et on a encore `res` $\equiv y'$ $[2^{w_f+\ell_f}]$ et $|y' - mres| < 2^{\ell_f}$ et y' est un arrondi fidèle de *mres* dans $\mathbb{F}_{\ell_f}$ (toujours par le théorème 5.12).

Par ailleurs, l'hypothèse sur w_f implique :

$$w_f \geq \log_2(|mres| + 2^{\ell_f}) - \ell_f + 1 \quad i.e. \quad |mres| + 2^{\ell_f} \leq 2^{w_f+\ell_f-1} \tag{5.130}$$

On a alors :

$$|y'| \leq |mres| + |y' - mres| < 2^{w_f+\ell_f-1} - 2^{\ell_f} + 2^{\ell_f} = 2^{w_f+\ell_f-1} \tag{5.131}$$

En particulier :

$$-2^{w_f+\ell_f-1} \leq y' < 2^{w_f+\ell_f-1} \tag{5.132}$$

Ainsi, `res` et y' sont tous deux dans la période principale avec `res` $\equiv y'$ $[2^{w_f+\ell_f}]$, donc `res` $= y'$. Cela conclut la preuve puisqu'on a vu que y' est un arrondi fidèle de *mres* dans $\mathbb{F}_{\ell_f}$. □

Variante : multiplications arrondies au plus proche. Comme en section 5.2.4, on peut s'intéresser au cas où les multiplications intermédiaires sont effectuées avec un arrondi au plus proche (avec n'importe quelle règle de *tie-breaking* périodique). On peut alors prouver des théorèmes similaires aux théorèmes 5.31 et 5.32 où le *lsb* de calcul ℓ_c est celui du théorème 5.13.

Formalisation. On se donne les mêmes paramètres qu'en section 5.2.4, ainsi que la largeur finale w_f et la largeur de calcul w_c. On suppose seulement que cette dernière est supérieure à la valeur donnée dans l'équation (5.118), vu que cela suffira pour tout ce qu'on veut prouver. (En effet, on remarque que dans la preuve du théorème 5.31, la seule utilisation de cette valeur est pour passer d'une égalité modulo $2^{w_c+\ell_c}$ à la même égalité modulo $2^{w_f+\ell_f}$, ce pour quoi $w_c + \ell_c \geq w_f + \ell_f$ suffit.)

```
Variable lf : Z.
Variables s t : seq R.
Notation n := (minn (size s) (size t)).
Variable lc : Z.

Variables wf wc : Z.
Hypothesis wc_ge : (wf + lf - lc ≤ wc)%Z.
```

Comme en section 5.2.4, on définit la sortie `SOP_guard_OW` de l'algorithme en fonction d'arrondis pour les différentes étapes.

```
Definition prods_guard_OW (rndmul : R → Z) : seq R :=
  map2 (mulFOW lc wc rndmul) s t.
Definition SOP_guard_aux_OW (rndmul rndadd : R → Z) : R :=
  \big[addFOW lc wc rndadd / 0]_(x ← prods_guard_OW rndmul) x.
Definition SOP_guard_OW (rndmul rndadd rndf : R → Z) : R :=
  roundFOW lf wf rndf (SOP_guard_aux_OW rndmul rndadd).
```

On prouve alors les théorèmes 5.31 et 5.32. Les modes d'arrondi doivent être valides, mais aussi quasi périodiques (prédicat `quasiperZ`). Enfin, le *lsb* de calcul doit au plus la valeur donnée par l'équation (5.37). Les preuves Coq sont similaires aux preuves données pour les théorèmes 5.31 et 5.32, en ajoutant un grand nombre de lemmes techniques. Notamment, la règle de *tie-breaking* τ', en fonction de la direction de l'arrondi $\bigcirc_{\ell_f,w_f}$ appliqué à q, a été fastidieuse à définir.

```
Theorem SOP_guard_OW_faithfulFOW (rndmul rndadd : R → Z)
                                 (choicef : Z → bool) :
  Valid_rnd rndmul → Valid_rnd rndadd →
  quasiperZ (2 ^ wc) rndmul →
  quasiperZ (2 ^ wc) rndadd →
  quasiperZ (2 ^ wf) (Znearest choicef) →
  (lc ≤ lf - 1 - log2_ceil n)%Z →
  faithfulFOW lf wf SOP_model
              (SOP_guard_OW rndmul rndadd (Znearest choicef)).

Theorem SOP_guard_OW_faithful (rndmul rndadd : R → Z)
                              (choicef : Z → bool) :
  Valid_rnd rndmul → Valid_rnd rndadd →
  quasiperZ (2 ^ wc) rndmul →
  quasiperZ (2 ^ wc) rndadd →
  quasiperZ (2 ^ wf) (Znearest choicef) →
  (lc ≤ lf - 1 - log2_ceil n)%Z →
  (Zceil (Rlog2 (Rabs (SOP_model s t) + twopow lf)) - lf + 1
     ≤ wf)%Z →
  faithful lf SOP_model
           (SOP_guard_OW rndmul rndadd (Znearest choicef)).
```

On prouve aussi les théorèmes similaires pour le cas où les multiplications sont arrondies au plus proche.

```
Theorem SOP_guard_OW_rN_faithfulFOW (choicemul : Z → bool)
                  (rndadd : R → Z) (choicef : Z → bool) :
  Valid_rnd rndadd →
  quasiperZ (2 ^ wc) (Znearest choicemul) →
  quasiperZ (2 ^ wc) rndadd →
  quasiperZ (2 ^ wf) (Znearest choicef) →
  (lc ≤ lf - 1 - log2_floor n)%coqZ →
  faithfulFOW lf wf SOP_model
    (SOP_guard_OW (Znearest choicemul) rndadd (Znearest choicef)).
```

```
Theorem SOP_guard_OW_rN_faithful (choicemul : Z → bool)
               (rndadd : R → Z) (choicef : Z → bool) :
  Valid_rnd rndadd →
  quasiperZ (2 ^ wc) (Znearest choicemul) →
  quasiperZ (2 ^ wc) rndadd →
  quasiperZ (2 ^ wf) (Znearest choicef) →
  (lc ≤ lf - 1 - log2_floor n)%coqZ →
  (Zceil (Rlog2 (Rabs (SOP_model s t) + twopow lf)) - lf + 1
     ≤ wf)%Z →
  faithful lf SOP_model
    (SOP_guard_OW (Znearest choicemul) rndadd (Znearest choicef)).
```

Les définitions et propriétés mises en place pour l'*overflow* modulaire permettent donc bien de formaliser la prise en compte des *overflows* pour un algorithme utilisé en pratique. Il reste à étendre cette formalisation à l'accumulateur de Kulisch (sections 5.2.2 et 5.2.3) et au nouvel algorithme correctement arrondi au plus proche (section 5.3) : cette perspective ne devrait pas poser de difficulté majeure.

Chapitre 6

Conclusion et perspectives

Les contributions de ma thèse sont récapitulées en section 6.1. La section 6.2 présente des commentaires et retours d'expérience sur le développement de ma formalisation en Coq. Enfin, la section 6.3 propose des perspectives.

6.1 Contributions

Ma contribution principale est une formalisation en Coq de plus de 15000 lignes de code. Cette formalisation regroupe de nombreux éléments participant à l'analyse du comportement en précision finie des filtres numériques LTI. Les principaux éléments formalisés et théorèmes prouvés sont :

- les définitions et propriétés basiques des filtres numériques LTI, du *State-Space* et de la SIF (chapitres 2 et 3).
- des traductions vers la SIF depuis les réalisations suivantes : formes directes I et II, forme directe II transposée, décompositions en parallèle et en cascade, et *State-Space* (section 3.3). Ces traductions préservent le filtre modèle (ce qui est prouvé en Coq pour chacune) et le filtre implémenté (que je n'ai pas formalisé indépendamment de la SIF).
- des transformations du *State-Space* et de la SIF préservant le filtre modèle et modifiant le filtre implémenté, dans le but d'améliorer les propriétés de celui-ci (section 3.4).
- le théorème des filtres d'erreurs qui décrit la propagation des erreurs dans les itérations d'un filtre donné sous forme de *State-Space* ou surtout de SIF (sections 4.1 et 4.2).
- le théorème du *Worst-Case Peak Gain* qui borne la sortie d'un filtre en fonction des coefficients du filtre et d'une borne sur l'entrée (section 4.3), accompagné des preuves d'optimalité et même d'atteignabilité du *Worst-Case Peak Gain* (sections 4.3.2 et 4.3.3).
- trois algorithmes de somme de produits en virgule fixe : l'accumulateur de Kulisch (section 5.2.2), un algorithme fidèle à base de bits de garde (section 5.2.4) et mon algorithme correctement arrondi (section 5.3). On suppose ici qu'il n'y a jamais d'*overflow*. Chaque algorithme est accompagné d'une borne prouvée sur l'erreur en sortie.
- la définition de différents *overflows* en virgule fixe (section 5.4.2) et plus particulièrement les propriétés de l'*overflow* modulaire.

— la correction de l'algorithme fidèle à bits de garde même lorsque des *overflows* modulaires ont potentiellement lieu dans les calculs intermédiaires (section 5.4.3).

Les concepts et résultats formalisés correspondent tous à des travaux préexistants dans la littérature, à l'exception de l'algorithme de la section 5.3. Cet algorithme de somme de produits en virgule fixe est correctement arrondi pour l'arrondi au plus proche suivant n'importe quelle règle de *tie-breaking* désirée, et manipule moins de bits que l'accumulateur de Kulisch.

6.2 Commentaires sur le développement Coq

Cette section présente des remarques, détails techniques et retours d'expérience sur le développement de ma formalisation. Elle s'adresse plutôt à un lecteur déjà familier avec Coq. La section 6.2.1 explique pourquoi et comment je me suis mise à utiliser la bibliothèque MathComp [86] dans ma formalisation. La section 6.2.2 discute les indices de matrices : il s'agit d'ordinaux dans MathComp mais j'ai parfois besoin d'accéder à des coefficients pour des indices donnés sous forme d'entiers naturels. La section 6.2.3 présente mes tactiques pour résoudre des buts qui font intervenir un mélange d'entiers relatifs `int` de MathComp, d'entiers naturels `nat` et d'ordinaux `'I_n`. Enfin, la section 6.2.4 explique en quoi une tactique similaire à `ring` mais pour des matrices de toute taille m'aurait été utile.

6.2.1 Utilisation de la bibliothèque MathComp

Ma formalisation initiale d'une grande partie des éléments des chapitres 2 à 4 était assez différente de la formalisation actuelle. Elle est présentée dans [42, 41] [1]. Je n'utilisais pas du tout la bibliothèque MathComp. À la place, j'utilisais les matrices et sommes de Coquelicot [18]. Mais la principale différence était que les signaux scalaires et vectoriels n'étaient pas traités de manière factorisée avec des valeurs dans un ensemble vectoriel `VT`. À la place, ils étaient définis séparément comme des fonctions causales `Z → R` et `Z → vect n` respectivement (où `Z` et `R` sont les entiers relatifs et réels de la bibliothèque standard, et `vect n` un type que j'avais défini à partir des matrices de taille `n` × 1 de Coquelicot). Je devais donc dupliquer la plupart des définitions et preuves pour couvrir les deux familles de signaux.

Cette duplication de code n'était pas une bonne stratégie sur le long terme. J'ai donc décidé d'utiliser une structure algébrique couvrant à la fois les cas scalaire et vectoriel [2]. J'aurais pu utiliser les structures de Coquelicot. Cependant, je me suis aussi rendu compte que la littérature sur l'analyse d'erreurs des filtres fait intervenir beaucoup d'algèbre linéaire (par exemple pour calculer précisément le *Worst-Case Peak Gain* [109]). Or les matrices de MathComp disposent

1. Les fichiers Coq correspondants sont disponibles aux adresses `www.lri.fr/~gallois/code/coq-digital-filters-CICM18.tgz` et `www.lri.fr/~gallois/code/coq-filtresnum-JFLA19.tgz` respectivement.

2. Comme indiqué en section 2.1.1, la structure algébrique retenue est un module sur l'anneau $\mathbb{R}$, et plus généralement sur n'importe quel anneau commutatif `RT` ; j'appelle cela un espace vectoriel contrairement à l'usage mathématique car le type de MathComp correspondant est `vectType`.

de beaucoup plus de propriétés dans ce domaine que celles de Coquelicot. Les grands opérateurs de MathComp sont aussi plus génériques et comptent plus de lemmes que les sommes de Coquelicot. J'ai donc décidé de refaire une grande partie de ma formalisation afin de simultanément passer aux matrices et grands opérateurs de MathComp, et me mettre à utiliser des structures algébriques pour les signaux (celles de MathComp aussi, pour leur compatibilité avec les matrices et grands opérateurs précédents).

Cette modification majeure a pris beaucoup de temps (environ quatre mois vers la fin de ma première année) mais a répondu à mes attentes. Elle m'a permis de passer d'environ 6000 lignes de Coq à seulement 3500. En effet, elle a d'une part fait disparaître beaucoup d'éléments dupliqués. D'autre part, elle a permis d'enlever des lemmes que j'avais ajoutés sur les sommes et matrices de Coquelicot mais qui existent déjà sur celles de MathComp (j'ai tout de même dû ajouter de nouveaux lemmes sur ces dernières, mais en moins grand nombre). Ma formalisation est devenue beaucoup plus générale, et y ajouter de nouveaux éléments a effectivement été plus facile et plus rapide que pour la version initiale. Je n'ai pas eu le temps d'aborder les théorèmes faisant intervenir beaucoup d'algèbre linéaire, qui sont donc laissés en perspective en section 6.3.3, mais leur ajout devrait aussi être beaucoup facilité.

Expérience personnelle avec MathComp et SSReflect. Une partie importante de la difficulté et de la longueur de cette modification majeure tient au fait que je n'avais aucune familiarité avec la bibliothèque MathComp ni le langage de tactiques SSReflect [47]. J'ai rapidement trouvé ces derniers plus pratiques et agréables à utiliser que les bibliothèques et tactiques par défaut de Coq, à l'exception près que j'avais occasionnellement beaucoup de mal (plus que quand j'avais appris Coq sans SSReflect) à comprendre pourquoi une tactique ou un lemme ne pouvait pas être appliqué. Mais ces difficultés ont naturellement diminué au cours du temps, et je suis contente d'avoir appris à utiliser ces outils.

Indices des matrices. Une autre difficulté provient du fait que les matrices de MathComp utilisent des ordinaux pour indices, tandis que j'ai parfois besoin de récupérer des coefficients pour des indices entiers de type `nat`. Ma solution à ce problème est présentée en section 6.2.2.

Types pour les entiers relatifs. Enfin, j'ai dû décider quel type utiliser pour les entiers relatifs : `Z` de la bibliothèque standard ou `int` de MathComp. Dans les chapitres 2 à 4, j'ai utilisé `int` qui est plus adapté au cadre de SSReflect. Mais dans le chapitre 5, je me repose beaucoup sur la bibliothèque Flocq qui utilise déjà `Z`. Utiliser deux types différents pour les entiers relatifs dans ma formalisation n'a pas posé de problème car ils représentent des concepts différents : `int` est utilisé pour des indices de signaux, tandis que `Z` sert surtout à décrire des exposants pour des nombres à virgule fixe. Je pense que je préférerais légèrement utiliser `int` partout par souci d'homogénéité et de compatibilité avec SSReflect. Le travail principal requis pour cela serait l'ajout de conversions entre `Z` et `int` chaque fois que ma formalisation de la virgule fixe du chapitre 5 utilise des éléments de Flocq. Cela ne me paraît pas très difficile, d'autant plus que l'arithmétique sur les entiers relatifs requise dans le chapitre 5 est assez légère. Ce n'est cependant pas une perspective très prioritaire.

6.2.2 Indices entiers de matrices

Les indices des matrices de MathComp sont représentés par des ordinaux. Cependant, j'ai parfois besoin d'accéder à des coefficients correspondant à des indices représentés par des entiers naturels. J'ai donc défini `fmxn` qui, à une matrice à coefficients dans `RT`, associe une fonction `nat → nat → RT`. J'utilise ensuite la notation `A _[i,j]` qui applique `fmxn` à la matrice `A` et aux indices entiers `i` et `j` (les arguments `RT`, `h` et `w` étant implicites).

```
Definition fmxn (RT : ringType) (h w : nat) :
  'M_(h, w) → nat → nat → RT.
Notation "A _[ i , j ]" := (fmxn A i j) (at level 10).
```

La fonction `nat → nat → RT` associée à une matrice `A` de taille `h × w` est construite de la manière suivante. Je teste si les indices `i j : nat` vérifient `i < h` et `j < w`. Si c'est le cas, je construis les ordinaux correspondants dans `'I_h` et `'I_w` et je les fournis à `fun_of_matrix` de MathComp pour obtenir le coefficient de `A` correspondant. Si au moins une de ces inégalités est fausse, je renvoie simplement le zéro de `RT`.

C'est pour pouvoir utiliser zéro comme valeur de retour par défaut que cette définition porte sur les matrices à coefficients dans un anneau `RT` et non dans n'importe quel `T : Type`. J'aurais pu utiliser n'importe quelle structure algébrique avec un élément neutre, ou même un type quelconque en prenant aussi en argument un élément de ce type à renvoyer par défaut. Mais un `ringType` était simple à écrire et suffisant pour l'usage que j'en fais dans ma formalisation.

J'ai prouvé quelques lemmes basiques sur `fmxn`, comme par exemple sa compatibilité avec l'addition ou la multiplication. J'ai développé une tactique `dvlp_fmxn` assez simple, basée sur des `rewrite`, qui utilise ces lemmes pour transformer par exemple `(A + B *m C) _[i,j]` en `A _[i,j] + \sum_(0 ≤ k < m) B _[i,k] * C _[k,j]` (similairement à ce que permet des réécritures répétées du lemme `mxE` de MathComp lorsque les indices sont des ordinaux).

Une conséquence de cette méthode est que je dois souvent prouver des égalités ou inégalités contenant de multiples constructions d'ordinaux à partir d'entiers, et coercions d'ordinaux vers `nat`. Ceci est géré par les tactiques sur les entiers de la section 6.2.3.

Le besoin d'accéder à des coefficients pour des indices entiers, qui a motivé cette approche, est partiellement un reliquat de ma formalisation quand je n'utilisais pas MathComp (section 6.2.1). J'aurais peut-être pu complètement éviter ce problème en représentant beaucoup plus d'éléments de ma formalisation par des ordinaux plutôt que par des `nat`. Cependant, cela aurait fait apparaître des problèmes différents, comme de nombreuses additions et soustractions d'ordinaux dont la gestion peut être assez fastidieuse pour vérifier qu'on obtient les bons types d'ordinaux. Quoi qu'il en soit, la seule difficulté de la solution que j'ai choisie a été les tactiques mélangeant entiers et ordinaux, dont j'avais de toute façon besoin par ailleurs comme on verra en section 6.2.3.

6.2.3 Tactiques pour les entiers

J'ai expliqué en section 2.1.6 que j'utilise le type `int` (entiers relatifs de MathComp) pour les indices des signaux. Une conséquence de ce choix est que je manipule des expressions qui mélangent des `int` et des `nat` grâce à des conversions de type (notons que ceci concerne les chapitres 2 à 4 donc le type `Z` de la bibliothèque standard n'apparaîtra pas du tout dans cette section). Un exemple sera donné plus bas. MathComp fournit la coercion (section 1.2.4) évidente de `nat` vers `int` (il s'agit du constructeur `Posz`). J'ai aussi défini une coercion `nat_of_int` : il s'agit d'une fonction totale qui envoie tous les entiers négatifs vers 0, mais est généralement appliquée à des `z : int` pour lesquels on a une hypothèse `(0 ≤ z)%int`.

Par ailleurs, on a vu en section 6.2.2 que ma gestion des indices de matrices fait apparaître des mélanges d'entiers naturels et d'ordinaux. MathComp fournit à nouveau la coercion évidente de `'I_n` vers `nat` appelée `nat_of_ord`, mais aussi plusieurs moyens de construire un ordinal à partir d'un entier : par exemple, `Ordinal` produit l'ordinal de `'I_n` correspondant à un `i : nat` pour lequel on fournit une hypothèse `(i < n)%nat`.

J'aurais pu éviter le mélange de `int` et `nat` dans les indices de signaux en définissant certaines valeurs dans `int` au lieu de `nat`, comme par exemple la taille n_x du vecteur d'état d'une SIF. Mais ce sont des valeurs aussi utilisées dans les indices de matrices, donc cela aurait fait apparaître des mélanges de `int` et d'ordinaux à l'aide de conversions passant par `nat`. Finalement, définir n_x dans `int` ne résout pas le problème, donc c'est mieux de le définir dans `nat` : cela évite d'avoir à énoncer séparément l'hypothèse qu'il est positif.

J'ai donc défini des tactiques `auton` et `autoz` afin de résoudre automatiquement des égalités ou inégalités dans `nat` ou `int` faisant intervenir des conversions d'ordinaux ou entiers naturels ou relatifs. Je m'appuie sur les tactiques `ssrnatlia`, `ssrnatomega` et `intlia` du fichier `lia_tactics` de Mahboubi et Sibut-Pinote[3]. Ces dernières effectuent un prétraitement pour transformer les opérations de MathComp sur `nat` ou `int` en celles de la bibliothèque standard sur `nat` ou `Z`, afin de pouvoir appliquer les tactiques classiques `lia` ou `omega`. Mes propres tactiques rajoutent un prétraitement pour simplifier certaines conversions entre `'I_n`, `nat` et `int`. J'ajoute aussi au contexte des variantes d'hypothèses existantes où certains types ont été convertis. Par exemple, si le contexte contient une variable `i : 'I_n`, j'ajoute l'hypothèse `(nat_of_ord i < n)%nat` grâce au lemme `ltn_ord` de MathComp.

Considérons un exemple venant de la preuve que la traduction de la forme directe I vers la SIF préserve le filtre modèle (section 3.3.1). Voici le contexte et le but affichés par Coq à un moment au cours de la preuve (rappelons que `.+1` est le successeur unaire dans `nat`).

3. `www.github.com/amahboubi/lia4mathcomp`

```
1 subgoal
R : comUnitRingType
n : nat
a, b : nat → R
n_gt0 : (0 < n)%nat
u : int → R
i : nat
k : int
y_SIF : int → R
i_lt : (i.+1 < n)%nat
Hyp0 : (n - 1 - i.+1 < n)%nat
Hyp1 : (n - 1 - i < n)%nat
j : 'I_i.+1
______________________________________(1/1)
  b (n - i + j) * u (k - 1 - 1 - j)
- a (n - i + j) * y_SIF (k - 1 - 1 - j)
=
  b (n - i.+1 + j.+1) * u (k - 1 - j.+1)
- a (n - i.+1 + j.+1) * y_SIF (k - 1 - j.+1)
```

En simplifiant les opérations et applications identiques dans les deux membres, on se retrouve avec les deux sous-buts suivants à prouver (le sous-but numéro 3 est identique au numéro 1, et le 4 au 2). Le contexte n'a pas changé.

```
______________________________________(1/4)
(n - i + j)%nat = (n - i.+1 + j.+1)%nat
______________________________________(2/4)
(k - 1 - 1 - j)%int = (k - 1 - j.+1)%int
```

En affichant les coercions :

```
______________________________________(1/4)
(n - i + nat_of_ord j)%nat = (n - i.+1 + (nat_of_ord j).+1)%nat
______________________________________(2/4)
k - 1 - 1 - Posz (nat_of_ord j) = k - 1 - Posz (nat_of_ord j).+1
```

Je résous ces buts avec `auton` et `autoz` respectivement.

Il se trouve qu'ici, le premier but est résolu directement par `ssrnatlia`. Comme mon prétraitement peut être assez long à calculer par Coq (en particulier les ajouts d'hypothèses), mes tactiques essaient, entre chaque étape de prétraitement, d'appliquer les tactiques plus rapides de `lia_tactics`. Donc pour le premier but ci-dessus, `auton` utilise directement `ssrnatlia`.

Pour le second but, le prétraitement consiste seulement à récrire le lemme `PoszS : forall n : nat, forall n : nat, Posz n.+1 = (Posz n) + 1`, puis la tactique `intlia` est appliquée.

Mes tactiques effectuent aussi une simplification structurelle plus complète que celle des tactiques de `lia_tactics`. Par exemple, considérons le but suivant.

```
Goal forall (i n m : nat) (b : bool),
  i - n = m.+1 → (b && (i ≤ m)) = false.
```

J'ai écrit une tactique `simplP` utilisée dans mes autres tactiques, qui transforme ce but en :

```
1 subgoal
i, n, m : nat
b : bool
Hsimpl0 : i - n = m.+1
______________________________________(1/1)
~ (is_true b ∧ is_true (i ≤ m))
```

Ce nouveau but est résolu par `ssrnatlia`. Ma tactique `auton` inclut un appel à `simplP` donc peut directement résoudre le but initial.

Ces tactiques ont été assez longues à développer. Elles ont cependant facilité beaucoup de preuves, en particulier celles des transformations de diverses réalisations vers la SIF. Elles sont plus larges que les tactiques existantes lorsque plusieurs des types `int`, `nat` et `'I_n` sont en jeu, même si elles sont très orientées vers les besoins de ma formalisation : les réécritures et ajouts d'hypothèses ont été enrichis au cas par cas quand cela s'avérait nécessaire pour résoudre un but donné.

6.2.4 S'il existait une tactique similaire à ring pour les matrices de taille quelconque

Pour un entier n donné, les matrices carrées de taille $n \times n$ sur un anneau `RT` (type `'M[RT]_n` de MathComp) forment aussi un anneau. Cependant, l'ensemble de toutes les matrices (de n'importe quelle taille) n'est pas un anneau puisqu'on ne peut pas toujours multiplier deux éléments : il faut que la largeur du premier soit égale à la hauteur du second. On ne peut donc pas utiliser la tactique `ring` de la bibliothèque standard pour résoudre des égalités sur des matrices non carrées, comme par exemple : `(A + B + 0) *m C = B *m I *m C + A *m C` où `A` et `B` sont de taille $n_1 \times n_2$, `C` est de taille $n_2 \times n_3$, et `I` et `0` sont la matrice identité et la matrice nulle de tailles appropriées.

Des tactiques similaires à `ring` et `ring_simplify`, mais applicables à des matrices de taille quelconque, auraient facilité ma formalisation. Par exemple, voici un but apparaissant dans la preuve du théorème des filtres d'erreur pour une SIF (section 4.2.4).

```
L' *m compute_t'S k (x' k) + R' *m x' k + S' *m u k + eps_y k
=
L *m (Jinv *m (M *m x sif u k + N *m u k)) + Rsif *m x sif u k +
 S *m u k +
(SIF_L sif_Dc *m (invmx (SIF_J sif_Dc) *m (SIF_M sif_Dc *m
                                           x sif_Dc u k +
                                           SIF_N sif_Dc *m
                                           u k)) +
 SIF_R sif_Dc *m x sif_Dc u k + (S' *m u k + - S *m u k)) +
(L' *m (invmx J' *m (M' *m x sif_Dop eps k +
                     row_mx (row_mx 1%:M 0) 0 *m
                     eps k)) + R' *m x sif_Dop eps k +
 row_mx (row_mx 0 0) 1%:M *m eps k)
```

Je n'ai pas inclus le très long contexte, mais toutes les expressions entre les opérateurs `+` et `*m` sont bien des matrices de tailles variées. Il reste des réécritures à faire de ces matrices individuelles, que je n'ai pas encore appliquées pour ne pas rendre le but encore plus long. Mais suite à celles-ci, on obtient un but prouvable uniquement à l'aide des propriétés basiques de l'addition et la multiplication matricielles (associativité, commutativité de l'addition, distributivité, élimination de `0` ou `I`). Un tel but pourrait donc être résolu par une tactique similaire à `ring`.

On remarque que le même terme `S' *m u k` apparaît de part et d'autre de l'égalité. Une tactique similaire à `ring_simplify` pourrait déjà simplifier ce terme dans le but tel qu'il est affiché. À la place, j'ai défini une tactique `strike_both_sides` à laquelle il faut fournir explicitement le terme à simplifier (ici `strike_both_sides (S' *m u k)`, et qui utilise des réécritures d'associativité et commutativité de l'addition pour isoler ce terme afin de pouvoir l'éliminer avec `f_equal`. De même, la tactique `strike_opposites (S *m u k)` cherche le terme `S *m u k` et son opposé d'un même côté de l'égalité et les simplifie. Grâce à ces deux tactiques, j'arrive à diminuer petit à petit le but, jusqu'au moment où je peux à la main appliquer les associativités et commutativités adéquates pour arriver à deux membres d'égalité identiques. Le niveau d'automatisation laisse à désirer puisque je dois décider quand appeler ces deux tactiques et leur fournir le terme à simplifer. Mais ce n'est pas vraiment un problème car ma formalisation ne contient que deux preuves avec des expressions suffisamment complexes pour que je décide d'utiliser ces tactiques.

La plupart des buts rencontrés qui auraient pu être résolus immédiatement par une tactique similaire `ring` sont beaucoup plus simples que l'exemple ci-dessus, et ne sont pas très longs à prouver en appliquant des lemmes élémentaires sur l'addition et la multiplication. Pour ma formalisation, une telle tactique n'aurait donc représenté ni plus ni moins qu'un léger gain de temps.

Une tactique similaire à `ring` pour des matrices de taille quelconque pourrait cependant être très utile pour tenter d'automatiser certaines preuves comme la formalisation de nouvelles réalisations, mais ceci est une perspective à long terme (section 6.3.3). Une telle tactique pourrait peut-être être implémentée en convertissant des matrices de taille quelconque vers l'anneau des matrices infinies (indexées sur $\mathbb{N} \times \mathbb{N}$) dont chaque ligne a un nombre fini d'éléments non nul [69] (ou bien chaque colonne a cette propriété, ou encore à la fois chaque ligne et chaque colonne).

6.3 Perspectives

Cette section propose différentes perspectives à court (section 6.3.1), moyen (section 6.3.2) et plus long (section 6.3.3) terme.

6.3.1 À court terme : expliciter la borne sur l'erreur finale

On a vu dans le chapitre 4 que l'erreur finale d'un filtre s'exprime en fonction d'une borne sur les erreurs locales dans les opérations de somme de produits. C'est pour cela qu'on a étudié les algorithmes de somme de produits dans le chapitre 5. Mais comme on a vu sur le graphe de dépendance des fichiers Coq

(figure 1.13 en section 1.6.3), ma formalisation ne met pas encore en commun les résultats de ces deux chapitres. La première perspective consiste donc à combiner ces éléments pour obtenir une expression de l'erreur finale d'un filtre en fonction des *Worst-Case Peak Gains* des filtres d'erreur, de l'algorithme de somme de produits utilisés et des formats de virgule fixe des différentes variables. Cependant, il manque encore des étapes pour rendre une telle expression de l'erreur finale utilisable en pratique. Notamment, le développement des expresssions algébriques pour les *Worst-Case Peak Gains* des filtres d'erreur risque d'être long et complexe, et est donc laissé en perspective à plus long terme en section 6.3.3.

Pour contourner ces étapes potentiellement difficiles, une autre perspective à court terme consiste à étudier la formalisation d'un filtre concret, donné sous forme de coefficients explicites pour n'importe laquelle des réalisations qui ont été formellement traduites vers la SIF en section 3.3. L'objectif est alors d'obtenir une valeur numérique d'une borne sur l'erreur finale de ce filtre. Pour cela, on doit aussi se donner des formats explicites de virgule fixe pour chaque variable et un algorithme de somme de produits choisi parmi ceux formalisés. On peut alors calculer explicitement les filtres d'erreurs et les bornes sur les erreurs locales. On a ensuite besoin de borner les *Worst-Case Peak Gains* des filtres d'erreurs explicites obtenus. Un algorithme existe pour cela [109], mais sa formalisation risque d'être assez complexe et sera donc discutée en section 6.3.3. Dans un premier temps, on pourrait peut-être prouver plus facilement une surapproximation moins précise de cette borne. On pourrait alors calculer une borne numérique formellement prouvée sur l'erreur finale du filtre concret étudié. Une question qui se pose d'ailleurs pour cette approche est le choix de ce filtre à étudier. On souhaite qu'il soit le plus représentatif possible des filtres utilisés en pratique. En effet, l'idée est d'évaluer le travail requis pour formaliser des filtres individuels de manière générale, ainsi que d'identifier d'éventuelles étapes manquantes.

6.3.2 Enrichissements de la formalisation à moyen terme

Ma formalisation peut être étendue dans de nombreuses directions. Les suggestions suivantes sont moins prioritaires que l'obtention d'une borne finale explicite, mais ne semblent pas très difficiles. On peut par exemple prouver des transformations vers la SIF pour d'autres réalisations que celles qui ont été traitées en section 3.3. Par ailleurs, je n'ai prouvé la prise en compte des *overflows* que pour l'algorithme fidèle à bits de garde en section 5.4.3. Mais je m'attends à ce que la prouver pour les autres algorithmes de somme de produits ne présente pas de difficulté majeure. Ou encore, on peut étendre la formalisation pour gérer explicitement le cas d'un système de contrôle-commande en boucle fermée (section 1.4.2, figure 1.9), par exemple en considérant une SIF décrivant un tel système [57].

On peut aussi s'intéresser au second objectif présenté en section 1.6.1 (le premier était borner l'erreur finale) : vérifier des formats de virgule fixe choisis pour les différentes variables du programme, en particulier s'assurer qu'il n'y a pas d'*overflow* imprévu. Comme expliqué en section 4.3.4, on peut facilement modifier la SIF étudiée pour qu'elle renvoie comme sorties les différentes variables du programme (à savoir les variables d'état $\boldsymbol{x}_i(k)$, les variables intermédiaires $\boldsymbol{t}_i(k)$ et les sorties $\boldsymbol{y}_i(k)$, en considérant qu'une somme de produits est un algorithme externe qui gère déjà ses propres variables). On peut alors appliquer le théorème du *Worst-Case Peak Gain* à ce nouveau filtre pour ob-

tenir des bornes sur ces variables, afin de déterminer des formats garantissant qu'il n'y aura pas d'*overflow* [29]. Formaliser la nouvelle SIF pour exprimer des bornes sur les variables du programme en fonction du *Worst-Case Peak Gain* de ce filtre ne devrait pas poser de difficulté. Cependant, comme on a vu en section 4.3.4, en déduire des valeurs numériques rigoureuses de ces bornes est plus complexe [108]. Entre autres, on a encore une fois besoin d'avoir prouvé une borne sur le calcul du *Worst-Case Peak Gain* d'un filtre (comme mentionné en section 6.3.1 et détaillé en section 6.3.3).

6.3.3 Diverses perpectives à plus long terme

Les perspectives suivantes sont à plus long terme car elles demandent beaucoup de temps, ou bien leur difficulté est difficile à évaluer sans avoir commencer à travailler dessus.

Calcul du *Worst-Case Peak Gain* en précision finie. Si on utilise l'approche consistant à calculer numériquement la borne d'erreur pour un filtre dont les coefficients sont donnés explicitement, il faudra entre autres déterminer une borne numérique formellement prouvée du *Worst-Case Peak Gain* de chaque filtre d'erreur. En effet, même lorsqu'on connaît explicitement les coefficients d'un filtre, son *Worst-Case Peak Gain* ne peut pas être simplement calculé : d'une part il faut prendre en compte les erreurs d'arrondi d'un tel calcul qui serait lui-même effectué en précision finie, et d'autre part on se heurte ici à une somme infinie. Volkova *et al.* [109] résolvent ce problème en proposant un algorithme qui permet de calculer le *Worst-Case Peak Gain* à un ε près, pour n'importe quel $\varepsilon > 0$ voulu. Mais cet algorithme risque d'être difficile à prouver en Coq car la preuve papier est déjà complexe. Elle est d'ailleurs basée sur de l'algèbre linéaire, avec entre autres une décomposition en valeurs propres pour une exponentiation de matrice : sa formalisation en Coq s'appuierait donc probablement beaucoup sur les propriétés d'algèbre linéaire fournies par la bibliothèque MathComp. Cependant, il n'est peut-être pas nécessaire de prouver formellement l'algorithme complet : on a seulement besoin de prouver que la borne obtenue majore effectivement la valeur exacte du *Worst-Case Peak Gain*. On peut envisager d'utiliser comme oracle un algorithme externe calculant cette borne et des certificats nous permettant de la prouver.

Expression algébrique d'une borne sur le *Worst-Case Peak Gain* des filtres d'erreur. Une autre perspective à long terme serait de borner le *Worst-Case Peak Gain* de chacun des deux filtres d'erreurs de manière systématique, c'est-à-dire de déterminer une expression algébrique pour chaque borne. Ceci a été discuté en section 4.3.5. Cette perspective s'inscrit dans la direction de théorèmes généraux sur des filtres universellement quantifiés, plutôt que pour des valeurs numériques particulières comme la perspective précédente.

Meilleur algorithme correctement arrondi de somme de produits. L'algorithme correctement arrondi proposé en section 5.3 est finalement plutôt un algorithme de somme. En effet, il suppose simplement que les produits ont été calculés avec suffisamment de précision pour être exacts. On peut peut-être diminuer la précision des produits (c'est-à-dire les calculer dans des formats de *lsb* plus grand) quitte à augmenter légèrement celle des sommes pour conserver la

propriété d'arrondi correct, et obtenir un algorithme plus efficace au total (moins de bits manipulés donc moins de temps de calcul et d'énergie consommée). C'est déjà ce que fait l'algorithme à bits de garde (section 5.2.4, [111]) mais celui-ci assure seulement un arrondi fidèle, tandis que l'objectif serait de garantir un arrondi correct (qui est déterministe et présente une erreur maximale deux fois plus petite).

Automatisation des traductions vers la SIF de nouvelles réalisations. Les formalisations de multiples réalisations vers la SIF (section 3.3) présentent beaucoup de similarités. En particulier, automatiser une grande partie des preuves de conservation du filtre modèle semble envisageable. Il faudrait cependant déjà disposer de tactiques similaires à `ring` pour des matrices de taille quelconque (comme discuté en section 6.2.4) vu que ces preuves font intervenir beaucoup d'égalités de sommes et produits de matrices.

Lien avec d'autres formalisations. On pourrait raccorder ma formalisation à d'autres formalisations en Coq. Par exemple, passer par le compilateur Lustre certifié de Bourke *et al.* [22] permettrait d'obtenir du code assembleur correspondant aux filtres formalisés. Cela fournirait aussi une définition de filtre implémenté indépendante de ma formalisation de la SIF : on pourrait alors énoncer et prouver la conservation du filtre implémenté par les traductions de diverses réalisations vers la SIF de la section 3.3.

Génération de code. Enfin, un objectif à long terme est de générer du code vérifié implémentant un filtre numérique, accompagné d'une borne certifiée sur l'erreur finale. On pourrait utiliser le mécanisme d'extraction de Coq [79]. Une autre possibilité serait d'utiliser le compilateur Lustre certifié mentionné ci-dessus [22]. Cependant, il faudrait peut-être l'adapter pour pouvoir utiliser un algorithme de somme de produits adapté au cas d'un filtre (chapitre 5). Ou encore, on pourrait passer par l'outil de vérification déductive Why3 [40] et sa propre extraction de code. Pour cela, une partie des définitions de la formalisation devraient être retranscrites en Why3, mais les preuves Coq pourraient être directement appelées. Cette approche pourrait d'ailleurs mettre à profit la traduction automatique d'un modèle en Simulink vers Why3 de Araiza-Illan et Eder [5].

Quel que soit le mécanisme de génération de code choisi, on pourrait ajouter une étape d'optimisation de code, où la structure des calculs est modifiée dans le but d'obtenir une meilleure implémentation (par exemple plus rapide à calculer ou ayant un meilleur comportement numérique). On pourrait pour cela s'appuyer sur des transformations automatiques de code généralistes [35], ou bien des techniques spécifiques aux algorithmes de filtre comme le changement de base ou la transposition de la SIF, qui ont été formalisés en section 3.4. Dans tous les cas, rappelons qu'il n'est pas nécessaire de certifier le procédé d'optimisation lui-même tant qu'on prouve que le résultat décrit encore le même filtre modèle.

Bibliographie

[1] Renato B Abreu, Lucas Cordeiro, et al. Verifying fixed-point digital filters using SMT-based bounded model checking. *arXiv preprint arXiv:1305.2892*, 2013.

[2] Behzad Akbarpour and Sofiène Tahar. Error analysis of digital filters using HOL theorem proving. *Journal of Applied Logic*, 5(4):651–666, 2007. from the 4th International Workshop on Computational Models of Scientific Reasoning and Applications.

[3] Behzad Akbarpour, Sofiène Tahar, and Abdelkader Dekdouk. Formalization of fixed-point arithmetic in HOL. *Formal Methods in System Design*, 27(1):173–200, Sep 2005.

[4] J Dwight Aplevich. *Implicit linear systems*, volume 83. Springer, 1991.

[5] Dejanira Araiza-Illan, Kerstin Eder, and Arthur Richards. Formal verification of control systems' properties with theorem proving. In *2014 UKACC International Conference on Control (CONTROL)*, pages 244–249. IEEE, 2014.

[6] David Báez-López, David Báez-Villegas, René Alcántara, Juan José Romero, and Tomás Escalante. Package for filter design based on MATLAB. *Computer Applications in Engineering Education*, 9(4):259–264, 2001.

[7] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[8] V Balakrishnan and S Boyd. On computing the worst-case peak gain of linear systems. *Systems & Control Letters*, 19(4):265–269, 1992.

[9] Mordechai Ben-Ari. The bug that destroyed a rocket. *ACM SIGCSE Bulletin*, 33(2):58–59, 2001.

[10] Yves Bertot. A short presentation of Coq. In *International Conference on Theorem Proving in Higher Order Logics*, pages 12–16. Springer, 2008.

[11] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[12] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In *International Conference on Theorem Proving in Higher Order Logics*, pages 86–101. Springer, 2008.

[13] Iury Bessa, Renato Abreu, João Edgar Filho, and Lucas Cordeiro. SMT-based bounded model checking of fixed-point digital controllers. In *IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society*, pages 295–301. IEEE, 2014.

[14] Iury Bessa, Hussama Ibrahim, Lucas Cordeiro, and Joao Edgar Chaves Filho. Verification of delta form realization in fixed-point digital controllers using bounded model checking. In *2014 Brazilian Symposium on Computing Systems Engineering*, pages 49–54. IEEE, 2014.

[15] M Blair, S Obenski, and P Bridickas. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. *United States Department of Defense, General Accounting Office, Washington, DC, Technical Report GAO/IMTEC-92-26*, 1992.

[16] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, page 196–207, New York, NY, USA, 2003. Association for Computing Machinery.

[17] Sylvie Boldo, Diane Gallois-Wong, and Thibault Hilaire. A correctly-rounded fixed-point-arithmetic dot-product algorithm. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, pages 9–16. IEEE, 2020.

[18] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015.

[19] Sylvie Boldo and Guillaume Melquiond. Emulation of FMA and Correctly-Rounded Sums: Proved Algorithms Using Rounding to Odd. *IEEE Transactions on Computers*, 57(4):462–471, 2008.

[20] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 243–252. IEEE, 2011.

[21] Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System*. Elsevier, 2017.

[22] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 586–601, 2017.

[23] Stephen Boyd and John Doyle. Comparison of peak and RMS gains for discrete-time systems. *Systems & control letters*, 9(1):1–6, 1987.

[24] Peter R Cappello and Willard L Miranker. Systolic super summation device, June 14 1988. US Patent 4,751,665.

[25] Victor A Carreño. Interpretation of IEEE-854 floating-point standard and definition in the HOL system. Technical report, NASA, Langley Research Center, 1995.

[26] Thiago Cavalcante, Iury Bessa, Lucas Cordeiro, et al. Formal non-fragile verification of step response requirements for digital state-feedback control systems. *Journal of Control, Automation and Electrical Systems*, pages 1–17, 2020.

[27] Lennon Chaves, Iury V Bessa, Hussama Ismail, Adriano Bruno dos Santos Frutuoso, Lucas Cordeiro, and Eddie Batista de Lima Filho.

DSVerifier-aided verification applied to attitude control software in unmanned aerial vehicles. *IEEE Transactions on Reliability*, 67(4):1420–1441, 2018.

[28] Lennon C Chaves, Hussama I Ismail, Iury V Bessa, Lucas C Cordeiro, and Eddie B de Lima Filho. Verifying fragility in digital systems with uncertainties using DSVerifier v2.0. *Journal of Systems and Software*, 153:22–43, 2019.

[29] Michael Christensen and Fred J Taylor. Fixed-point-IIR-filter challenges. *EDN Netw*, 51(23):111–122, 2006.

[30] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19(1):7–34, 2001.

[31] George Constantinides, Peter YK Cheung, and Wayne Luk. *Synthesis and optimization of DSP algorithms*. Springer Science & Business Media, 2007.

[32] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering*, 38(4):957–974, 2011.

[33] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[34] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *European Symposium on Programming*, pages 21–30. Springer, 2005.

[35] Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. Improving the numerical accuracy of programs by automatic transformation. *International Journal on Software Tools for Technology Transfer*, 19(4):427–448, 2017.

[36] Marc Daumas and David W Matula. Validated Roundings of Dot Products by Sticky Accumulation. *IEEE Transactions on Computers*, 46(5):623–629, 1997.

[37] Florent De Dinechin, Matei Istoan, and Abdelbassat Massouri. Sum-of-product architectures computing just right. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 41–47. IEEE, 2014.

[38] Jérôme Feret. Static Analysis of Digital Filters. In D.A. Schmidt, editor, *the 13th European Symposium on Programming - ESOP 2004*, volume 2986 of *Lecture Notes in Computer Science*, pages 33–48, Barcelona, Spain, March 2004. Springer.

[39] Jean-Christophe Filliâtre. Deductive software verification, 2011.

[40] Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where programs meet provers. In *European symposium on programming*, pages 125–128. Springer, 2013.

[41] Diane Gallois-Wong. Formalisation en Coq d'algorithmes de filtres numériques. In *Journées Francophones des Langages Applicatifs*, 2019.

[42] Diane Gallois-Wong, Sylvie Boldo, and Thibault Hilaire. A Coq formalization of digital filters. In *International Conference on Intelligent Computer Mathematics*, pages 87–103. Springer, 2018.

[43] Herman Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[44] Michel Gevers and Gang Li. *Parametrizations in control, estimation and filtering problems: accuracy aspects*. Springer Science & Business Media, 2012.

[45] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[46] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179. Springer, 2013.

[47] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *Journal of formalized reasoning*, 3(2):95–152, 2010.

[48] Stef Graillat, Christoph Q. Lauter, Ping Tak Peter Tang, Naoya Yamanaka, and Shin'ichi Oishi. Efficient Calculations of Faithfully Rounded l2-Norms of n-Vectors. *ACM Transactions on Mathematical Software*, 41(4):24:1, October 2015.

[49] H Hanselmann. Implementation of digital controllers – A survey. *Automatica*, 23(1):7–32, 1987.

[50] John Harrison. Floating point verification in HOL light: the exponential function. In *International Conference on Algebraic Methodology and Software Technology*, pages 246–260. Springer, 1997.

[51] John Harrison. *Handbook of practical logic and automated reasoning*. Cambridge University Press, 2009.

[52] Osman Hasan and Sofiene Tahar. Formal verification methods. In *Encyclopedia of Information Science and Technology, Third Edition*, pages 7162–7170. IGI Global, 2015.

[53] Nicholas J Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002.

[54] T. Hilaire, P. Chevrel, and Y. Trinquet. Implicit state-space representation: a unifying framework for FWL implementation of LTI systems. *IFAC Proceedings Volumes*, 38(1):285–290, 2005.

[55] Thibault Hilaire. *Analyse et synthèse de l'implémentation de lois de contrôle-commande en précision finie : étude dans le cadre des applications automobiles sur calculateur embarqué*. PhD thesis, Université de Nantes, 2006.

[56] Thibault Hilaire, Philippe Chevrel, and James F Whidborne. A unifying framework for finite wordlength realizations. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 54(8):1765–1774, 2007.

[57] Thibault Hilaire, Philippe Chevrel, and James F Whidborne. Finite wordlength controller realisations using the specialised implicit form. *International Journal of Control*, 83(2):330–346, 2010.

[58] Thibault Hilaire and Benoit Lopez. Reliable implementation of linear filters with fixed-point arithmetic. In *SiPS 2013 Proceedings*, pages 401–406. IEEE, 2013.

[59] Thibault Hilaire, Daniel Ménard, and Olivier Sentieys. Roundoff noise analysis of finite wordlength realizations with the implicit state-space framework. In *2007 15th European Signal Processing Conference*, pages 1019–1023. IEEE, 2007.

[60] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[61] Institute of Electrical and Electronics Engineers. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985, 1985.

[62] Institute of Electrical and Electronics Engineers. IEEE Standard SystemC(R) Language Reference Manual. IEEE Std 1666-2005, 2006.

[63] Institute of Electrical and Electronics Engineers. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008, 2008.

[64] Institute of Electrical and Electronics Engineers. IEEE Standard for Standard SystemC Language Reference Manual. IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005) - Redline, 2012.

[65] Institute of Electrical and Electronics Engineers. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008), 2019.

[66] International Organization for Standardization. Programming languages – C – Extensions to support embedded processors. ISO/IEC TR 18037:2008, 2008.

[67] Hussama I Ismail, Iury V Bessa, Lucas C Cordeiro, Eddie B de Lima Filho, and João E Chaves Filho. DSVerifier: A bounded model checking tool for digital systems. In *International SPIN Workshop on Model Checking of Software*, pages 126–131. Springer, 2015.

[68] L Jackson, J Kaiser, and H McDonald. An approach to the implementation of digital filters. *IEEE Transactions on Audio and Electroacoustics*, 16(3):413–421, 1968.

[69] Nathan Jacobson. *Structure of rings*, volume 37. American Mathematical Soc., 1956.

[70] Thomas Kailath. *Linear systems*, volume 156. Prentice-Hall Englewood Cliffs, NJ, 1980.

[71] Reinhard Kirchner and Ulrich Kulisch. Arithmetic for vector processors. In *1987 IEEE 8th Symposium on Computer Arithmetic (ARITH)*, pages 256–269. IEEE, 1987.

[72] Ali Kireçci, Mehmet Topalbekiroglu, and Ilyas Eker. Experimental evaluation of a model reference adaptive control for a hydraulic robot: a case study. *Robotica*, 21(1):71, 2003.

[73] Andreas Knofel. Fast hardware units for the computation of accurate dot products. In *Proceedings 10th IEEE Symposium on Computer Arithmetic*, pages 70–71. IEEE Computer Society, 1991.

[74] Robbert Krebbers and Bas Spitters. Type classes for efficient exact real arithmetic in Coq. *arXiv preprint arXiv:1106.3448*, 2011.

[75] Ulrich Kulisch. *Computer arithmetic and validity: theory, implementation, and applications*, volume 33. Walter de Gruyter, 2013.

[76] Vincent Lefèvre. Correctly rounded arbitrary-precision floating-point summation. *IEEE Transactions on Computers*, 66(12):14, 2017.

[77] Catherine Lelay. *Repenser la bibliothèque réelle de Coq : vers une formalisation de l'analyse classique mieux adaptée*. PhD thesis, Université Paris Sud-Paris XI, 2015.

[78] Xavier Leroy et al. The CompCert verified compiler, 2012.

[79] Pierre Letouzey. A new extraction for Coq. In *International Workshop on Types for Proofs and Programs*, pages 200–219. Springer, 2002.

[80] Nancy G Leveson and Clark S Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[81] Gang Li, Jian Chu, and Jun Wu. A matrix factorization-based structure for digital filters. *IEEE transactions on signal processing*, 55(10):5108–5112, 2007.

[82] Gang Li, Michel Gevers, and Youxian Sun. Performance analysis of a new structure for digital filter implementation. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 47(4):474–482, 2000.

[83] Gang Li and Zixue Zhao. On the generalized DFIIt structure and its state-space realization in digital filter implementation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 51(4):769–778, 2004.

[84] Xiaoye S Li, James W Demmel, David H Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y Kang, Anil Kapur, Michael C Martin, et al. Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions on Mathematical Software (TOMS)*, 28(2):152–205, 2002.

[85] Benoit Lopez. *Implémentation optimale de filtres linéaires en arithmétique virgule fixe*. PhD thesis, Université Pierre et Marie Curie-Paris VI, 2014.

[86] Assia Mahboubi and Enrico Tassi. Mathematical components, 2017.

[87] Micaela Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique*. PhD thesis, Paris 6, 2001.

[88] Steve McConnell. *Code complete*. Pearson Education, 2004.

[89] Daniel Menard and Olivier Sentieys. A methodology for evaluating the precision of fixed-point systems. In *2002 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages III–3152. IEEE, 2002.

[90] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. *Handbook of Floating-Point Arithmetic*, volume 1. Springer, 2018.

[91] Michael Muller, Christine Rub, and W Rulling. Exact accumulation of floating-point numbers. In *Proceedings 10th IEEE Symposium on Computer Arithmetic*, pages 64–65. IEEE Computer Society, 1991.

[92] Adam Naumowicz and Artur Korniłowicz. A brief overview of Mizar. In *International Conference on Theorem Proving in Higher Order Logics*, pages 67–72. Springer, 2009.

[93] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[94] Katsuhiko Ogata. *Discrete-time control systems*. Prentics-Hall, Inc, 1994.

[95] Takeshi Ogita, Siegfried M Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

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

[97] Sam Owre, John M Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

[98] Junkil Park, Miroslav Pajic, Insup Lee, and Oleg Sokolsky. Scalable verification of linear controller software. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 662–679. Springer Berlin, 2016.

[99] Junkil Park, Miroslav Pajic, Oleg Sokolsky, and Insup Lee. Automatic verification of finite precision implementations of linear controllers. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 153–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017.

[100] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *International Conference on Mathematical Foundations of Programming Semantics*, pages 209–228. Springer, 1989.

[101] Adnan Rashid, Umair Siddique, and Sofiène Tahar. Formal verification of cyber-physical systems using theorem proving. In Osman Hasan and Frédéric Mallet, editors, *Formal Techniques for Safety-Critical Systems*, pages 3–18, Cham, 2020. Springer International Publishing.

[102] Siegfried M Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224, 2008.

[103] Siegfried M Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate floating-point summation part II: Sign, K-fold faithful and rounding to nearest. *SIAM Journal on Scientific Computing*, 31(2):1269–1302, 2009.

[104] Umair Siddique, Mohamed Yousri Mahmoud, and Sofiene Tahar. On the formalization of Z-Transform in HOL. In *International Conference on Interactive Theorem Proving*, pages 483–498. Springer, 2014.

[105] Umair Siddique, Mohamed Yousri Mahmoud, and Sofiène Tahar. Formal Analysis of Discrete-Time Systems using z-Transform. *Journal of Applied Logic*, pages 1–32, 2018. Elsevier.

[106] Vedat Tavsanoglu and Lothar Thiele. Optimal design of state-space digital filters by simultaneous minimization of sensitivity and roundoff noise. *IEEE transactions on circuits and systems*, 31(10):884–888, 1984.

[107] Andrzej Trybulec. Non negative real numbers. Part I. *Journal of Formalized Mathematics*, 1998.

[108] Anastasia Volkova. *Towards reliable implementation of digital filters*. PhD thesis, Université Pierre et Marie Curie-Paris VI, 2017.

[109] Anastasia Volkova, Thibault Hilaire, and Christoph Lauter. Reliable evaluation of the Worst-Case Peak Gain matrix in multiple precision. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 96–103. IEEE, 2015.

[110] Anastasia Volkova, Thibault Hilaire, and Christoph Lauter. Arithmetic approaches for rigorous design of reliable Fixed-Point LTI filters. *IEEE Transactions on Computers*, 69(4):489–504, 2019.

[111] Anastasia Volkova, Matei Istoan, Florent De Dinechin, and Thibault Hilaire. Towards hardware IIR filters computing just right: Direct form I case study. *IEEE Transactions on Computers*, 68(4):597–608, 2018.

[112] Timothy E Wang, Pierre-Loïc Garoche, Pierre Roux, Romain Jobredeaux, and Éric Féron. Formal analysis of robustness at model and code level. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, pages 125–134, 2016.

[113] Bernard Widrow and István Kollár. Quantization noise. *Cambridge University Press*, 2:5, 2008.

Table des figures

Table des algorithmes

Remerciements

Je remercie avant tout Sylvie et Thibault, mes encadrants de thèse, vous qui m'avez tant apporté sur les plans scientifique et humain. C'est grâce à vous que ces années de thèse ont été une expérience à la fois plaisante et enrichissante.

Je remercie Yves Bertot et Eric Feron d'avoir accepté de rapporter ma thèse, ainsi que pour vos remarques pertinentes sur le manuscrit. Merci aussi à Jérôme Feret, Florent Hivert, Mioara Joldes, Assia Mahboubi et Jean-Michel Muller d'avoir fait partie de mon jury ; c'était un plaisir et un honneur de soutenir ma thèse devant vous.

Un grand merci à l'équipe VALS/Toccata/LMF qui m'a accueillie dans un cadre émulant et chaleureux. Merci particulièrement à Florian, mon grand frère académique, qui m'as fait bénéficier de ton expérience pour toutes les étapes de la thèse, de l'entretien préliminaire avec l'ED à la soutenance. Merci à Kim, Christine, Sylvain et tous les autres pour qui j'ai eu la chance d'enseigner. Merci aussi à Robin, Quentin, Rébecca, Alexandrina, Antoine, Kostia, Glen, Mattias, Hai, Mário, Martin, Benedikt, Claude, Guillaume, Jean-Christophe, Andrei, Véronique, Fred, Chantal, Katia, et tous les autres (il manque beaucoup de noms à cette liste, mais même si je la continue il en manquera encore, donc j'espère que vous me pardonnerez). Merci à vous tous pour votre amitié, votre aide et les conversations du vendredi tous les jours.

Merci à Anastasia et Clothilde pour votre bonne humeur, et votre aide avec le premier exposé de cette thèse. Merci aux autres membres de l'équipe Pequan, et à tous ceux avec qui j'ai eu la chance d'échanger au cours de conférences et autres événements. Merci aussi aux encadrants de mes divers stages de recherche, et aux nombreux professeurs qui ont marqué ma scolarité.

« Ce travail a bénéficié du soutien financier de la "Fondation CFM pour la Recherche". » Autrement dit, c'est celle-ci qui a fourni ma bourse de thèse, et je la remercie pour sa générosité et son efficacité.

Merci à tous mes amis ; vous vous reconnaîtrez (j'ai déjà tenté d'écrire une liste de trop plus haut).

Enfin, je remercie ma famille, qui m'a toujours soutenue de toutes les manières possibles. Merci Maman, la première à m'avoir transmis le goût des maths, Papa, Victor, Amatxi, Ailli, mes tantes et mes oncles. Et merci Raphaël, pour tout.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Formalisation en Coq des algorithmes de filtre numérique calculés en précision finie
Mots clés : méthodes formelles, assistant de preuve Coq, traitement du signal, filtre numérique linéaire, arithmétique des ordinateurs, arithmétique en virgule fixe

Résumé : Les filtres numériques sont utilisés dans de nombreux domaines, des télécommunications à l'aérospatiale. En pratique, ces filtres sont calculés sur machine en précision finie (virgule flottante ou virgule fixe). Les erreurs d'arrondi résultantes peuvent être particulièrement problématiques dans les systèmes embarqués. En effet, de fortes contraintes énergétiques et spatiales peuvent amener à privilégier l'efficacité des calculs, souvent au détriment de leur précision. De plus, les algorithmes de filtres enchaînent de nombreux calculs, au cours desquels les erreurs d'arrondi se propagent et risquent de s'accumuler.

Comme certains domaines d'application sont critiques, j'analyse les erreurs d'arrondi dans les algorithmes de filtre en utilisant l'assistant de preuve Coq. Il s'agit d'un logiciel qui garantit formellement que cette analyse est correcte. Un premier objectif est d'obtenir des bornes certifiées sur la différence entre les valeurs produites par un filtre implémenté (calculé en précision finie) et par le filtre modèle initial (défini par des calculs théoriques exacts). Un second objectif est de garantir l'absence de comportement catastrophique comme un dépassement de capacité supérieur imprévu.

Je définis en Coq les filtres numériques linéaires invariants dans le temps (LTI), considérés dans le domaine temporel. Je formalise une forme universelle appelée la SIF, à laquelle on peut ramener n'importe quel algorithme de filtre LTI sans modifier ses propriétés numériques. Je prouve ensuite le théorème des filtres d'erreurs et le théorème du *Worst-Case Peak Gain*, qui sont deux théorèmes essentiels à l'analyse numérique d'un filtre sous forme de SIF. Par ailleurs, cette analyse dépend aussi de l'algorithme de somme de produits utilisé dans l'implémentation. Je formalise donc plusieurs algorithmes de somme de produits offrant différents compromis entre précision du résultat et vitesse de calcul, dont un algorithme original correctement arrondi au plus proche. Je définis également en Coq les dépassements de capacité supérieurs modulaires, afin de prouver la correction d'un de ces algorithmes même en présence de tels dépassements de capacité.

Title: Coq formalization of digital filter algorithms computed using finite precision arithmetic
Keywords: formal methods, Coq proof assistant, signal processing, linear digital filter, computer arithmetic, fixed-point arithmetic

Abstract: Digital filters have numerous applications, from telecommunications to aerospace. To be used in practice, a filter needs to be implemented using finite precision (floating- or fixed-point arithmetic). Resulting rounding errors may become especially problematic in embedded systems: tight time, space, and energy constraints mean that we often need to cut into the precision of computations, in order to improve their efficiency. Moreover, digital filter programs are strongly iterative: rounding errors may propagate and accumulate through many successive iterations.

As some of the application domains are critical, I study rounding errors in digital filter algorithms using formal methods to provide stronger guaranties. More specifically, I use Coq, a proof assistant that ensures the correctness of this numerical behavior analysis. I aim at providing certified error bounds over the difference between outputs from an implemented filter (computed using finite precision) and from the original model filter (theoretically defined with exact operations). Another goal is to guarantee that no catastrophic behavior (such as unexpected overflows) will occur.

Using Coq, I define linear time-invariant (LTI) digital filters in time domain. I formalize a universal form called SIF: any LTI filter algorithm may be expressed as a SIF while retaining its numerical behavior. I then prove the error filters theorem and the Worst-Case Peak Gain theorem. These two theorems allow us to analyze the numerical behavior of the filter described by a given SIF. This analysis also involves the sum-of-products algorithm used during the computation of the filter. Therefore, I formalize several sum-of-products algorithms, that offer various trade-offs between output precision and computation speed. This includes a new algorithm whose output is correctly rounded-to-nearest. I also formalize modular overflows, and prove that one of the previous sum-of-products algorithms remains correct even when such overflows are taken into account.

Maison du doctorat de l'Université Paris-Saclay
2ème étage aile ouest, Ecole normale supérieure Paris-Saclay
4 avenue des Sciences
91190 Gif-sur-Yvette, France