Architecture Synthesis for Linear Time-Invariant Filters

Antoine Martinet

CITI lab,
INRIA's SOCRATE Team,

Under the supervision of
Florent de Dinechin

2 February - 31 July, 2015

Table of Contents

Introduction

LTI filters: essential tools in signal processing and control communities Implementation in software works well. Idea: implementing LTI Filters in hardware can be very useful

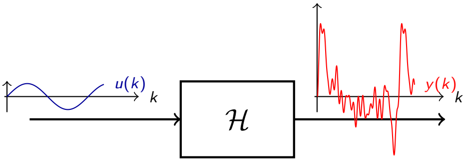
Table of Contents

Table of Contents

Fundamentals: signal processing and filters
LTI Filters

LTI (Linear Time Invariant) filters are particular filters that are:

- Linear: outputs are linear combinations of inputs (allows to use linear algebra definitions)
- Time-Invariant: all coefficients are constant



What is the purpose of this work?

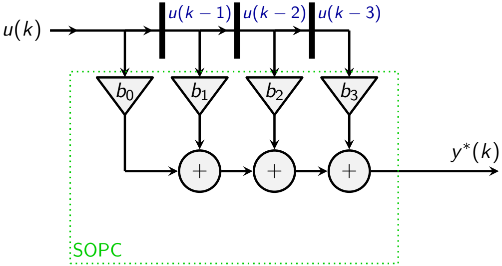
- Lopez's PhD thesis states how to compute LTI filters in software:
  - ordonancing issues
  - fixed size
- The goal here is to do such a work in hardware, where we have more flexibility:
  - full parallelism
  - arbitrary size

In this context, constraints become degrees of freedom.

Table of Contents

FIR definition:

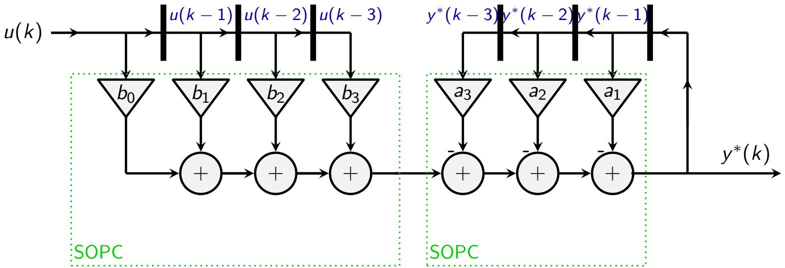
$$y(k) = \sum_{i=0}^{n} b_i u(k-i) \qquad (1)$$



Abstract architecture for the direct form realization of an LTI filter

IIR definition:

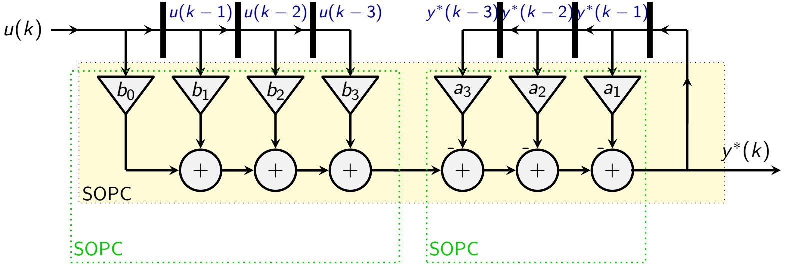
$$y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=0}^{n} a_i y(k-i) \tag{1}$$



Abstract architecture for the direct form realization of an LTI filter

IIR definition:

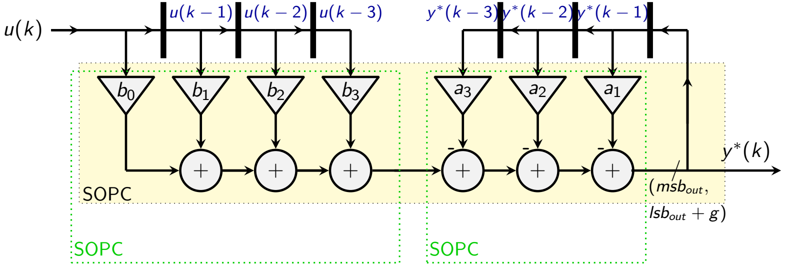
$$y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=0}^{n} a_i y(k-i) \qquad (1)$$



Abstract architecture for the direct form realization of an LTI filter

IIR definition:

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



Abstract architecture for the direct form realization of an LTI filter

Here we have loop-backs, so registers:

$$\text{Loop-backs} \iff \text{Registers}$$

We can generalise by introducing state registers.

Table of Contents

State-Space representation

The "ABCD" form

Let's define $\boldsymbol{x}(k)$ a state vector (hardware register)

$$\begin{cases} \boldsymbol{x}(k+1) = \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}\boldsymbol{u}(k) \\ \boldsymbol{y}(k+1) = \boldsymbol{C}\boldsymbol{x}(k) + \boldsymbol{D}\boldsymbol{u}(k) \end{cases} \tag{2}$$

With:

$$\boldsymbol{A} \in \mathbb{R}^{n_x \times n_x}, \boldsymbol{B} \in \mathbb{R}^{n_x \times n_u},$$
$$\boldsymbol{C} \in \mathbb{R}^{n_y \times n_x}, \boldsymbol{D} \in \mathbb{R}^{n_y \times n_u}$$

Equivalent matrix formulation:

$$\begin{pmatrix} \boldsymbol{x}(k+1) \\ \boldsymbol{y}(k+1) \end{pmatrix} = \begin{pmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}(k) \\ \boldsymbol{u}(k) \end{pmatrix} \tag{3}$$

Table of Contents

Definition of the SIF

Problems with ABCD:

- *lsb* and *msb* computations have to be rebuilt for each new filter in this form.
- this doesn't gives an explicit order in the operations

The SIF generalizes the state-space.
Addition: $\boldsymbol{t}(k)$ describes the operations order:

$$\begin{pmatrix} \boldsymbol{J} & \boldsymbol{0} & \boldsymbol{0} \\ -\boldsymbol{K} & \boldsymbol{I}_{n_x} & \boldsymbol{0} \\ -\boldsymbol{L} & \boldsymbol{0} & \boldsymbol{I}_{n_y} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k+1) \\ \boldsymbol{x}(k+1) \\ \boldsymbol{y}(k) \end{pmatrix} = \begin{pmatrix} \boldsymbol{0} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{0} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{0} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k) \\ \boldsymbol{x}(k) \\ \boldsymbol{u}(k) \end{pmatrix} \quad (4)$$

The SIF as an algorithm

for *int* $i = 0$; $i \leq n_t$; $i{+}{+}$ **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* $i = 0$; $i \leq n_x$; $i{+}{+}$ **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* $i = 0$; $i \leq ny$; $i{+}{+}$ **do**

$$\boldsymbol{y}_i(k) \leftarrow \sum_{j=1}^{n_t} \boldsymbol{L}_{ij}\boldsymbol{t}_j(k+1) + \sum_{j=1}^{n_x} \boldsymbol{R}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{S}_{ij}\boldsymbol{u}_j(k)$$

end

Algorithm 1: Computation of SIF outputs from inputs

Workflow

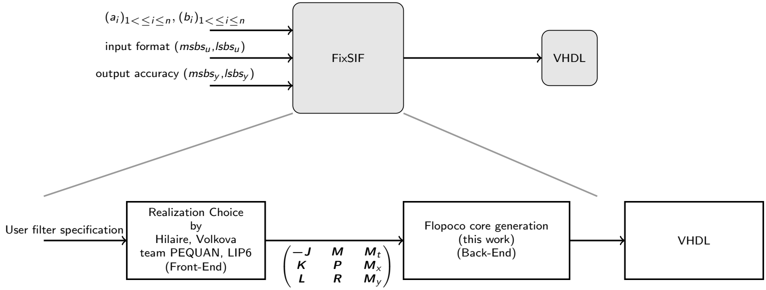


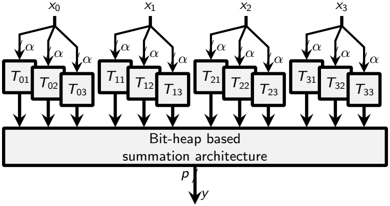
Figure: Workflow overview of tools usage

Table of Contents

Here we use FloPoCo, which is a C++ framework which first purpose is to generate floating point cores in VHDL. It is described

in Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo.
IEEE Design & Test of Computers, 28(4):18–27, July 2011
http://flopoco.gforge.inria.fr/

Table of Contents



KCM-based SOPC architecture for $n_c = 3$, each input being split into 3 chunks

Table of Contents

Architecture generation algorithm

computePrecisions([*msbs*, *lsbs*][][]) //get the matrix of msbs lsbs, functions of the wcpg.

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], "T","X","U",[msbs,lsbs][i][j])

  end

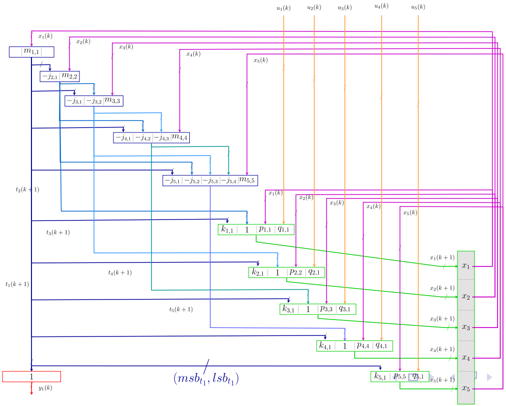
  Second pass for wiring.

end

Algorithm 2: Architecture Generation Algorithm

Example

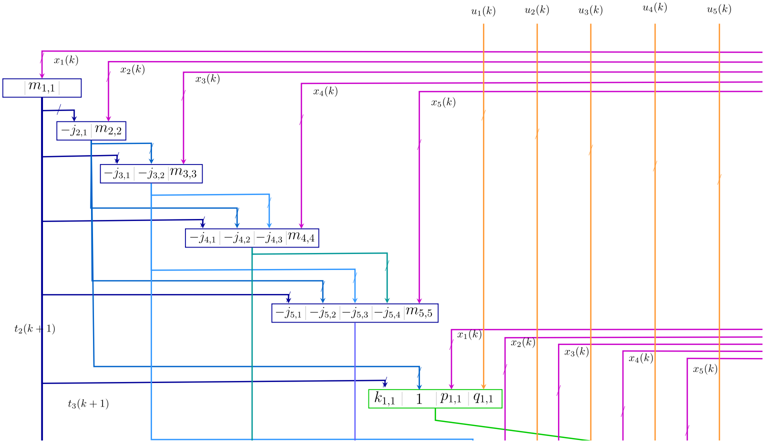
$$Z = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & m_{1,1} & 0 & 0 & 0 & 0 & 0 \\
j_{2,1} & 1 & 0 & 0 & 0 & 0 & m_{2,2} & 0 & 0 & 0 & 0 \\
j_{3,1} & j_{3,2} & 1 & 0 & 0 & 0 & 0 & m_{3,3} & 0 & 0 & 0 \\
j_{4,1} & j_{4,2} & j_{4,3} & 1 & 0 & 0 & 0 & 0 & m_{4,4} & 0 & 0 \\
j_{5,1} & j_{5,2} & j_{5,3} & j_{5,4} &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



Example

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

Example


$u_1(k)$
$u_2(k)$
$u_3(k)$
$u_4(k)$
$u_5(k)$
$x_1(k)$
$x_2(k)$
$x_3(k)$
$x_4(k)$
$x_5(k)$
$m_{1,1}$
$-j_{2,1}$ $m_{2,2}$
$-j_{3,1}$ $-j_{3,2}$ $m_{3,3}$
$-j_{4,1}$ $-j_{4,2}$ $-j_{4,3}$ $m_{4,4}$
$-j_{5,1}$ $-j_{5,2}$ $-j_{5,3}$ $-j_{5,4}$ $m_{5,5}$
$t_2(k+1)$
$t_3(k+1)$
$k_{1,1}$ 1 $p_{1,1}$ $q_{1,1}$

Example

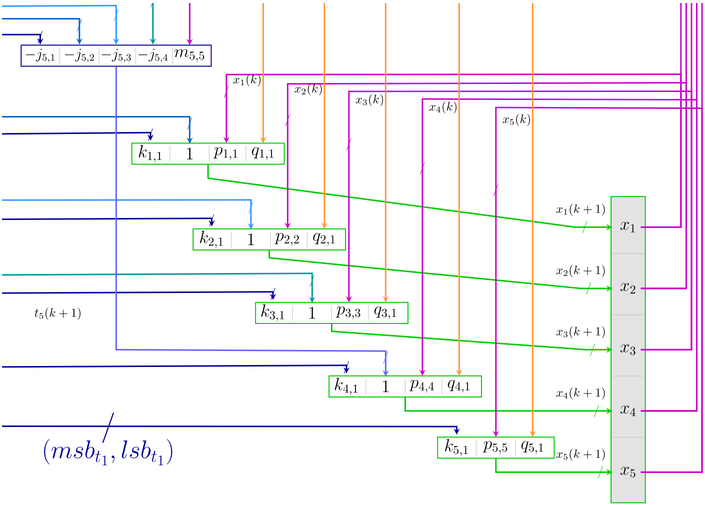

$-j_{5,1}$ $-j_{5,2}$ $-j_{5,3}$ $-j_{5,4}$ $m_{5,5}$
$x_1(k)$
$x_2(k)$
$x_3(k)$
$x_4(k)$
$x_5(k)$
$k_{1,1}$ 1 $p_{1,1}$ $q_{1,1}$
$k_{2,1}$ 1 $p_{2,2}$ $q_{2,1}$
$k_{3,1}$ 1 $p_{3,3}$ $q_{3,1}$
$k_{4,1}$ 1 $p_{4,4}$ $q_{4,1}$
$k_{5,1}$ $p_{5,5}$ $q_{5,1}$
$t_5(k+1)$
(msb_{t_1}, lsb_{t_1})
$x_1(k+1)$
$x_2(k+1)$
$x_3(k+1)$
$x_4(k+1)$
$x_5(k+1)$
x_1
x_2
x_3
x_4
x_5

Table of Contents

Error definition

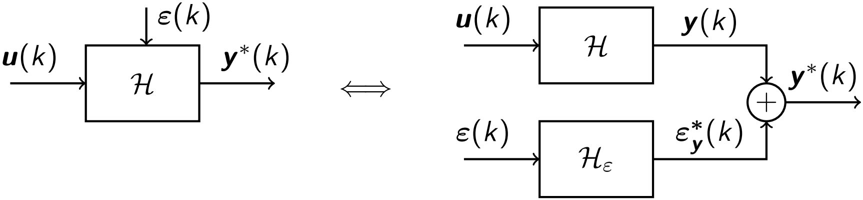
Let's define: $v' = n_t + n_x + n_y$

ξ will be the desired error. Errors introduced by SOPCs:

$$\varepsilon'_v(k) = \begin{pmatrix} \varepsilon_t(k) \\ \varepsilon_x(k) \\ \varepsilon_y(k) \end{pmatrix} = \begin{pmatrix} \varepsilon_{t_1}(k) \\ \varepsilon_{t_2}(k) \\ \vdots \\ \varepsilon_{t_{n_t}}(k) \\ \varepsilon_{x_1}(k) \\ \varepsilon_{x_2}(k) \\ \vdots \\ \varepsilon_{x_{n_x}}(k) \\ \varepsilon_{y_1}(k) \\ \varepsilon_{y_2}(k) \\ \vdots \\ \varepsilon_{y_{n_y}}(k) \end{pmatrix} \tag{5}$$

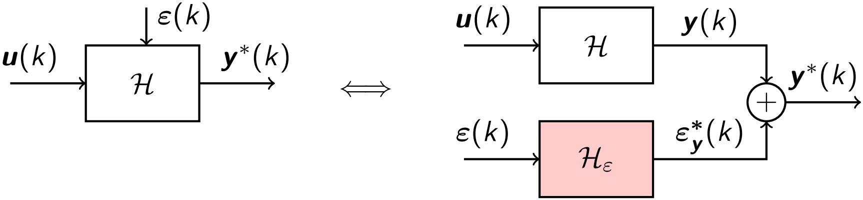
Total error: $\varepsilon^*_{v'}(k)$ defined as $\varepsilon'_v(k)$

Point of view about error



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

Point of view about error



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

Impact of errors

for *int* $i = 0$; $i \leq n_t$; i++ **do**

$$\boldsymbol{t}_i(k+1) \leftarrow -\sum_{j<i} \boldsymbol{J}_{ij}\boldsymbol{t}_j(k+1) + \sum_{j=1}^{n_x} \boldsymbol{M}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{N}_{ij}\boldsymbol{u}_j(k) + \varepsilon_{t_i}$$

end

for *int* $i = 0$; $i \leq n_x$; i++ **do**

$$\boldsymbol{x}_i(k+1) \leftarrow \sum_{j=1}^{n_t} \boldsymbol{K}_{ij}\boldsymbol{t}_j(k+1) + \sum_{j=1}^{n_x} \boldsymbol{P}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{Q}_{ij}\boldsymbol{u}_j(k) + \varepsilon_{x_i}$$

end

for *int* $i = 0$; $i \leq ny$; i++ **do**

$$\boldsymbol{y}_i(k) \leftarrow \sum_{j=1}^{n_t} \boldsymbol{L}_{ij}\boldsymbol{t}_j(k+1) + \sum_{j=1}^{n_x} \boldsymbol{R}_{ij}\boldsymbol{x}_j(k) + \sum_{j=1}^{n_u} \boldsymbol{S}_{ij}\boldsymbol{u}_j(k) + \varepsilon_{y_i}$$

end

Algorithm 3: Computation of SIF outputs from inputs

Main constraint:

$$\varepsilon_v < 2^{-lsb_v} \tag{6}$$

Proposed solution:

$$|\langle\langle\mathcal{H}_\varepsilon\rangle\rangle| \cdot 2^{lsb_{v'}+1} < 2^{-lsb_{y_i}} \tag{7}$$

Then: constraint on each line of the computation algorithm

Table of Contents

Definition of the Worst-Case-Peak-Gain (WCPG)

$$\|\mathcal{H}\|_{wcpg} = \sup_{u \neq 0} \frac{\|h * u\|_{l^\infty}}{\|u\|_{l^\infty}} \tag{8}$$

This computation is done by colleagues in LIP6.

Table of Contents

Future Work

- Removal of power of two: adapt KCM and SOPC FloPoCo cores
- Sub-filter detection: open question for either Front-End and Back-End
- Precision calculations improvement
- File format re-specification

Conclusion

To conclude

- Adapting LTI Filters generation from software to hardware
- Reuse of Lopez's calculations in our context
- Implementation for FPGAs in a parametric view

References

Small bibliography

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

Any question?

Full definition of the SIF

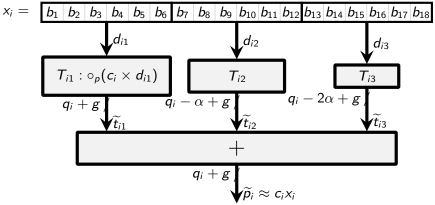
$$\begin{pmatrix} \boldsymbol{J} & \boldsymbol{0} & \boldsymbol{0} \\ -\boldsymbol{K} & \boldsymbol{I}_{n_x} & \boldsymbol{0} \\ -\boldsymbol{L} & \boldsymbol{0} & \boldsymbol{I}_{n_y} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k+1) \\ \boldsymbol{x}(k+1) \\ \boldsymbol{y}(k) \end{pmatrix} = \begin{pmatrix} \boldsymbol{0} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{0} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{0} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k) \\ \boldsymbol{x}(k) \\ \boldsymbol{u}(k) \end{pmatrix} \tag{9}$$

With:

$$\begin{aligned} \boldsymbol{J} \in \mathbb{R}^{n_t \times n_t}, \boldsymbol{M} \in \mathbb{R}^{n_t \times n_x}, \boldsymbol{N} \in \mathbb{R}^{n_t \times n_u}, \\ \boldsymbol{K} \in \mathbb{R}^{n_x \times n_t}, \boldsymbol{P} \in \mathbb{R}^{n_x \times n_x}, \boldsymbol{Q} \in \mathbb{R}^{n_x \times n_u}, \quad (10) \\ \boldsymbol{L} \in \mathbb{R}^{n_y \times n_t}, \boldsymbol{R} \in \mathbb{R}^{n_y \times n_x}, \boldsymbol{S} \in \mathbb{R}^{n_y \times n_u}, \end{aligned}$$

(11)

KCM multiplier



The FixRealKCM method when x_i is split in 3 chunks

Mathematical definition of a filter $\mathcal{H}$

Definition of a filter: $\boldsymbol{y} = \mathcal{H}(\boldsymbol{u})$ With $dim(\boldsymbol{y}) = n_y$ and, $dim(\boldsymbol{u}) = n_u$

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:

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

Impulse response

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

Where δ_l is a Dirac impulsion centered in l:

$$\delta_l(k) = \begin{cases} 1 & when \;\; k = l \\ 0 & else \end{cases} \tag{12}$$

Computation of the outputs: