A Coq formalization of digital filters

Diane Gallois-Wong, Sylvie Boldo, Thibault Hilaire

A Coq formalization of digital filters

Diane Gallois-Wong[1,2⋆], Sylvie Boldo[3,2], Thibault Hilaire[3,2,4]

[1] Université Paris-Sud
[2] LRI, CNRS & Univ. Paris-Sud, Université Paris-Saclay, bâtiment 650, Université Paris-Sud, F-91405 Orsay Cedex, France
[3] Inria
[4] Sorbonne Université, F-75005 Paris, France

Abstract. Digital filters are small iterative algorithms, used as basic bricks in signal processing (filters) and control theory (controllers). They receive as input a stream of values, and output another stream of values, computed from their internal states and from the previous inputs. These systems can be found in communication, aeronautics, automotive, robotics, etc. As the application domain may be critical, we aim at providing a formal guarantee of the good behavior of these algorithms. In particular, we formally proved in Coq some error analysis theorems about digital filters, namely the Worst-Case Peak Gain theorem and the existence of a filter characterizing the difference between the exact filter and the implemented one. Moreover, the digital signal processing literature provides us with many equivalent algorithms, called realizations. We formally defined and proved the equivalence of several realizations (Direct Forms and State-Space).

1 Introduction

Most embedded systems, from planes to MP3 players, rely on numerical signal processing filters. Such a filter takes as inputs an infinite sequence of values, such as measurements of sensors, and returns an infinite sequence of values, such as a sound to be played or a value to control a nuclear power plant (see also §2.2). Applications of digital filters are therefore numerous, from trivial to life-critical, and formal methods have already been applied to such systems. A similar work has been done in HOL [1]: the authors define filters and somehow define the error filter as done in §5.1. But then, they do not bound this error, while we do it using the Worst-Case Peak Gain (WCPG) Theorem of §5.2. We indeed benefit from the most recent advances for filter error bounding as in [2]. A previous work is about the comparison between floating-point and fixed-point implementations of digital filters to ensure their similar behavior [3]. But this work requires many hypotheses, including the absence of overflow in the computations. Another formal work aims at finding specifications of filters and then proving their correctness [4], assuming that all the computations are exact. They then relax this assumption, but bound the floating-point error (they

⋆ This work is supported by a grant from the "Fondation CFM pour la Recherche".

do not consider fixed-point) for one iteration step only, while the difficult part is the propagation of these errors [5]. Last, abstract interpretation has gotten interested in digital filters. A specific abstract domain has been developed [6]. A more generic work with a focus on the compositional analysis of filters has been done to bound all the variables, with a use of the Worst-Case Peak Gain (WCPG) Theorem of §5.2, but without any formal proof. Finally, references in digital signal processing can be found in §2.

This work aims at formalizing digital filters in the Coq proof assistant [7,8]. The goal is to have a formal definition of what is a filter and what it is supposed to do. The interest is twofold. First, it gives us a mathematical reference for a correct ideal filter, in order to compare with our practical filters. Second, it will serve as base to formalize and prove the behavior of real filters that compute using fixed-point or floating-point arithmetic. But this work requires a clean definition of exact filters and of the possible algorithms a filter may be implemented with. Indeed, there are many ways to represent a digital filter, and many (mathematically) equivalent algorithms for computing the result of applying a digital filter. This work presents several equivalent algorithms, called realizations, that implement this filter and that are used in signal processing (such as the direct forms I and II, the State-Space realization) and their equivalences and builds upon the Coquelicot library [9].

This paper is organized as follows. Section §2 gives some background on signal processing and numerical filters. Section §3 presents the formalization choices. Section §4 presents the various formalized realizations. Section §5 presents some theorems done (also) to test the practicality of the formalization. Section §6 concludes and gives some perspectives.

Notation: throughout the article matrices are in uppercase boldface (e.g. $\boldsymbol{A}$), vectors are in lowercase boldface (e.g. $\boldsymbol{a}$), scalars are in lowercase (e.g. a). The matrix $\boldsymbol{I}_n$ is the identity matrix of size n.

2 Digital filters

2.1 Signals and operations

A discrete-time signal x is an ordered sequence of numbers denoted $x(k)$, where k is an integer. In practical setting, such sequence can often arise from periodic sampling of continuous time signal $x_c(t)$ where t represents time.

The signal x can be defined for any integer k or for some finite continuous set of integers. We here restrict k to be in $\mathbb{N}$, or more precisely k to be in $\mathbb{Z}$, but with $x(k) = 0$ for $k < 0$. A signal can be real ($x(k) \in \mathbb{R}$) or vector ($\boldsymbol{x}(k) \in \mathbb{R}^{p\times 1}$).

The simplest signal is probably the *impulse* signal (also called *Dirac* signal), denoted δ and defined as

$$\delta(k) = \begin{cases} 1 & \text{if } k = 0 \\ 0 & \text{elsewhere.} \end{cases} \tag{1}$$

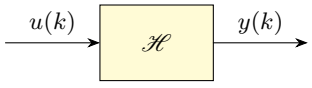
This signal is central in linear signal processing theory since any signal can be expressed as a linear combination of suitably weighted and shifted impulse signals (see §2.3).

Three elementary operations on signals can be defined:

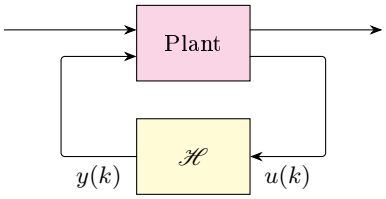
- Addition: The sum of two signals x_1 and x_2 is their term-by-term sum, i.e. $y = x_1 + x_2$ means $\forall k,\ y(k) = x_1(k) + x_2(k)$.
- Scaling: $y = \alpha x$ is the sequence x scaled by $\alpha \in \mathbb{R}$, i.e. $\forall k,\ y(k) = \alpha x(k)$.
- Time shifting: Let y be the sequence x shifted in time by an integer K, i.e. $\forall k,\ y(k) = \begin{cases} x(k-K) & \text{if } k \geq K \\ 0 & \text{elsewhere.} \end{cases}$

2.2 Linear Time Invariant filters

A filter $\mathscr{H}$ is a mathematical transformation (or operator) that maps an input signal u into an output signal $y = \mathscr{H}\{u\}$, as shown in Figure 1a. At each time k, the filter produces an output $y(k)$ from the input $u(k)$. But, contrary to mathematical functions, the output $y(k)$ depends on the input at time k, but also on the internal state of the filter (i.e. depends on the initial condition of the filter and the inputs).



(a) Filter in open-loop context.



(b) Filter in closed-loop context (controller)

Fig. 1: A filter used in open-loop and closed-loop contexts.

A linear time invariant (LTI) filter satisfies the two following properties:

- Linearity: The filter is linear with respect to its input, i.e.

$$\mathscr{H}\{\alpha u_1 + \beta u_2\} = \alpha \mathscr{H}\{u_1\} + \beta \mathscr{H}\{u_2\} \tag{2}$$

- Shift invariance: if the input of the filter is delayed by K samples, then the output is also delayed by K samples, i.e. if $u_1(k) = u_2(k-K)$, then

$$\forall k, \quad \mathscr{H}\{u_1\}(k) = \begin{cases} \mathscr{H}\{u_2\}(k-K) & \text{if } k \geq K \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

The filter can have as inputs and outputs either scalars (Single-Input Single-Output filter, aka SISO filter), or vectors (Multiple-Input Multiple-Output filter, aka MIMO filter). Moreover, due to the linearity of LTI filters, a q-input p-output

MIMO filter can be seen as the assembly of $p \times q$ SISO filters, where the ij^{th} element is the filter that captures the effect of the j^{th} input on the i^{th} output.

LTI filters are used in signal processing, but also in control theory. In that context, they are called LTI controllers. Some outputs of a plant to be controlled are measured and used as input of a linear filter (denoted *controller*). This filter will control the plant since its outputs are used as input of the plant, see Figure 1b. Contrary to signal processing context, a closed-loop involving plant and controller is to be considered.

LTI systems are composition of the three elementary operations on signals (multiplication by constant, addition or subtraction, and delay). Data-flow graphs using these operations as blocks and signal as streams are widely used in signal and control theory to describe the different realizations, as shown in Figures 3a, 3b, and 4a.

2.3 Impulse response

The impulse response of a SISO filter $\mathscr{H}$, denoted h, is the answer (output) of the filter to an impulse input δ (i.e. $h = \mathscr{H}\{\delta\}$).

Since any input u can be written as a sum of weighted shifted impulses:

$$u(k) = \sum_{l \in \mathbb{Z}} u(l)\delta(k-l) \tag{4}$$

then, by linearity of the considered filter, the output y of u through $\mathscr{H}$ can be obtained by

$$y(k) = \mathscr{H}\{u\}(k) = \sum_{l \in \mathbb{Z}} u(l)h(k-l) \tag{5}$$

The output y is obtained as a convolution of signals u and h. For that reason, the impulse response of a filter fully defines it.

From the impulse response characterization, the LTI filters can be divided in two types: the Finite Impulse Response (FIR) filters, where h is null above a certain time k, and the Impulse Finite Response (IIR) filters, where h has infinite support [10].

2.4 Constant-Coefficient Difference Equation

An important subclass of LTI systems consists of those for which a n-order constant-coefficient difference equation exists between inputs and outputs, i.e. the last n inputs and outputs are linked by:

$$\sum_{i=0}^{n} a_i y(k-i) = \sum_{i=0}^{n} b_i u(k-i), \quad \forall k \tag{6}$$

where the $\{a_i\}_{1 \leq i \leq n}$ and $\{b_i\}_{1 \leq i \leq n}$ are constant coefficients.

We also assume that $a_0 = 1$, so that the equation is rearranged as

$$y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=1}^{n} a_i y(k-i). \tag{7}$$

This relationship describes an IIR filter as soon as the a_i's coefficients are not null, otherwise it describes a FIR system.

The value n is said to be the order of the system. These coefficients are also the coefficients of the *transfer function* of the filter: this mathematical object describes its input-output relationship in frequency domain (whereas (7) describes it in time domain). It has also been formalized in Coq, but it is not used yet, and thus not presented here.

3 Formalization

3.1 Signals and filters

The first components required to work with digital filters are signals, presented in §2.1. For now, we consider real signals (sequences of real numbers). We define them as functions from $\mathbb{Z}$ to $\mathbb{R}$ that take the value 0 for every $k < 0$.

```
Definition causal (x : Z → R) := (forall k : Z, (k < 0)%Z → x k = 0%R).
Record signal := { signal_val :> Z → R ; signal_prop : causal signal_val }.
```

We rely on two axioms: `FunctionalExtensionality` and `ProofIrrelevance` to ensure that signals are fully characterised by their values for $k \geq 0$.

This definition of signals makes them easy to build recursively. Notably, order-n recursion (when $x(k)$ is built from $x(k-1), x(k-2), ..., x(k-n)$) is very common when working with filters: for example, (7) is an order-n recursive relation on y. Order-n recursion over $\mathbb{N}$ requires initial values for 0, 1, ..., $n-1$. For signals however, the negative values are known (since they are zero), so we may use an order-n recursion at even for 0 or 1 (until $n-1$). See §3.2 for more details on recursive constructions.

This explains why we work with relative indexes even if we could have defined them as functions from $\mathbb{N}$ to $\mathbb{R}$. Ironically, working with $\mathbb{N}$ instead of $\mathbb{Z}$ would bring back problems of initialization. We would need to be especially careful as trying to use the recursive relation for 0 would mean we are using the values for $0-1$, $0-2$, ..., $0-n$ which are all equal to 0 when using subtraction over $\mathbb{N}$ in Coq, and this would make no sense. We have considered several solutions to this problem that would have allowed us to work with $\mathbb{N}$, but we have discarded them: they either complicated proofs by requiring to handle additional non trivial cases, or used alternative notations that made theorems more difficult to read and compare to their usual signal processing formulation. Ultimately, we felt that the readability and more intuitive subtraction offered by $\mathbb{Z}$ were worth needing to adapt a few libraries to relative indexes (§3.3).

We define the three elementary operations on signals described in §2.1, namely addition, scaling by a real and time shift and the proof they provide a signal

from a signal. The only interesting point is in the time shift by an integer K (which to x associates y such that $\forall k,\ y(k) = x(k-K)$), as the result may not be a signal when $K < 0$. But it is reasonable to assume that $K \geq 0$, as it makes sense to delay a signal, but not to know in advance the future inputs of a stream of values. Therefore, we define `delay (K : Z) (x : signal) : signal` as the usual time shift of `x` by `K` when $\texttt{K} \geq 0$, and arbitrarily as the null signal when $\texttt{K} < 0$ so that the function is total. Note that this restriction does not change the shift invariance property from the definition of LTI filters.

We also define the type `vect_signal` of vector signals, that are functions from $\mathbb{Z}$ to $\mathbb{R}^p$ that return the null vector for every $k < 0$, where $p \in \mathbb{Z}$ is an implicit parameter that we will assume to be positive whenever we need it.

Once we have signals, filters are simply defined as functions from signals to signals: `Definition filter := signal -> signal`. From the three elementary operations on signals defined above (addition, scaling by a real and time shifting), we define compatibility of a filter with addition, compatibility with scalar multiplication and shift invariance. Finally, we define `LTI_filter : filter -> Prop` (linear time invariant filter) as the logical "and" of these three properties (see §2.2 and note that our first two properties are the two components of linearity).

As explained in §2.2, a filter can be SISO (Single-Input Single-Output) when it handles real signals as in the definition above, or MIMO (Multiple-Input Multiple-Output) with vector signals. MIMO filters are also easy to define: `Definition MIMO_filter {N_in N_out : Z} := @vect_signal N_in -> @vect_signal N_out.` However, since a p-input q-output MIMO filter can been as the composition of $p * q$ SISO filters, one for the contribution of each, it is sufficient to focus on SISO filters. Most our theorems are therefore dedicated to SISO filters, as in textbooks. Proving in Coq these results in MIMO from the SISO ones would not be difficult, only technical with many conversions between vectors with a single non zero coefficient and reals. Moreover, it would make the theorems less readable. Nevertheless, we define the State-Space characterization of a filter, presented in §4.3, for MIMO filters, as we explicitly need this to study error propagation in §5.1.

3.2 Recursion over $\mathbb{Z}$

We define several inductions and recursive constructions over $\mathbb{Z}$. We solve the problem of $\mathbb{Z}$ having no initial point simply by initializing at least over every $k < 0$. This means that our recursions are characterized by their behavior for $k \geq 0$, but having values for $k < 0$ as well will make them easier to use for signals. Our inductions `Z_peano_ind` and `Z_strong_ind` are similar to respectively `natlike_ind` and `Zlt_0_ind` from the standard library (`Coq.ZArith.Wf_Z`), except for the initialization over negative indexes where the latter prove the property only for non negative indexes. Our recursive constructions `Z_peano_rec` and `Z_strong_rec` are not as close to standard-library `natlike_rec` and `Zlt_0_rec`. They are actual functions from $\mathbb{Z}$ to an implicit type T, whereas the latter are lemmas that prove that such functions can be constructed.

Concerning the usual recursion, `Z_peano_ind` takes an initialization of the property for any $k \leq 0$, then an induction step from k to $k+1$ when $k \geq 0$, to ensure that the property holds for all $k \in \mathbb{Z}$. `Z_peano_rec` takes an initialization function `f_le_0` that will only be used for $k \leq 0$, and a step function `f_succ` that computes the image of $k+1$ for $k \geq 0$, depending on the image of k. It builds the function $f : \mathbb{Z} \to T$ such that

$$\begin{cases} \forall k \leq 0, f(k) = \texttt{f\_le\_0}(k) \\ \forall k \geq 0, f(k+1) = \texttt{f\_succ}(k+1)(f(k)) \end{cases} \tag{8}$$

```
Lemma Z_peano_ind (P : Z -> Prop) : (forall k, (k <= 0)%Z -> P k) ->
  (forall k, (k >= 0)%Z -> P k -> P (k + 1)) -> forall k, P k.
Definition Z_peano_rec (f_le_0 : Z -> T) (f_succ : Z -> T -> T) (k : Z) : T := ...
```

We prove (8) in two lemmas that serve as an interface so that outside of their own proofs, the definition of `Z_peano_rec` is never expanded.

Concerning the strong recursion, `Z_strong_ind` takes an initialization of the property for any $k < 0$, then an induction step from all j such that $j < k$ to k when $k \geq 0$, to ensure that the property holds for all $k \in \mathbb{Z}$. `Z_strong_rec` takes an initialization function `f_lt_0` that will only be used for $k < 0$, and a step function `f_rec` that computes the image of k for $k \geq 0$, depending on the image of j for any $j < k$. If we note $f|_{<k}$ the restriction of f to $(-\infty, k)$, `Z_strong_rec` builds the function $f : \mathbb{Z} \to T$ such that

$$\begin{cases} \forall k < 0, f(k) = \texttt{f\_lt\_0}(k) \\ \forall k \geq 0, f(k) = \texttt{f\_rec}(k)(f|_{<k}) \end{cases} \tag{9}$$

```
Lemma Z_strong_ind : forall P : Z -> Prop, (forall k, k < 0 -> P k)%Z ->
  (forall k, k >= 0 -> (forall j, j < k -> P j) -> P k)%Z -> forall k, P k.
Definition Z_strong_rec (f_lt_0 : Z -> T) (f_rec : Z -> (Z -> T) -> T) (k : Z) : T := ...
```

But, as we choose to work with total functions, the second argument of `f_rec` is total. When we write $f|_{<k}$, we actually mean that it is equal to previously computed values of f for $j < k$, and to default values for $j \geq k$. As `f_rec` is just an argument of `Z_strong_rec`, it could evaluate $f|_{<k}$ for $j \geq k$, which would make no sense from a recursion perspective. To have a proper recursive construction, the argument `f_rec` needs to verify the property `f_rec_well_formed`, which means that when we call `f_rec (k : Z) (g : Z -> T)` with $k \geq 0$, the result does not depend on the values of `g` for $j \geq k$. Lemmas `Z_strong_rec_lt_0` and `Z_strong_rec_ge_0` express (9), the second one needing the hypothesis that the argument `f_rec` is well-formed. Once again, they serve as an interface so that outside of their own proofs, the 13-line definition is never expanded.

```
Definition f_rec_well_formed (f_rec : Z -> (Z -> T) -> T) : Prop :=
  forall (k : Z) (g1 g2 : Z -> T), (k >= 0)%Z ->
  (forall j : Z, (j < k)%Z -> g1 j = g2 j) -> f_rec k g1 = f_rec k g2.
Lemma Z_strong_rec_ge_0 f_lt_0 f_rec k : (k >= 0)%Z -> f_rec_well_formed f_rec ->
  Z_strong_rec f_lt_0 f_rec k = f_rec k (Z_strong_rec f_lt_0 f_rec).
```

Defining signals only requires order-n recursion (§3.1). However, our stronger recursion is easier to use.

For example, consider a signal x defined by the recursive relation $x(k) = x(k-1) + x(k-2)$. We define the function representing this relation: `f_rec := fun (k : Z) (g : Z -> R) => (g(k-1)%Z + g(k-2)%Z)%R`, and we can prove that it is well-formed. As a signal should be zero for $k < 0$, the initialization function is `f_lt_0 := fun (k : Z) => 0%R`. Then, `(Z_strong_rec f_lt_0 f_rec) : Z -> R` is the function that corresponds to our signal x.

3.3 Adapting existing libraries to relative indexes

Unsurprisingly, we need many lemmas about the sum of consecutive terms of a sequence. There exists such a function in the Coquelicot library [9], but with natural indexes. We need relative indexes, so we had to build upon Coquelicot to define `sum {G : AbelianGroup} (low up : Z) (x : Z -> G) : G` and all the needed theorems. Its features are that bounds are included, `sum low up x` is equal to `x low + x (low+1) + ... + x up`, and that `sum low up x` is zero when `low > up`.

Similarly, we need matrices with relative indexes. Building upon matrices with natural indexes from Coquelicot, we denote by `mtx (h w : Z)` a matrix of size `h` × `w` with coefficients in an implicit abelian group or ring G. Indexes start at 1 (relevant indexes for the matrix above are $1 \leq i \leq$ `h` and $1 \leq j \leq$ `w`). We also provide two functions to serve as an interface: the first one creates a matrix from a $\mathbb{Z} \to \mathbb{Z} \to G$ function (using only its values for relevant indexes), and the second one gets the coefficient (i, j) of a matrix (it is total with arguments i and j in $\mathbb{Z}$, and it returns default values if the indexes are outside of the bounds). Thanks to these interface functions, the actual definition of matrices does not matter, so we can easily switch to other existing libraries. In particular, we may use the Mathcomp library [**?**] to rely on its linear algebra theorems for evaluating the WCPG defined in §5.2.

4 Filter realizations

To be implemented, an associated algorithm should be made explicit: it produces at each time step k an output $y(k)$ from the input $u(k)$ and its internal states (containing information about all the previous inputs).
In the literature, a lot of algorithms exist to implement a linear filter [10]: direct forms, State-Space realizations, Second-Order Sections, cascade or parallel decomposition, Lattice Wave Digital filters [11,12], δ- or ρ-operator based filters [13,14,15], etc. Each of them presents some advantages with respect to the number of coefficients, the software or hardware adequacy, the finite-precision behavior, etc. and the choice of the realization is itself a research domain [17].

We present here three classical realizations: the Direct Form I (that comes from the constant-coefficients difference equation (7)), the Direct Form II (that uses the same coefficients in another algorithm) and the State-Space realizations. In practice, these realizations are described by a data-flow graph, that describes how data (signals) are processed by a system in terms of inputs and outputs.

4.1 Direct Form I

Direct Form I (DFI) comes directly from the constant-coefficient difference equation presented in §2.4: $y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=1}^{n} a_i y(k-i)$. It depends on the $2n+1$ transfer-function coefficients $a_1, a_2, ..., a_n, b_0, b_1, ..., b_n$.

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

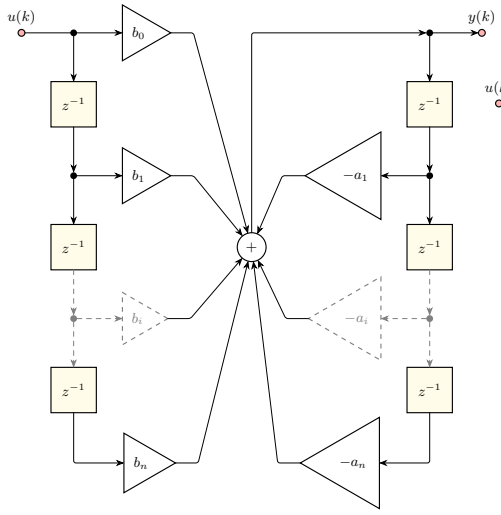
(a) Direct Form I

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

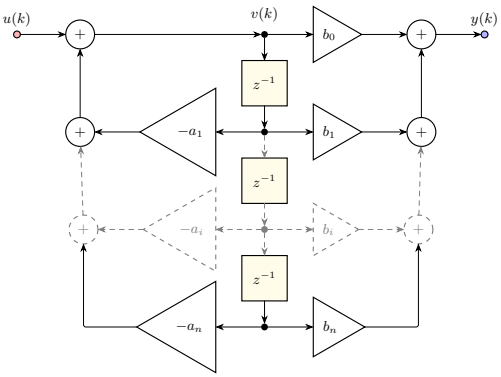
(b) Direct Form II

Fig. 2: Direct Form I and II algorithms.

The corresponding algorithm is presented in Algorithm 2a and the data flow graph is in Figure 3a. The real program is slightly more complex: the n previous values of u and y are stored in memory (the yellow squares in the data flow graph 3a represent *delay*, each one is a memory element).



(a) Direct Form I



(b) Direct Form II

Fig. 3: Direct Form I and II data-flow graphs.

In Coq, we define a type `TFCoeffs` for the transfer function coefficients. They are given as an order n and two sequences $a, b : \mathbb{Z} \to \mathbb{R}$. In practice, n should be

positive, and we will only use the values $a(i)$ for $1 \leq i \leq n$ and the values $b(i)$ for $0 \leq i \leq n$. So far, we have not needed to enforce this in the definition.

```
Record TFCoeffs := { TFC_n : Z ; TFC_a : Z -> R ; TFC_b : Z -> R }.
```

The main component needed for our recursive construction of a signal x is a function, noted `f_rec` in our definition of strong recursion (§3.2), that builds $x(k)$ for $k \geq 0$ given all previous values of x. `(DFI_f_rec tfc u) : Z -> (Z -> R) -> R` is such a function: it expresses the recursive relation characterizing the output signal for input `u` of the filter defined by Direct Form I with the coefficients `tfc`.

From this, we define the function that builds a filter from its transfer function coefficients using Direct Form I, called `filter_from_TFC` because Direct Form I is the canonical way to build a filter from these coefficients.

```
Definition DFI_f_rec (tfc : TFCoeffs) (u : signal) :=
  (fun (n : Z) (y_before_n : Z -> R) =>
      (sum 0 (TFC_n tfc) (fun i => (TFC_b tfc i) * u (n-i)%Z)%R)
    - (sum 1 (TFC_n tfc) (fun i => (TFC_a tfc i) * y_before_n (n-i)%Z)%R)).
Definition filter_from_TFC (tfc : TFCoeffs) : filter :=
  fun (u : signal) => build_signal_rec (DFI_f_rec tfc u).
```

Finally, we prove that a filter built this way is LTI. This proof presents no real difficulty once we have adequate lemmas about sum of consecutive value of a sequence and recursive building of signal.

4.2 Direct Form II

Direct Form II is quite similar to Direct Form I. It uses the same coefficients, those from the transfer function. The main difference is that it only requires n delays (instead of $2n$), so is more efficient to implement. It can be described by the data flow graph of Figure 3b, or by Algorithm 2b.

Where Direct Form I builds the output y from previous values of both y and the input u, Direct Form II builds an intermediary signal e from previous values of itself and only the current value of u, then it builds y from previous values of e. The Coq definitions reflect this. As the construction of e is recursive, we define `DFII_u2e_f_rec tfc u`, which is the main function allowing to build e given the transfer function coefficients and an input signal.

Combining the constructions of the intermediary signal e and of the output signal y, we define `filter_by_DFII` which builds a filter from transfer function coefficients using Direct Form II.

```
Definition DFII_u2e_f_rec (tfc : TFCoeffs) (u : signal) :=
  fun (n : Z) (e_before_n : Z -> R) =>
    u n - sum 1 (TFC_n tfc) (fun i => (TFC_a tfc i) * e_before_n (n-i)%Z).
Definition DFII_e2y (tfc : TFCoeffs) (e : signal) := Build_signal
  (fun n => sum 0 (TFC_n tfc) (fun i => (TFC_b tfc i) * e (n-i)%Z)%R)
  (DFII_e2y_prop tfc e).
Definition filter_by_DFII (tfc : TFCoeffs) : filter :=
  fun (u : signal) => DFII_e2y tfc (build_signal_rec (DFII_u2e_f_rec tfc u)).
```

Finally, as Direct Form I and Direct Form II use the same coefficients, which are also the coefficients associated to the transfer function, implementing either of these algorithms with the same set of coefficients should produce the same filter, meaning the same output even if the algorithms are different.

```
Theorem DFI_DFII_same_filter (tfc : TFCoeffs) : filter_from_TFC tfc = filter_by_DFII tfc.
```

Since a filter built using `filter_from_TFC` (which is by definition built using Direct Form I) is LTI, so is a filter built using Direct Form II.

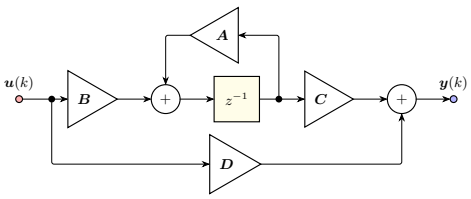
4.3 State-Space

Until now, we were considering SISO filters, but here we provide definitions for both MIMO filters (because this will be needed for error analysis in §5.1) and SISO filters (to link with previous results).

For a p-input q-output MIMO filter $\mathscr{H}$, the State-Space representation is described by four matrices $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$. The corresponding algorithm is:

$$\begin{cases} \boldsymbol{x}(k+1) = \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}\boldsymbol{u}(k) \\ \quad \boldsymbol{y}(k) = \boldsymbol{C}\boldsymbol{x}(k) + \boldsymbol{D}\boldsymbol{u}(k) \end{cases} \tag{10}$$

where $\boldsymbol{x}(k) \in \mathbb{R}^{n\times 1}$, $\boldsymbol{u} \in \mathbb{R}^{q\times 1}$ and $\boldsymbol{y}(k) \in \mathbb{R}^{p\times 1}$ are the state vector, input vector and output vector, respectively. The matrices $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, $\boldsymbol{B} \in \mathbb{R}^{n\times q}$, $\boldsymbol{C} \in \mathbb{R}^{p\times n}$ and $\boldsymbol{D} \in \mathbb{R}^{p\times q}$ determined the system $\mathscr{H}$. Figure 4a exhibits its data-flow graph, and Algorithm 4b its algorithm (scalar rewriting of (10)).



(a) Data-flow graph.

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

(b) Algorithm

Fig. 4: State-Space data-flow graph and algorithm.

In Coq, we define a State-Space as a record containing the size of the state vector and the four matrices.

```
Context { N_in N_out : Z }.
Record StateSpace := { StSp_n : Z ;
                       StSp_A : @mtx R_Ring StSp_n StSp_n ;
                       StSp_B : @mtx R_Ring StSp_n N_in ;
                       StSp_C : @mtx R_Ring N_out StSp_n ;
                       StSp_D : @mtx R_Ring N_out N_in       }.
```

For a given State-Space and a given input vector signal u, we first define the vector signal of state vectors x by recursion, then the input vector signal y from u and x, following closely the relationship in (10). We obtain a function that builds the MIMO filter corresponding to a State-Space. We also define a SISO State-Space as a State-Space where the context variables `N_in` and `N_out` are both 1. Then we define `filter_from_StSp : SISO_StateSpace -> filter` that associates a SISO filter to such a State-Space. We also prove that a filter built from a State-Space is always LTI.

The impulse response h (introduced in §2.3) of a filter defined by a State-Space can be computed from the matrices of the State-Space with:

$$h(k) = \begin{cases} \boldsymbol{D} & \text{if } k = 0 \\ \boldsymbol{C}\boldsymbol{A}^{k-1}\boldsymbol{B} & \text{if } k > 0 \end{cases} \tag{11}$$

Moreover, in SISO, a State-Space can also be built from the transfer function coefficients. Below is one of many ways to build such a State-Space, requiring $(n+1)^2$ coefficients for the four matrices combined (note that $\boldsymbol{C}$ is a single-line matrix, $\boldsymbol{B}$ single-column, and $\boldsymbol{D}$ single-line single-column):

$$\boldsymbol{A} = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \dots & \dots & 0 & 1 \\ -a_n & \dots & \dots & -a_2 & -a_1 \end{pmatrix}, \quad \boldsymbol{B} = \begin{pmatrix} 0 \\ \vdots \\ \vdots \\ 0 \\ 1 \end{pmatrix} \tag{12}$$

$$\boldsymbol{C} = \begin{pmatrix} b_n - a_n b_0 & \dots & \dots & b_1 - a_1 b_0 \end{pmatrix}, \quad \boldsymbol{D} = \begin{pmatrix} b_0 \end{pmatrix} \tag{13}$$

```
Notation n := (TFC_n tfc).
Definition SISO_StSp_from_TFC : SISO_StateSpace := @Build_StateSpace 1 1
  (*StSp_n*) n
  (*StSp_A*) (make_mtx n n (fun i j =>
                if Z_eq_dec i n then (- (TFC_a tfc (n+1-j)))
                else if Z_eq_dec (i+1) j then 1 else 0))
  (*StSp_B*) (make_mtx n 1 (fun i j => if Z_eq_dec i n then 1 else 0))
  (*StSp_C*) (make_mtx 1 n (fun i j =>
                TFC_b tfc (n+1-j) - TFC_a tfc (n+1-j) * TFC_b tfc 0))
  (*StSp_D*) (mtx_of_K (TFC_b tfc 0)).
Theorem StSp_TFC_same_filter (tfc : TFCoeffs) : (TFC_n tfc >= 0)%Z ->
  filter_from_StSp (SISO_StSp_from_TFC tfc) = filter_from_TFC tfc.
```

We define a Coq function that associates a (Single Input Single Output) State-Space to transfer function coefficients, by simply following the construction above. More importantly, we prove that the filter built from such a State-Space is the same as the filter obtained directly from the transfer function coefficients. As we have seen previously, there are at least two ways to build a filter directly from these coefficients: Direct From I and Direct From II. Although Direct Form I is the canonical one, it is much easier to use Direct Form II in this proof. The

main step is to prove that for any k, the state vector $\boldsymbol{x}(k)$ contains $e(k-n)$ as its first coefficient, $e(k-(n-1))$ as its second one, etc. up to $e(k-1)$ as its n-nth coefficient, where e is the auxiliary signal that appears in Direct Form II (Figure 2b). This is done by strong induction, with trivial initialization for $k<0$ since everything is zero. For $k \geq 0$, the *ones* just above the diagonal in $\boldsymbol{A}$ and the *zeros* in $\boldsymbol{B}$ mean that the coefficients 1 to $n-1$ of $\boldsymbol{x}(k)$ are the coefficients 2 to n of $\boldsymbol{x}(k-1)$, so the only point left to prove is that $\boldsymbol{x}(k)_n = e(k-1)$. This is ensured by the last line of $\boldsymbol{A}$ and the last coefficient of $\boldsymbol{B}$, which make the matrix operations in (10) unfold into exactly the computation of $e(k)$ in Figure 2b.

Note that the matrices $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ are not uniquely defined for a given filter: distinct State-Spaces describe distinct algorithms, but they may build the same filter, that it the same function from signals to signals. So it is possible to search for the *optimal* State-Space when considering the effect of the finite precision degradations [18].

5 Error analysis

5.1 Error filter

Using finite precision arithmetic (floating- or fixed-point arithmetic), some arithmetic operations (addition or multiplication) may introduce errors (mainly because the number of bits to represent the values is finite, and may be not enough to represent the exact result). A quantization can be modelled as the addition of an extra term, called *roundoff error*.

We consider a State-Space $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$. Note that this is the most general of the realizations that we have presented: we have proven that from the other realizations we can build a State-Space so that the filter defined is the same.

At each step k, the evaluation of the states and outputs (see Algorithm 4b) is composed of sum-of-products (SoP), one per state and output. Since they are not performed in exact arithmetic, each SoP may produce an error, compared to the exact sum. So (10) is changed to

$$\begin{aligned} \boldsymbol{x}^*(k+1) &\leftarrow \boldsymbol{A}\boldsymbol{x}^*(k) + \boldsymbol{B}\boldsymbol{u}(k) + \boldsymbol{\varepsilon}_{\boldsymbol{x}}(k) \\ \boldsymbol{y}^*(k) &\leftarrow \boldsymbol{C}\boldsymbol{x}^*(k) + \boldsymbol{D}\boldsymbol{u}(k) + \boldsymbol{\varepsilon}_{\boldsymbol{y}}(k) \end{aligned} \tag{14}$$

where $\boldsymbol{\varepsilon}_{\boldsymbol{x}}(k)$ and $\boldsymbol{\varepsilon}_{\boldsymbol{y}}(k)$ are the vectors of roundoff errors due to the sum-of-products evaluation. Denote $\boldsymbol{\varepsilon}(k)$ the column vector that aggregates those errors:

$$\boldsymbol{\varepsilon}(k) = \begin{pmatrix} \boldsymbol{\varepsilon}_{\boldsymbol{x}}(k) \\ \boldsymbol{\varepsilon}_{\boldsymbol{y}}(k) \end{pmatrix} \in \mathbb{R}^{n+p}. \tag{15}$$

In order to capture the effects of the finite precision implementation we must take into account the propagation of the roundoff errors through the data-flow. The output error $\boldsymbol{\Delta y}(k)$ is defined as the difference between the implemented and the exact systems: $\boldsymbol{\Delta y}(k) = \boldsymbol{y}^*(k) - \boldsymbol{y}(k)$.

This difference system can be obtained by subtracting (10) to (14). It follows that $\boldsymbol{\Delta y}(k)$ can be seen as the output of the error vector $\boldsymbol{\varepsilon}(k)$ through the

filter $\mathscr{H}_\varepsilon$ with $(\boldsymbol{A}, \boldsymbol{B}_\varepsilon, \boldsymbol{C}, \boldsymbol{D}_\varepsilon)$ as State-Space: $\boldsymbol{B}_\varepsilon = \begin{pmatrix}\boldsymbol{I}_n & \boldsymbol{0}\end{pmatrix}$, $\boldsymbol{D}_\varepsilon = \begin{pmatrix}\boldsymbol{0} & \boldsymbol{I}_p\end{pmatrix}$. Equivalently (thanks to the linearity of the considered system), the implemented system can be seen as the sum of the exact system $\mathscr{H}$ and the error system $\mathscr{H}_\varepsilon$ with $\boldsymbol{\varepsilon}(k)$ as input, as shown on Figure 5.

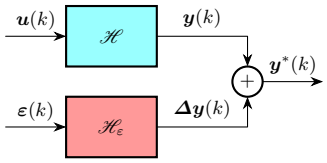


Fig. 5: Equivalent system, with errors separated.

The system $\mathscr{H}_\varepsilon$ expresses how the error propagates through the filter, and knowing some properties on the roundoff errors $\boldsymbol{\varepsilon}(k)$ will lead to properties on the output error $\boldsymbol{\Delta y}(k)$, and hence on the accuracy of the implemented algorithm.

In Coq, we assume we are given a State-Space `stsp`, a vector input signal `u` and vector error signals `err_x` and `err_y`. We also assume the obviously reasonable hypotheses that the implicit size of output vectors for `stsp` and the order of `stsp` are non negative ((`N_out` >= 0)%Z and (`StSp_n stsp` >= 0)%Z). We define `x'` (recursively) then `y'` (using `x'`) the vector signals corresponding to $\boldsymbol{x}^*$ and $\boldsymbol{y}^*$ in (14). We define `B_err` and `D_err` the matrices corresponding to $\boldsymbol{B}_\varepsilon$ and $\boldsymbol{D}_\varepsilon$ and `errors` as the vertical concatenation of `err_x` and `err_y`. We prove that `y'` is indeed the sum of the outputs of the exact filter described by `stsp` for input `u`, and of the error filter described by a new State-Space `stsp_err` for input `errors`.

5.2 Worst-case Peak Gain Theorem

The Worst-case Peak Gain Theorem is used to provide the maximum possible value for the outputs of a state-space filter. Applied on the system $\mathscr{H}_\varepsilon$, it gives a bound of the output error due to the finite precision computations.

We consider $\mathscr{H}$ a SISO LTI filter. Its Worst-Case Peak Gain (WCPG) [20,21], noted $\langle\!\langle \mathscr{H} \rangle\!\rangle$, is an element of $\overline{\mathbb{R}}$ defined as:

$$\langle\!\langle \mathscr{H} \rangle\!\rangle = \sum_{k=0}^{\infty} |h(k)| \tag{16}$$

where h is the impulse response of $\mathscr{H}$ (see §2.3). If an input signal u is bounded by M ($\forall k,\ |u(k)| \leq M$), then the corresponding output signal y is bounded by $\langle\!\langle \mathscr{H} \rangle\!\rangle\, M \in \overline{\mathbb{R}}$. We can also write this as:

$$\forall u, \quad \sup_{k\in\mathbb{Z}}(\mathscr{H}\{u\}(k)) \ \leq\ \langle\!\langle \mathscr{H} \rangle\!\rangle \sup_{k\in\mathbb{Z}}(u(k)) \tag{17}$$

Moreover, the WCPG is optimal: it is the smallest number that verifies (17). In Coq, we define the WCPG exactly as in (16) using `Lim_seq` from Coquelicot [9].

```
Definition sum_abs_IR (H : filter) (n : nat) :=
  sum 0 (Z.of_nat n) (fun k : Z => Rabs ((impulse_response H) k)).
Definition wcpg (H : filter) : Rbar := Lim_seq (sum_abs_IR H).
```

To prove (17), we rely on the fact that the image by a LTI filter of a signal u can be obtained as the convolution of u and the impulse response of the filter: $\mathscr{H}\{u\}(k) = \sum_{l\in\mathbb{Z}} u(l)h(k-l)$ (5)We also prove the optimality of the WCPG, with two theorems depending on whether the WCPG is finite. For both of them, from the definition of $\langle\!\langle \mathscr{H} \rangle\!\rangle$ as an infinite sum, we can get an index N such that $\sum_{k=0}^{N} |h(k)|$ is sufficiently close to $\langle\!\langle \mathscr{H} \rangle\!\rangle$. Then, we define a signal u such that for $0 \leq k \leq N$, $u(k)$ is in $\{1, -1\}$ and has the same sign as $h(N-k)$. We obtain $\mathscr{H}\{u\}(N) = \sum_{l\in\mathbb{Z}} u(l)h(N-l) = \sum_{0\leq l\leq N} |h(l)|$.

```
Theorem wcpg_theorem (H : filter) (u : signal) (M : R) :
  LTI_filter H -> (forall k : Z, Rabs (u k) <= M) ->
    (forall k : Z, Rbar_le (Rabs (H u k)) (Rbar_mult wcpg M)).
Theorem wcpg_optimal (H : filter) : LTI_filter H -> is_finite (wcpg H) ->
  forall epsilon : R, epsilon > 0 -> exists (u : signal) (N : Z),
    ( (forall k : Z, Rabs (u k) <= 1) /\ (H u) N > wcpg H - epsilon ).
Theorem infinite_wcpg_optimal (H : filter) : LTI_filter H ->
  wcpg = p_infty -> forall M : R, exists (u : signal) N,
    ( (forall k : Z, Rabs (u k) <= 1) /\ H u N > M ).
```

Another important property of a filter is that, for any bounded input ($\exists M,\ \forall k,$ $|u(k)| \leq M$), the output is bounded as well (by a number M' independently from M *a priori*). This property is known as the Bounded Input Bounded Output (BIBO) stability [22], and is shared by most filters that are of practical interest. The WCPG of a LTI filter verifying this is always finite.

Defining the `bounded` property for signals, then the `BIBO` property for filters is straightforward. The formulation of the theorem is quite simple, even if it has required the most intermediary lemmas by far, some being rather long to prove.

```
Theorem BIBO_wcpg_finite (H : filter) : LTI_filter H -> BIBO H -> is_finite (wcpg H).
```

The principle is to prove the contrapositive: if the WCPG is infinite then we will build an input u bounded by 1 such that the output is unbounded. As seen in the proof of optimality of the WCPG, for any bound M we can construct an input u bounded by 1 and an index N such that $|\mathscr{H}\{u\}(N)| > M$. What allows us to get from this construction where u can be chosen after M, to a single unbounded signal, is firstly that the property $|\mathscr{H}\{u\}(N)| > M$ only depends on values of u for $0 \leq k \leq N$, and secondly that we are able to adapt this construction to leave an arbitrary number of preset input values unchanged: for any $M \in \mathbb{R}$, $K \in \mathbb{Z}$ and imposed values $u(0), ..., u(K)$ that are bounded by 1, we can extend them into a signal u still bounded by 1 such that $|\mathscr{H}\{u\}(N)| > M$ for some index N. Iterating this construction, we build a signal such that its image has at least a term exceeding 0, a term exceeding 1, and so on at least a term exceeding M for any $M \in \mathbb{Z}$. These repeated constructions are what is tricky to build in Coq. We started using the `{...|...}` strong existential construction rather than

existential quantifier.We relied on the Limited Principle of Omniscience (LPO) to actually construct an index N where a sequence exceeds a given bounds.

6 Conclusions and perspectives

We have formalized digital filters in the Coq proof assistant and provided several realizations. For one of these realizations, namely the State-Space, we have proved theorems about error analysis, that will be useful when finite-precision arithmetic will come into play. This formalization was not that difficult as the mathematics are clear in textbooks. The main difficulty was the induction on $\mathbb{Z}$ as described in §3.2: deciding that the standard library results were not exactly what we needed and stating the corresponding theorems and total functions.

A part of the corresponding mathematics that we did not focus on is the transfer function. We defined it but we have not yet linked it to the rest of the development. The Z-transform has been formalized in HOL [26] and it will be interesting to see if similar theorems may be proved with our Coq formalization.

The Worst-Case Peak Gain Theorem has been proved for the SISO filters, including State-Space. The general formula $\langle\!\langle \mathscr{H} \rangle\!\rangle = |\boldsymbol{D}| + \sum_{k=0}^{\infty} |\boldsymbol{C}\boldsymbol{A}^k\boldsymbol{B}|$ has been proved with $\boldsymbol{D}$ and $\boldsymbol{C}\boldsymbol{A}^k\boldsymbol{B}$ being 1×1 matrices, implicitly converted to real numbers. To be applied on the error filter, the proof needs to be generalized to MIMO filters, which is not difficult but cumbersome due to matrix manipulation.

To handle more realizations and develop proofs (such as error analysis proofs) only once, we may use another realization called the Specialized Implicit Framework (SIF) [27]. It was designed as a unifying tool to describe and encompass all the possible realizations of a given transfer function (like the direct forms, State-Spaces, cascade or parallel decompositions, lattice filters, etc.). SIF is an extension of the State-Space realization, modified in order to allow chained Sum-of-Products operations.

A natural perspective is to handle floating-point and fixed-point computations. Indeed, digital filters are run on embedded software that cannot compute with real numbers. As far as floating-point arithmetic is concerned, the Flocq library [28,29] will suit our needs, but fixed-point will be more complicated. Even if Flocq has a fixed-point format and the corresponding theorems, we want to take overflow into account and this is hardly done within Flocq: only the IEEE-754 formalization of binary floating-point numbers assume an upper bound on the exponent. Moreover, we may want to have several handling of overflow as done in [3]. We want at least three modes: i) ensuring that no overflow happens; ii) two's complement arithmetic, where a modulo operation is used when overflow happens; iii) saturation arithmetic, where the maximal value is used when overflow happens. Adding two's complement and overflow modes to Flocq will be a necessary step towards the formal proof of the behaviors of real digital filters.

References

1. Akbarpour, B., Tahar, S.: Error analysis of digital filters using hol theorem proving. Journal of Applied Logic **5**(4) (2007) 651 – 666 Selected papers from the

4th International Workshop on Computational Models of Scientific Reasoning and Applications.
2. Hilaire, T., Lopez, B.: Reliable implementation of linear filters with fixed-point arithmetic. In: Proc. IEEE Workshop on Signal Processing Systems (SiPS). (2013)
3. Akbarpour, B., Tahar, S., Dekdouk, A.: Formalization of fixed-point arithmetic in HOL. Formal Methods in System Design **27**(1) (Sep 2005) 173–200
4. Park, J., Pajic, M., Lee, I., Sokolsky, O. In: Scalable Verification of Linear Controller Software. Springer Berlin Heidelberg, Berlin, Heidelberg (2016) 662–679
5. Park, J., Pajic, M., Sokolsky, O., Lee, I. In: Automatic Verification of Finite Precision Implementations of Linear Controllers. Springer Berlin Heidelberg, Berlin, Heidelberg (2017) 153–169
6. Feret, J.: Static Analysis of Digital Filters. In Schmidt, D., ed.: the 13th European Symposium on Programming - ESOP 2004. Volume 2986 of Lecture Notes in Computer Science., Barcelona, Spain, David A. Schmidt, Springer (March 2004) 33–48
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Texts in Theoretical Computer Science. Springer (2004)
8. The Coq Development Team: The Coq Proof Assistant Reference Manual v8.6. (2016)
9. Boldo, S., Lelay, C., Melquiond, G.: Coquelicot: A user-friendly library of real analysis for coq. Mathematics in Computer Science **9**(1) (2015) 41–62
10. Oppenheim, A.V., Schafer, R.W., Buck, J.R.: Discrete-time Signal Processing (2Nd Ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1999)
11. Fettweiss, A.: Wave digital filters: Theory and practice. Proc. of the IEEE **74**(2) (1986)
12. Gazsi, L.: Explicit formulas for lattice wave digital filters. IEEE Trans. Circuits & Systems **32**(1) (1985)
13. Middleton, R., Goodwin, G.: Digital Control and Estimation, a unified approach. Prentice-Hall International Editions (1990)
14. Palaniswami, M., Feng, G.: Digital estimation and control with a new discrete time operator. In: Proc. 30th IEEE Conf. Decision Contr., Brighton, U.K. (December 1991) 1631–1632
15. Li, G., Wan, C., Bi, G.: An improved rho-dfiit structure for digital filters with minimum roundoff noise. IEEE Trans. on Circuits and Systems **52**(4) (April 2005) 199–203
16. Feng, Y., Chevrel, P., Hilaire, T.: A practical strategy of an efficient and sparse FWL implementation of LTI filters. In: European Control Conference (ECC'09). (2009) 1383–1388
17. Hanselmann, H.: Implementation of digital controllers - a survey. Automatica **23**(1) (January 1987) 7–32
18. Gevers, M., Li, G.: Parametrizations in Control, Estimation and Filtering Probems. Springer-Verlag (1993)
19. Hilaire, T.: On the transfer function error of state-space filters in fixed-point context. IEEE Trans. on Circuits & Systems II **56**(12) (December 2009) 936–940
20. Balakrishnan, V., Boyd, S.: On computing the worst-case peak gain of linear systems. Systems & Control Letters **19** (1992) 265–269
21. Boyd, S.P., Doyle, J.: Comparison of peak and rms gains for discrete-time systems. Syst. Control Lett. **9**(1) (June 1987) 1–6
22. Kailath, T.: Linear Systems. Prentice-Hall (1980)

23. Lopez, J., Carreras, C., Nieto-Taladriz, O.: Improved interval-based characterization of fixed-point LTI systems with feedback loops. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **26**(11) (November 2007) 1923–1933
24. Kim, S., Kum, K., Sung, W.: Fixed-point optimization utility for C and C++ based digital signal processing programs. IEEE Transactions on Circuits and Systems **45**(11) (November 1998) 1455–1464
25. Volkova, A., Hilaire, T., Lauter, C.Q.: Reliable evaluation of the Worst-Case Peak Gain matrix in multiple precision. In: 22nd IEEE Symposium on Computer Arithmetic, Jun 2015, Lyon, France. 2015, Lyon, France (2015)
26. Siddique, U., Mahmoud, M.Y., Tahar, S. In: On the Formalization of Z-Transform in HOL. Springer International Publishing, Cham (2014) 483–498
27. Hilaire, T., Chevrel, P., Whidborne, J.: A unifying framework for finite wordlength realizations. IEEE Trans. on Circuits and Systems **8**(54) (August 2007) 1765–1774
28. Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in Coq. In Antelo, E., Hough, D., Ienne, P., eds.: Proceedings of the 20th IEEE Symposium on Computer Arithmetic, Tübingen, Germany (July 2011) 243–252
29. Boldo, S., Melquiond, G.: Computer Arithmetic and Formal Proofs. ISTE Press - Elsevier (December 2017)