Architecture synthesis for linear time-invariant filters

Antoine Martinet

June 11, 2015

Contents

Introduction

Filters are nowadays essential tools for designing responsive systems. In signal processing, filters are used in radio signal coding and decoding, image and sound processing, and so on. In addition, they are also used in many control applications.

For most applications, filters can be computed in software. However, for performance reasons hardware implementations are needed in many applications. There is an interest in FPGAs implementations be cause they can help for prototyping a software defined radio (SDR).

SDR is the main research subject of the SOCRATE team, at the CITI lab, where this work has been conducted. The goal of this internship was to design a parametric architecture generator for Linear Time Invariant (LTI) filters. This work has been integrated to FloPoCo tool, whose purpose is to generate architecture cores for computing just right.

This report first presents the context of LTI filters, SIF and their implementation in part 1. Then we will describe the details of our implementation in arbitrary precision in part 2. Finally, an overview of the future work will be presented in part 3.

1 Tools for expressing filters and signal processing

1.1 Fundamentals about signal processsing

1.1.1 Definition of a signal

Lopez's PhD [6] gives a good presentation of the state of the art. Many notations and definitions are kept from this PhD.

Definition 1. *(Signal) Generally, a signal is a temporal variable, which takes a value from $\mathbb{R}$ at each time t. We denote $x(t)$ the value of the signal x at the instant t. When dealing with discrete time events, the time will be represented by k. Then we denote about $x(k)$, which is said to be a sample. $\{x(k)\}_{k\geq 0}$ denotes all the values possible for the signal x. In the rest of this report, we will discuss about vectors of signals $\boldsymbol{x}$, where $\boldsymbol{x}(k) \in \mathbb{R}^n$*

1.1.2 Linear Time Invariant Filters (LTI filters)

A filter, denoted by its transfer function $\mathcal{H}$, is an application which transforms a signal vector $\boldsymbol{u}$ (with $dim(\boldsymbol{u}) = n_u$) into a signal $\boldsymbol{y} = \mathcal{H}(\boldsymbol{u})$, of size $dim(\boldsymbol{y}) = n_y$. When $n_u = n_y = 1$, we speak about Single Input Single Output (SISO) filters. In other cases, we speak about Multiple Input Multiple Output (MIMO) filters.

Definition 2. *(Linear Time Invariant Filter) Linearity:*

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

Time invariance:

$$\{\mathcal{H}(\boldsymbol{u})(k-k_0)\}_{k\geq 0} = \mathcal{H}(\{\boldsymbol{u}(k-k_0)\}_{k\geq 0})$$

1.1.3 Impulse response

Definition 3. *(Impulse Response) A SISO filter may be defined by its impulse response, denoted h. h is the impulse response of H to the impulsion of Dirac. Indeed each input can be described as a sum of Dirac impulsions:*

$$u = \sum_{i\geq 0} u(l)\delta_l$$

where δ_l is a Dirac impulsion centered in l, that is:

$$\delta(k) = \begin{cases} 1 & when\ \ k = l \\ 0 & else \end{cases} \tag{1.1.1}$$

The linearity condition of $\mathcal{H}$ implies: $\mathcal{H}(u) = \sum_{l\geq 0} u(l)\mathcal{H}(\delta_l)$. Time invariance gives: $\mathcal{H}(\delta_l)(k) = h(k-l)$. Then the computation from inputs to outputs takes this form:

$$y(k) = \sum_{l\geq 0} u(l)h(k-l) = \sum_{l=0}^{k} u(k)h(k-l)$$

This corresponds with the convolution product definition of u by h, denoted $y = h * u$. Dealing with MIMO filters, we have $\boldsymbol{h} \in \mathbb{R}^{n_y \times n_u}$ as the impulse response of $\mathcal{H}$. $\boldsymbol{h}_{i,j}$ is the response on the ith output to the Dirac implusion on the j-th input. The precedent equation becomes:

$$y_i(k) = \sum_{j=1}^{n_u} \sum_{l\geq}^{l} u_j(l)h_{i,j}(k-l),\ \forall 1 \leq i \leq n_y$$

1.1.4 Worst-Case Peak Gain (WCPG) of a Filter: the maximum amplification expectable

Definition 4. *(Worst-Case Peak Gain) The worst case peak gain is defined as the maximum amplification possible over all potential inputs through the filter.*

$$\|\mathcal{H}\|_{wcpg} = \sup_{u\neq 0} \frac{\|h*u\|_{l^\infty}}{\|u\|_{l^\infty}}$$

with h the impulse response of $\mathcal{H}$, u the input signal, and $h*u$ the convolution product of h by u (output of the filter).

1.2 FIR and IIR: two filters families

There is two types of LTI filters: *Finite impulse response* (FIR) and *Infinite impulse response* (IIR) filters. Formally, we define the impulse response as finite when:

$$\exists n \in \mathbb{N} | \forall k \geq n, h(k) = 0 \tag{1.2.1}$$

The smallest n verifying 1.2.1 is referred as the order of the filter. So a n-order FIR can be described by the following equation:

$$y(k) = \sum_{i=0}^{n} b_i u(k-i) \tag{1.2.2}$$

An IIR will be described as following:

$$y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=0}^{n} a_i y(k-i) \tag{1.2.3}$$

Here one can observe that the output at time k depends also on all previous n outputs (loopback). One can also see that a FIR can be seen as an IIR with $\forall i \in [0,n], a_i = 0$ The impulse response can then be deduced from 1.2.3 by resolving the recurrence relation:

$$h(k) = \begin{cases} 0 & when k < l \\ b_k - \sum_{l=1}^{n} a_l h(k-l) & when 0 \leq k \leq n \\ \sum_{l=1}^{n} a_l h(k-l) & when n < k \end{cases} \tag{1.2.4}$$

1.3 Different realizations: how to compute the output of a filter?

Definition 5. *(realization) A realization can be defined as an algorithm describing how to compute outputs from inputs. However, a realization does not describes the details of basic operations (format, size, order, rounding, etc...)*

It is important to know that all realizations of a filter are mathematically equivalent to each other (infinite precision). But in finite precision, rounding aspects have a huge impact on the correctness of results. Most of the time in this report, we will describe realizations as forms.

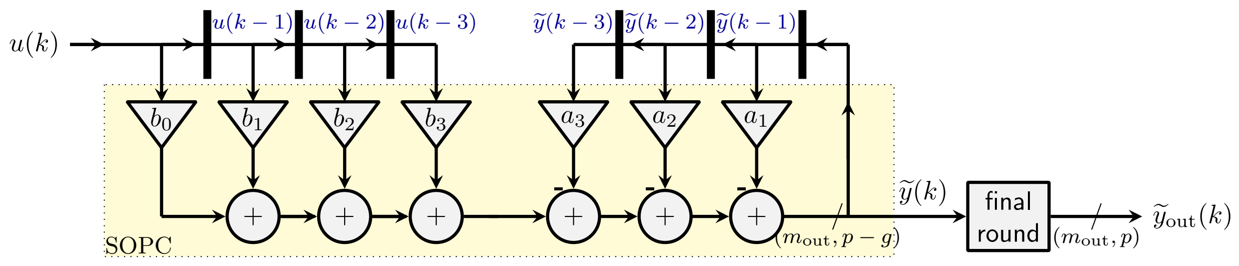


Figure 1: Abstract architecture for the direct form realization of an LTI filter

1.3.1 Direct and transposed forms

Direct and transposed forms are classic realizations. A good description of these forms can be found in Lopez' and Hilaire's PhDs [6] [5]. The direct form has been implemented and in the FoPoCo project, you can see the hardware implementation an IIR on figure 1.3.1

1.3.2 State-space representation: a recurrence to define an infinite response

This type of realization consists in expressing the evolution of a system considering it's state at time k. In continuous time, it is described by differential equations at first order. In discret time (in which we are interested in), it is described by a simple recurrence:

$$\begin{cases} \boldsymbol{x}(k+1) = \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}\boldsymbol{u}(k) \\ \boldsymbol{y}(k+1) = \boldsymbol{C}\boldsymbol{x}(k) + \boldsymbol{D}\boldsymbol{u}(k) \end{cases} \tag{1.3.1}$$

Where $\boldsymbol{x}(k) \in \mathbb{R}^{n_x}$ is the state vector, $\boldsymbol{u}(k) \in \mathbb{R}^{n_u}$ is the input vector and $\boldsymbol{y}(k) \in \mathbb{R}^{n_y}$ is the output vector, at time k. The matrices $\boldsymbol{A} \in \mathbb{R}^{n_x \times n_x}$, $\boldsymbol{B} \in \mathbb{R}^{n_x \times n_u}$, $\boldsymbol{C} \in \mathbb{R}^{n_y \times n_x}$, and $\boldsymbol{D} \in \mathbb{R}^{n_y \times n_u}$, with $\boldsymbol{x}(0)$ are sufficient to describe an LTI filter, with convention $\boldsymbol{x}(k) = \boldsymbol{u}(k) = 0 \mid \forall k < 0$.

1.4 The Specialized Implicit Form (SIF): a unified representation

1.4.1 Definition

The specialized implicit form (SIF) was introduced in [8] and is well detailed in Hilaire's PhD and associated papers [5, 7]. The classical state-space representation is intuitive, but it doesn't takes into account the reality of implementation. Indeed, dealing with finite precision and error amplification, the order in which operations are done becomes crucial. Another idea is to have a unique representation for any realization of LTI filters, that allows to compute every degradation measures instead of redevelopping them for each new realization. This form distinguishes computations done at one time from computations done the other times. As well as in a state-space, we have $\boldsymbol{x}_i$ as state variables, but in addition to that, $\boldsymbol{t}_i$ are intermediate variables. The SIF is described as following:

$$\begin{pmatrix} \boldsymbol{J} & 0 & 0 \\ -\boldsymbol{K} & \boldsymbol{I}_{n_x} & 0 \\ -\boldsymbol{L} & 0 & \boldsymbol{I}_{n_y} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k+1) \\ \boldsymbol{x}(k+1) \\ \boldsymbol{y}(k) \end{pmatrix} = \begin{pmatrix} 0 & \boldsymbol{M} & \boldsymbol{N} \\ 0 & \boldsymbol{P} & \boldsymbol{Q} \\ 0 & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k) \\ \boldsymbol{x}(k) \\ \boldsymbol{u}(k) \end{pmatrix} \tag{1.4.1}$$

With n_t, n_x, n_y and n_u the sizes of $\boldsymbol{t}$,$\boldsymbol{x}$,$\boldsymbol{y}$ and $\boldsymbol{u}$, respectively. $\boldsymbol{J}$, is a lower triangular matrix, with diagonal entries equal to 1. Then we have the following dimensions for the previous matrices:

$$\begin{aligned} \boldsymbol{J} \in \mathbb{R}^{n_t \times n_t}, \boldsymbol{M} \in \mathbb{R}^{n_t \times n_x}, \boldsymbol{N} \in \mathbb{R}^{n_t \times n_u}, \\ \boldsymbol{K} \in \mathbb{R}^{n_x \times n_t}, \boldsymbol{P} \in \mathbb{R}^{n_x \times n_x}, \boldsymbol{Q} \in \mathbb{R}^{n_x \times n_u}, \\ \boldsymbol{L} \in \mathbb{R}^{n_y \times n_t}, \boldsymbol{R} \in \mathbb{R}^{n_y \times n_x}, \boldsymbol{S} \in \mathbb{R}^{n_y \times n_u}, \end{aligned} \tag{1.4.2}$$

(1.4.3)

The best way to understand the SIF may be to see it as an algorithm, each line of the equation 1.4.1 corresponding to a sequential step of the computation. The algorithm results as follows:

```
for int i = 0 ; i ≤ $n_t$; i++ do
    $\boldsymbol{t}_i(k+1) \leftarrow -\sum_{j<i} \boldsymbol{J}_{ij}\boldsymbol{t}_j(k+1) + \sum_{j=1}^{n_x} \boldsymbol{M}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{N}_{ij}\boldsymbol{u}_j(k)$
end
for int i = 0 ; i ≤ $n_x$; i++ do
    $\boldsymbol{x}_i(k+1) \leftarrow -\sum_{j=1}^{n_t} \boldsymbol{K}_{ij}\boldsymbol{t}_j(k+1) + \sum_{j=1}^{n_x} \boldsymbol{P}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{Q}_{ij}\boldsymbol{u}_j(k)$
end
for int i = 0 ; i ≤ ny; i++ do
    $\boldsymbol{y}_i(k+1) \leftarrow -\sum_{j=1}^{n_t} \boldsymbol{L}_{ij}\boldsymbol{t}_j(k+1) + \sum_{j=1}^{n_x} \boldsymbol{R}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{S}_{ij}\boldsymbol{u}_j(k)$
end
```

Algorithm 1: Computation of SIF outputs from inputs

Here, it is important to see that the form of $\boldsymbol{J}$ allows to compute the $\boldsymbol{t}_i$ sequentially. The algorithm can then be described as follows:

```
for int i = 0 ; i ≤ $n_t$; i++ do
    $\boldsymbol{t}_i(k+1) \leftarrow -\boldsymbol{J'}_i\boldsymbol{t}(k+1) + \boldsymbol{M}_i\boldsymbol{x}(k) + \boldsymbol{N}_i\boldsymbol{u}(k)$
end
for int i = 0 ; i ≤ $n_x$; i++ do
    $\boldsymbol{x}_i(k+1) \leftarrow -\boldsymbol{K}_i\boldsymbol{t}(k+1) + \boldsymbol{P}_i\boldsymbol{x}(k) + \boldsymbol{Q}_i\boldsymbol{u}(k)$
end
for int i = 0 ; i ≤ ny; i++ do
    $\boldsymbol{y}_i(k+1) \leftarrow -\boldsymbol{L}_i\boldsymbol{t}(k+1) + \boldsymbol{R}_i\boldsymbol{x}(k) + \boldsymbol{S}_i\boldsymbol{u}(k)$
end
```

Algorithm 2: Simplified matricial algorithm

With $\boldsymbol{J'} = \boldsymbol{J} - I_{n_t}$.

Values of the vector $\boldsymbol{t}(k+1)$ are computed and used at the same iterations, so they are not kept in memory. As the equation is mostly full of zeros, it is more convenient to use it's compressed formulation, wich is denoted as the $\boldsymbol{Z}$:

$$\boldsymbol{Z} = \begin{pmatrix} -\boldsymbol{J} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} \tag{1.4.4}$$

The community usually takes $-\boldsymbol{J}$ for simplicity within further computations.

We can of course bind the SIF with the ABCD (classic state-space) form, which gives:

$$\begin{aligned} \boldsymbol{A_Z} = \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{P},\ \ \boldsymbol{B_Z} = \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{Q}, \\ \boldsymbol{C_Z} = \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{R},\ \ \boldsymbol{D_Z} = \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{S}, \end{aligned} \tag{1.4.5}$$

With:

$$\begin{aligned} \boldsymbol{A_Z} \in \mathbb{R}^{n_x \times n_x}, \boldsymbol{B_Z} \in \mathbb{R}^{n_x \times n_u}, \\ \boldsymbol{C_Z} \in \mathbb{R}^{n_y \times n_x}, \boldsymbol{D_Z} \in \mathbb{R}^{n_x \times n_u}, \end{aligned} \tag{1.4.6}$$

1.4.2 Optimal SIF realization

Here, we won't discuss the optimality of the SIF description. This optimality is up to the user, who may have a lot of good reasons not to design a realization that seem optimal for us. The optimization of the realization is a job done at LIP6 in the PEQUAN team, under the metalibm ANR project. The present work is just a hardware backend for the optimization part. The expected use of this work is shown on figure 2.

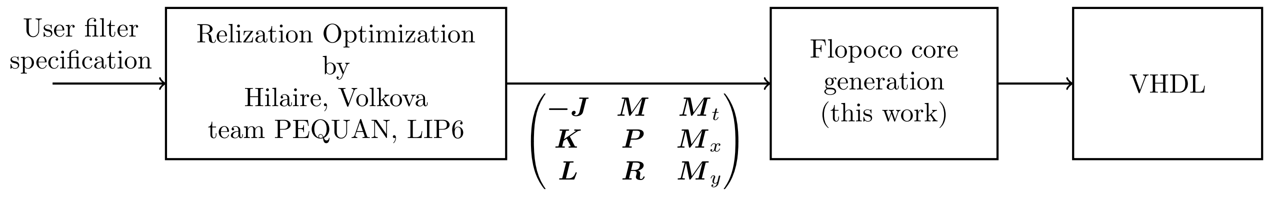

User filter specification
Relization Optimization by Hilaire, Volkova team PEQUAN, LIP6
$\begin{pmatrix} -J & M & M_t \\ K & P & M_x \\ L & R & M_y \end{pmatrix}$
Flopoco core generation (this work)
VHDL

Figure 2: Workflow overview of tools usage

1.5 Implementation: what can be done in practise?

1.5.1 The FloPoCo Framework: computing just right

FloPoCo is a C++ framework [3] which first purpose is to generate floating point cores in VHDL. Although it may have some pertinence in the ASIC design market, it main target is the configuration of FPGAs as computing units. Indeed, FPGAs are a rising market for embedded computing, because they offer full reconfigurability and a relatively good balance between computation power and price.

More information is available at http://flopoco.gforge.inria.fr/

In this work, we propose to implements SIFs using the SOPCs arithmetical core from FloPoCo. SOPC stands for Sum of Producs by Constants, and is a KCM-multiplier based architecture. The details of the SOPC architecture has been described in [4].

1.5.2 KCM multipliers: an architecture to efficiently compute products on FPGAs

To be quick, a KCM multiplier is designed to fully use the power of look-up tables (LUTs) in an FPGA. To do so, it partitions the constant into chunks which match the size of a LUT. Then, tabulating the multiplications, we end up with a final sum to compute the product. This has been first described by K.Chapman in [1]. The detail of implementation is visible on figure 3

1.5.3 SOPCs: exploiting full parallel architecures

The SOPC architecture keeps the same idea. The difference is, the summation architecture is mutualized through the n_c products in a bit-heap based architecture implemented by flopoco. The detail is visible on figure 4.

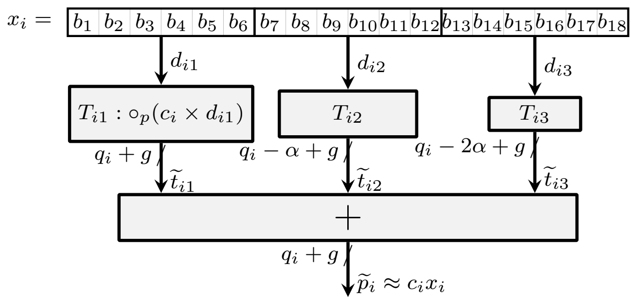

$x_i =$
$b_1\ b_2\ b_3\ b_4\ b_5\ b_6\ b_7\ b_8\ b_9\ b_{10} b_{11} b_{12} b_{13} b_{14} b_{15} b_{16} b_{17} b_{18}$
d_{i1}
d_{i2}
d_{i3}
$T_{i1} : \circ_p(c_i \times d_{i1})$
T_{i2}
T_{i3}
$q_i + g$
$q_i - \alpha + g$
$q_i - 2\alpha + g$
$\widetilde{t}_{i1}$
$\widetilde{t}_{i2}$
$\widetilde{t}_{i3}$
$+$
$q_i + g$
$\widetilde{p}_i \approx c_i x_i$

Figure 3: The FixRealKCM method when x_i is split in 3 chunks

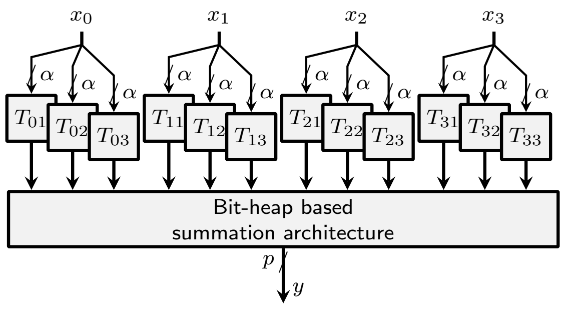


Figure 4: KCM-based SPOC architecture for $n_c = 4$, each input being split into 3 chunks

1.5.4 Precision concerns

In SOPCs architectures, the accuracy is deducible from the inputs/outputs specifications and the size of the constants.

This is described in [4].

Dealing with feedback inputs, the question of precision is more complicated. Indeed, when results loop back to inputs, as soon as we are in finite precision, the error is amplified by a certain amount, depending on the coefficients, at each pass through the filter.

In this case the solution in the industry is to build an equivalent FIR filter by resolving the recurrence. This leads to build an accurate hardware, but at the price of a huge waste of logic. The present work is an answer to this problem, trying to compute just right, keeping the recurrence and saving hardware.

The main idea to dimension such filters is to consider the total error as a single filter. The result of this filter is then added to the perfect filter to get the final output.

In finite precision, sizes are constrained to be all the same. The demonstration of the size computation has been described in Lopez' PhD [6]. The idea now is to see what we can do in arbitrary precision, trying to save a maximum of logic while keeping a right result on the precision required by the user.

Here we have to compute each size at each step of the computation. Indeed, the WCPG is not useful for the first part of a FIR, as it has no loop. So we just need the WCPG for the second part, because it is just in this part of the circuit that there is a potential error amplification.

1.5.5 Size computation in finite precision

When there is many potential realizations for a single LTI filter, we can see that the choice of one realization among the others is very related to the error analysis in output. The precision of our computations can then be defined so that the following condition is satisfied:

$$\varepsilon_{y_i} < 2^{-lsb_{y_i}} \ \ \forall i \in [1, n_y] \tag{1.5.1}$$

With lsb_{y_i} the last significant bit of the output y_i, and ε_{y_i} the error on the computation of the output y_i

Following the same model, we define an error quantum for each SOPC involved in the computation. All these error quantas will be functions of the most significant and last significant bits (respectively msb and lsb) of every input, output and intermediate signal.

We define the following vectors:

$$\boldsymbol{\varepsilon}(k) = \begin{pmatrix} \boldsymbol{\varepsilon}_t(k) \\ \boldsymbol{\varepsilon}_x(k) \\ \boldsymbol{\varepsilon}_y(k) \end{pmatrix} = \begin{pmatrix} \varepsilon_{t_1}(k) \\ \varepsilon_{t_2}(k) \\ \vdots \\ \varepsilon_{t_{n_t}}(k) \\ \\ \varepsilon_{x_1}(k) \\ \varepsilon_{x_2}(k) \\ \vdots \\ \varepsilon_{x_{n_x}}(k) \\ \\ \varepsilon_{y_1}(k) \\ \varepsilon_{y_2}(k) \\ \vdots \\ \varepsilon_{y_{n_y}}(k) \end{pmatrix} \tag{1.5.2}$$

1.5.6 Error analysis

Considering a filter $\mathcal{H}$ and it's SIF, following the algorithm 2, in finite precision, we get the following equations:

$$\boldsymbol{t}^*(k+1) = -\boldsymbol{J'}\boldsymbol{t}^*(k+1) + \boldsymbol{M}\boldsymbol{x}^*(k) + \boldsymbol{N}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_t}(k) \tag{1.5.3}$$
$$\boldsymbol{x}^*(k+1) = \boldsymbol{K}\boldsymbol{t}^*(k+1) + \boldsymbol{P}\boldsymbol{x}^*(k) + \boldsymbol{Q_i}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_x}(k) \tag{1.5.4}$$
$$\boldsymbol{y}^*(k+1) = \boldsymbol{L}\boldsymbol{t}^*(k+1) + \boldsymbol{R}\boldsymbol{x}^*(k) + \boldsymbol{S_i}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_y}(k) \tag{1.5.5}$$

Here $\boldsymbol{t}^*$, $\boldsymbol{x}^*$ and $\boldsymbol{y}^*$ are the computed vectors, so with computations errors.

As $\boldsymbol{\varepsilon}$-errors come only from the SOPCs cores, we have another error quantum that will symbolize the total error. We denote

$$\boldsymbol{\delta t}(k+1) = \boldsymbol{t}_i^*(k) - \boldsymbol{t}_i(k) \tag{1.5.6}$$
$$\boldsymbol{\delta x}(k+1) = \boldsymbol{x}_i^*(k) - \boldsymbol{x}_i(k) \tag{1.5.7}$$
$$\boldsymbol{\delta y}(k+1) = \boldsymbol{y}_i^*(k) - \boldsymbol{y}_i(k) \tag{1.5.8}$$

the total error at instant k, considering computations errors and loopback, for $\boldsymbol{t}$, $\boldsymbol{x}$ and $\boldsymbol{y}$. We get then:

$$\boldsymbol{\delta t}(k+1) = -\boldsymbol{J'}\boldsymbol{\delta t}(k+1) + \boldsymbol{M}\boldsymbol{\delta x}(k) + \boldsymbol{\varepsilon_t}(k) \tag{1.5.9}$$
$$\boldsymbol{\delta x}(k+1) = \boldsymbol{K}\boldsymbol{\delta t}(k+1) + \boldsymbol{P}\boldsymbol{\delta x}(k) + \boldsymbol{\varepsilon_x}(k) \tag{1.5.10}$$
$$\boldsymbol{\delta y}(k+1) = \boldsymbol{L}\boldsymbol{\delta t}(k+1) + \boldsymbol{R}\boldsymbol{\delta x}(k) + \boldsymbol{\varepsilon_y}(k) \tag{1.5.11}$$

This new algorithm corresponds here to the algorithm of the SIF of a filter $\mathcal{H}_{\boldsymbol{\varepsilon}}$, which describes the behaviour of computation errors at time k on the output. The linearity condition allows to decompose the real $\mathcal{H}^*$ filter in two distinct filters:

- $\mathcal{H}$ the absolute filter in infinite precision
- $\mathcal{H}_{\boldsymbol{\varepsilon}}$ the error filter

According to 1.4.4, we have:

$$\boldsymbol{Z_\varepsilon} = \begin{pmatrix} -\boldsymbol{J} & \boldsymbol{M} & \boldsymbol{M}_t \\ \boldsymbol{K} & \boldsymbol{P} & \boldsymbol{M}_x \\ \boldsymbol{L} & \boldsymbol{R} & \boldsymbol{M}_y \end{pmatrix} \tag{1.5.12}$$

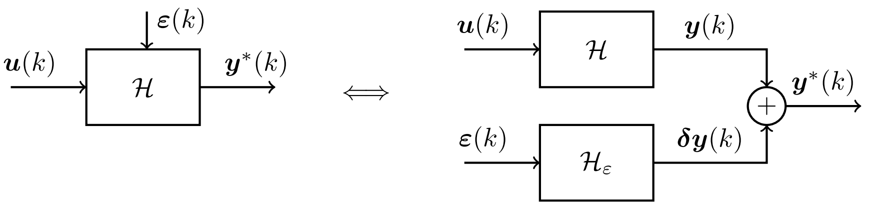


Figure 5: A signal view of the error propagation with respect to the ideal filter

with:

$$\boldsymbol{M}_t = (\boldsymbol{I}_{n_t}\boldsymbol{0}_{n_t\times n_x}\boldsymbol{0}_{n_t\times n_y}), \tag{1.5.13}$$

$$\boldsymbol{M}_x = (\boldsymbol{0}_{n_x\times n_x}\boldsymbol{I}_{n_x}\boldsymbol{0}_{n_x\times n_y}), \tag{1.5.14}$$

$$\boldsymbol{M}_y = (\boldsymbol{0}_{n_y\times n_t}\boldsymbol{0}_{n_y\times n_x}\boldsymbol{I}_{n_y}), \tag{1.5.15}$$

$\mathcal{H}_{\boldsymbol{\varepsilon}}$ is a filter with $(n_t + n_x + n_u)$ inputs and n_y outputs.

Proposition 1. The transfert function of filter $\mathcal{H}_{\boldsymbol{\varepsilon}}$, denoted $\boldsymbol{H}_\varepsilon$, is defined as follows:

$$\boldsymbol{H}_\varepsilon :\to \boldsymbol{C}_{\boldsymbol{Z}}(z\boldsymbol{I}_n - \boldsymbol{A}_{\boldsymbol{Z}})^{-1}\boldsymbol{M}_1 + \boldsymbol{M}_2 \ \ \forall z \in \mathbb{C} \tag{1.5.16}$$

with $\boldsymbol{A}_{\boldsymbol{Z}}$ and $\boldsymbol{C}_{\boldsymbol{Z}}$ the matrices defined by 1.4.5 and

$$\boldsymbol{M}_{\boldsymbol{1}} = (\boldsymbol{K}\boldsymbol{J}^{-1} \quad \boldsymbol{I}_{n_x} \quad \boldsymbol{0}),\ \ \boldsymbol{M}_{\boldsymbol{2}} = (\boldsymbol{L}\boldsymbol{J}^{-1} \quad \boldsymbol{0} \quad \boldsymbol{I}_{n_y}), \tag{1.5.17}$$

The demonstration is well detailed in Lopez' PhD [6].

Corollary 1. Considering a filter $\mathcal{H}$, $\varepsilon(k)$ the vector of computation errors at time k in the finite precision of $\mathcal{H}$, and $\mathcal{H}\boldsymbol{\varepsilon}$ the error filter associated to $\mathcal{H}$. The behaviour of error can be described from $\varepsilon(k)$ and $\mathcal{H}\boldsymbol{\varepsilon}$. We consider the error as an interval vector, denoted by its center and radius $\langle\boldsymbol{\varepsilon}_m, \boldsymbol{\varepsilon}_r\rangle$ and the interval vector of global error $\boldsymbol{\delta y}$, denoted $\langle\boldsymbol{\delta y}_m, \boldsymbol{\delta y}_r\rangle$. In practise, all inputs are centered around zero, which is not the case in the command community, where this notation makes sense. So we consider that $\boldsymbol{\varepsilon}_m = 0$. The results are the following:

$$\boldsymbol{\delta y}_m = 0 \tag{1.5.18}$$

$$\boldsymbol{\delta y}_r = \langle\langle\mathcal{H}_{\boldsymbol{\varepsilon}}\rangle\rangle_{wcpg} \cdot \boldsymbol{\varepsilon}_r \tag{1.5.19}$$

In the future we will then get rid of $boldsymbol\delta y_m$.
Let's define:

$$n' = n_t + n_x + n_y$$

and:

$$\boldsymbol{v'} = \begin{pmatrix} \boldsymbol{t}(k+1) \\ \boldsymbol{x}(k+1) \\ \boldsymbol{y}(k) \end{pmatrix} \tag{1.5.20}$$

Then, following Lopez's computations, we can derive precisions for every intermediate step:

$$|\boldsymbol{\delta y}_i| \leq \sum_{j=1}^{n'} |\langle\langle\mathcal{H}_{\boldsymbol{\varepsilon}}\rangle\rangle_{i,j}| \cdot \boldsymbol{2}^{\boldsymbol{l}_{\boldsymbol{v'_j}}} \tag{1.5.21}$$

To formalize with a matricial formulation, we get:

$$|\boldsymbol{\delta y}| \leq |\langle\langle\mathcal{H}_{\boldsymbol{\varepsilon}}\rangle\rangle| \cdot \boldsymbol{2}^{\boldsymbol{lsb}_{\boldsymbol{v'}}} \tag{1.5.22}$$

To satisfy the condition 1.5.1, we can simply define ξ as the minimal error the user wants. We can then state:

$$|\langle\langle\mathcal{H}_{\boldsymbol{\varepsilon}}\rangle\rangle| \cdot \mathbf{2}^{\boldsymbol{lsb_{v'}}} < \xi \tag{1.5.23}$$

That is,

$$\mathfrak{A} \cdot \mathbf{2}^{lsb_{v'} - msb_{v'} - 1} < \mathbf{1}_{n_y} \tag{1.5.24}$$

Where:

$$\mathfrak{A}_{i,j} = |\langle\langle\mathcal{H}_{\boldsymbol{\varepsilon}}\rangle\rangle_{i,j}| \cdot \frac{\mathbf{2}^{\boldsymbol{msb_{v'_j}+1}}}{\xi_i} \tag{1.5.25}$$

So, for computations in arbitrary precision, we have here a sufficient condition (1.5.24) to implement the filters.

1.5.7 Worst-case peak gain computation

Computing the worst-case peak gain accurately in finite precision is a very complex problem. As we can't afford doing such a work, we are about to integrate code developped by Anastasia Lozanova Volkova from Hilaire's team at LIP6 [9], under the metalibm project.

2 Implementation

2.1 Architecture generation algorithm

```
for i=1; i=Z.size(); i++ do
    row[ ]= Z[i][ ] //pick first row of Z
    for j=1; j=1; j=Z.size() j++ do
        assign(SOPC[i], row[j], TXU) //where TXU, is the indicator of the signal (this is just determined
        by the position of the coefficient)
    end
    Second pass for wiring.
end
```

An example of implementation for a real-life case is given on figure 6.
On the figure 6, the colors in the matrix represent the different steps of computaions:

- blue for t computations
- green for x computations
- green for y computations

On the architecture scheme, we have:

- the grey background for indicating x registers
- purple for loopback form the x registers
- orange for inputs

A small precision is needed here. Here, most of coefficients of the $\boldsymbol{J}$ coefficient are null. We just kept them under this form to show how the algorithm works and how the architecture is built.

2.2 Particular Forms, degenerated cases

2.3 ABCD Form: still achievable

The ABCD Form can be considered as a degenerated form of the SIF, with $n_t = 0$. The algorithm will work in this case too.

2.4 When $n_x = 0$

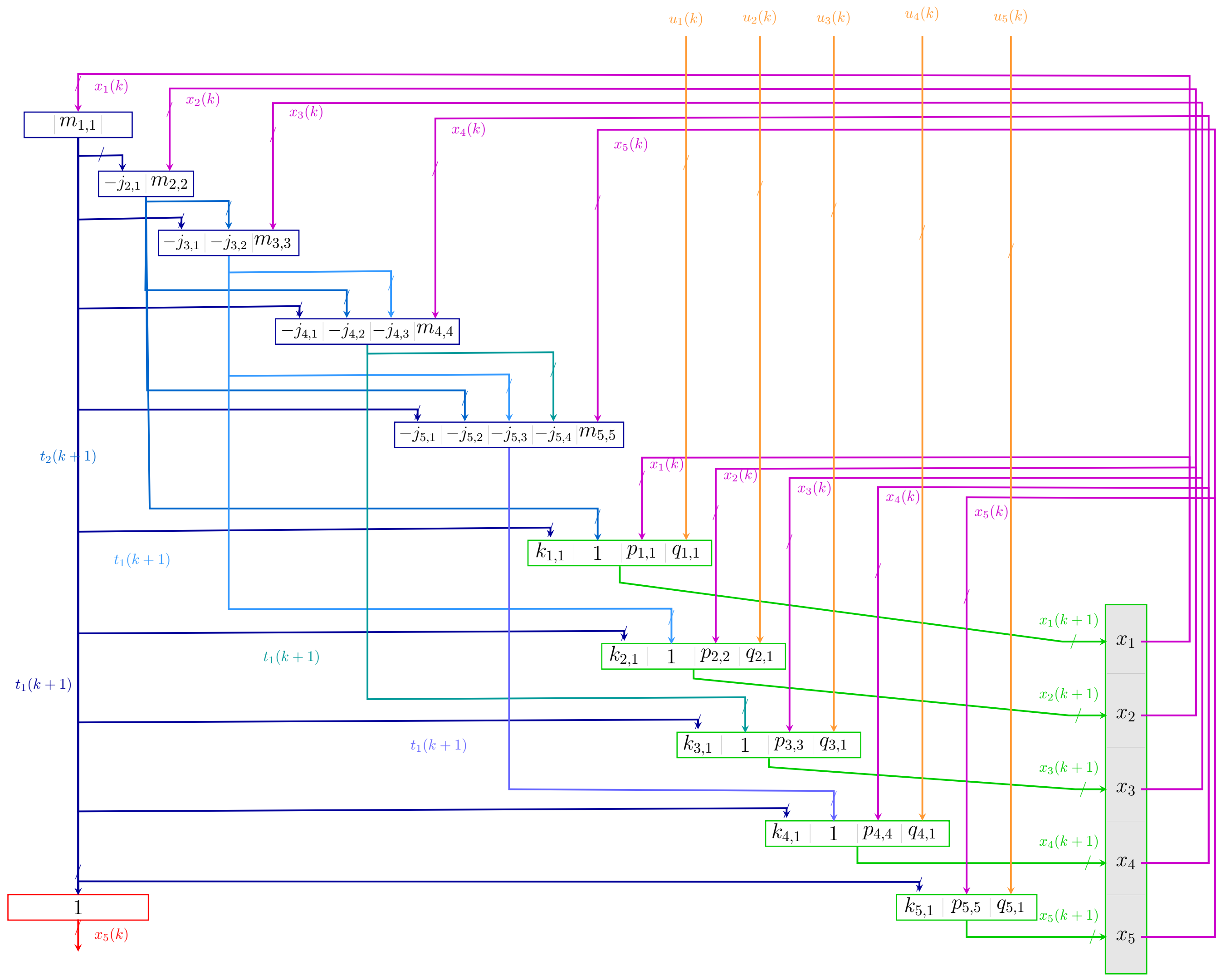
When $n_x = 0$, the interest of using implicit form is of course very limited. This case is equivalent to building a FIR, and no loopback is needed. Still, the algorithm will work, allocating only SOPCs operators. The computation of precisions are then equivalent to the FIR precision computaion, described in an article that is about to be published.

2.5 Optimizations to do on this technology

2.5.1 Sparse matrices

The Z matrix of a SIF is most of the time sparse. So it is useful to remove zeros coefficients before allocating SOPCs. Indeed, it prevents useless inputs to be declared and can save a lot of hardware, although the HDL compiler might be able to optimize the hardware and remove "dead code". Anyway, it is healthy to keep a low compile time (either in flopoco or in the HDL compiler). Keeping the VHDL clean is always important, first for debugging issues, but also for comprehensiveness.

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & m_{1,1} & 0 & 0 & 0 & 0 & 0 \\
j_{2,1} & 1 & 0 & 0 & 0 & 0 & m_{2,2} & 0 & 0 & 0 & 0 \\
j_{3,1} & j_{3,2} & 1 & 0 & 0 & 0 & 0 & m_{3,3} & 0 & 0 & 0 \\
j_{4,1} & j_{4,2} & j_{4,3} & 1 & 0 & 0 & 0 & 0 & m_{4,4} & 0 & 0 \\
j_{5,1} & j_{5,2} & j_{5,3} & j_{5,4} & 1 & 0 & 0 & 0 & 0 & m_{5,5} & 0 \\
k_{1,1} & 1 & 0 & 0 & 0 & p_{1,1} & 0 & 0 & 0 & 0 & q_{1,1} \\
k_{2,1} & 0 & 1 & 0 & 0 & 0 & p_{2,2} & 0 & 0 & 0 & q_{2,1} \\
k_{3,1} & 0 & 0 & 1 & 0 & 0 & 0 & p_{3,3} & 0 & 0 & q_{3,1} \\
k_{4,1} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & p_{4,4} & 0 & q_{4,1} \\
k_{5,1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & p_{5,5} & q_{5,1} \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$



$u_1(k)$ $u_2(k)$ $u_3(k)$ $u_4(k)$ $u_5(k)$
$x_1(k)$ $x_2(k)$ $x_3(k)$ $x_4(k)$ $x_5(k)$
$m_{1,1}$
$-j_{2,1}$ $m_{2,2}$
$-j_{3,1}$ $-j_{3,2}$ $m_{3,3}$
$-j_{4,1}$ $-j_{4,2}$ $-j_{4,3}$ $m_{4,4}$
$-j_{5,1}$ $-j_{5,2}$ $-j_{5,3}$ $-j_{5,4}$ $m_{5,5}$
$t_2(k+1)$
$t_1(k+1)$
$k_{1,1}$ 1 $p_{1,1}$ $q_{1,1}$
$k_{2,1}$ 1 $p_{2,2}$ $q_{2,1}$
$k_{3,1}$ 1 $p_{3,3}$ $q_{3,1}$
$k_{4,1}$ 1 $p_{4,4}$ $q_{4,1}$
$k_{5,1}$ $p_{5,5}$ $q_{5,1}$
1
$x_5(k)$
$x_1(k+1)$ $x_2(k+1)$ $x_3(k+1)$ $x_4(k+1)$ $x_5(k+1)$
x_1 x_2 x_3 x_4 x_5

Figure 6: Architecture generation for implementing a SIF (example from a rho-DFII filter), with $n_t = 5$, $n_x = 5$ and $n_y = 1$

2.5.2 Power of two coefficients

Coefficients equal to one or a power of two in the Z matrix can be interpreted as simple wires instead of multiplications in the SOPC. So, we could eventually replace entries in SOPCs by simple additions with the result of the SOPC. Here, we should investigate to see what solution is the best in terms of hardware consumption (speed is not concerned here because the speed is determined by the length of the loop).

2.6 Concrete work

This implementation work has been done in the FixFilter section of the FloPoCo project. It is for now about 850 lines and contains:

- the implentation of the SIF algorithm
- the specification parser
- the testing framework (emulate method)

The testing framework consists on pre-computing expected results using the sollya lib [2]. Then the FloPoCo framework generates a VHDL testbench that embed the hardware architecture and the test of expected results. As an exhaustive testbench is impossible to generate($2^{msb_{In}-lsb_{In}}{}^{2^msbIn-lsbIn}$ values to test for each input), FloPoCo generates a user-defined number of random tests that should be enough, if the number is large enough, to validate the implementation.

2.7 Filter specification interface

To communicate with our SIF operator in FloPoCo, we can't simply pass the coefficients in command line as it is done for the first order of direct form. So we defined a simple file format to store all the coefficients. The format is defined as follows:

```
X l c
x_1_1 x_1_2 ... x_1_c
x_2_1 x_2_2 ... x_2_c
.
.
.
x_l_1 x_l_2 ... x_l_c
```

Where X is the name of the matrix (X $\in$ { J, K, L, M, N, P, Q, R, S, T}), x_i_j is the coefficient (with $i \in [1, l]$ and $j \in [1, c]$), l is the number of lines and c the numer of columns.

All the matrices are specified after each other in the file.

For now another file is used to specify sizes because we do not have not the integration of the "wcpg-code" done.

3 Further work

3.1 Static analysis on coefficients

For now, we are performing detection on null coefficients. Detecting ones entries in the coefficients is an improvement very simple to implement. This will open the discussion of the efficiency of integrating or not these one entries in the SOPCs, because we don't know for now what choice is the best between:

- integrating the one entry in the SOPC
- keeping the one entry out of the SOPC and adding it to the result of the SOPC

Moreover, depending on the number of one entries, the implementations concerns might vary from one option to the other one in terms of logic consumption.

3.2 Sub-filter detection

Detecting independent loops can be very interesting in the context of saving hardware. Considering a sub-filter with its own loop can permit to use its WCPG to compute the precisions for this sub-filter. We get then another problem with a precision specification that depends on the rest of the filter (logic after the sub-filter in the pipeline).

3.3 Precision calculations improvement

In this work, we kept original calculations trying to adapt them to our context. However, lots of approximations are done through this calculations. Working on this part, going back over all these calculations could really improve the efficiency and the size of all implementations.

3.4 File format re-specification

Specifying new formats could help to improve the visibility and the usability of this new operator. The idea is to avoid specification output formatting operations for the user. So we could stick to the basic output of Python and Matlab matrices.

Conclusion

In this work, I tried to give an overview of LTI filters, and the considerations to take into account when trying to implement them. I tried to adapt concepts and calculation from Lopez's and Hilaire's work to the context of generating architectures computing just right. This context is merging ideas from the communities of automatic, signal, and computer arithmetics, so there was a part of unification in notations and conventions. Finally, I began the implementation of a parametric definition in the FloPoCo framework. This was permitted through the definition of a specification language and the integration of external code from LIP6. Further work will take into account many improvements and optimizations, such as:

- static analysis on coefficients
- sub-filter detection
- bounds and precision computation improvements
- format re-specification

References

[1] K.D. Chapman. Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner). *EDN magazine*, 39(10):80, May 1993.

[2] S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An environment for the development of numerical codes. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 28–31, Heidelberg, Germany, September 2010. Springer.

[3] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, July 2011.

[4] Abdelbassat Massouri Florent de Dinechin, Matei Istoan. Sum-of-product architectures computing just right. In *IEEE*, 2007.

[5] Thibaut Hilaire. Analyse et synthèse de l'implémentation de lois de contrôle-commande en précision finie. In *PhD*, 2006.

[6] Benoit Lopez. Implémentation optimale de filtre linéaire en arithmétique virgule fixe. In *PhD*, 2014.

[7] P.Chevel T.Hilaire and J.F.Whidbornz. A unifying framework for finite wordlength realizations. In *IEEE*, 2007.

[8] P.Chevel T.Hilaire and Y.Trinquet. Implicit state-space representation: a unifying framework for fwl implementation of lti systems. In *Proc. of the 16th IFAC World Congress.*, 2005.

[9] A. Volkova, T. Hilaire, and C. Lauter. Reliable evaluation of the Worst-Case Peak Gain matrix in multiple precision. In *IEEE Symposium on Computer Arithmetic*, 2015.

Appendix: Notations and reminders about signal processing

LTI filters in general are usually defined as sums of products. Several quantities are useful to understand their characteristics

3.5 Notations

During the entire report we will use several notations and conventions:

- in signal processing, t is the common notation for continue time and k the notation for discrete time. This report keeps this convention.
- $\langle\langle \mathcal{H} \rangle\rangle_{WCPG}$ is the worst case peak gain of the filter $\mathcal{H}$
- y is an output variable
- x is a state variable
- t is an intermediate variable
- a bold symbol (for example $\boldsymbol{h}$) denotes a matrix, that may be a vector
- a normal symbol (for example $\varepsilon_{t_1}(k)$) denotes a single value

3.6 Reminders about signal processing

Definition 6. *(Signal) Generally, a signal is a temporal variable, which takes a value from $\mathbb{R}$ at each time t. We denote x(t) the value of the signal x at the instant t. When dealing with discrete time events, the time will be represented by k. Then we talk about x(k), which is said to be a sample. $\{x(k)\}_{k\geq 0}$ denotes all the values possible for the signal x. In the rest of this report, we will talk about vectors of signals $\boldsymbol{x}$, where $\boldsymbol{x}(k) \in \mathbb{R}^n$*

Definition 7. *(ℓ^1 norm) The ℓ^1 norm of a scalar signal x, denoted $\|x\|_{\ell^1}$, is the sum of absolute values of $x(k)$ at each instant k:*

$$\|x\|_{\ell^1} = \sum_{k=0}^{+\infty} |x(k)| \tag{3.6.1}$$

This norm exists only if x is ℓ^1-sommable, that is, if and only if the equation 3.6.1 converges.

Definition 8. *(ℓ^2 norm) The ℓ^2 norm of a scalar signal x, denoted $\|x\|_{\ell^2}$, is defined as follows:*

$$\|x\|_{\ell^2} = \sqrt{\sum_{k=0}^{+\infty} x(k)^2} \tag{3.6.2}$$

This norm exists only if x is square-sommable, that is, if and only if the equation 3.6.2 converges.

Definition 9. *(ℓ^∞ norm) The ℓ^∞ norm of a scalar signal x, denoted $\|x\|_{\ell^\infty}$, is the smallest upper bound among all values (absolute values) possible for the signal x, that is:*

$$\|x\|_{\ell^\infty} = \sup_{k\in\mathbb{N}} |x(k)| \tag{3.6.3}$$

Definition 10. *(Dirac) The Dirac function, denoted δ describes an impulsion of infinite amplitude at a discrete time k (supposed to be instaneous):*

$$\delta(k) = \begin{cases} 1 & \textit{when } k = 0 \\ 0 & \textit{else} \end{cases} \tag{3.6.4}$$