Reliable implementation of linear filters with fixed-point arithmetic
Thibault Hilaire, Benoit Lopez

To cite this version:
Thibault Hilaire, Benoit Lopez. Reliable implementation of linear filters with fixed-point arithmetic. Signal Processing Systems (SiPS), Oct 2013, Taipei, Taiwan. pp.401 - 406, 10.1109/SiPS.2013.6674540. hal-01076048

HAL Id: hal-01076048
https://hal.sorbonne-universite.fr/hal-01076048
Submitted on 20 Oct 2014

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

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
This work has been sponsored by french ANR agency under grant No ANR-11-INSE-008.

ABSTRACT
This article deals with the implementation of linear filters or controllers with fixed-point arithmetic. The finite precision of the computations and the roundoff errors induced may have an important impact on the numerical behavior of the implemented system. Moreover, the fixed-point transformation is a time consuming and error-prone task, specially with the objective of minimizing the quantization impact.

Based on a formalism able to describe every structure of linear filters/controllers, this paper proposes an automatic method to generate fixed-point version of the inputs-to-outputs algorithm and an analysis of the global error added on the output. An example illustrates the approach.

Index Terms— Fixed-point arithmetic, error analysis, filter implementation, code generation.

1. INTRODUCTION

The great majority of embedded signal processing or control algorithms is implemented using digital devices such as general purpose micro-controllers, DSP, ASIC or FPGA. For cost, size and power consumption reasons, the computation on these devices is mainly based on integer arithmetic (rather than floating-point arithmetic). The fixed-point arithmetic is used as an approximation of real numbers based on integers (mantissa) and implicit scaling (exponent). Unfortunately, the quantization of the embedded coefficients and the roundoff errors occurring in the finite precision computations lead to degradation in performance of the algorithm when implemented [1, 2, 3].

Unfortunately, there is no general tool to automatize and analyze the transformations required to obtain the final code from the filter or controller (as mathematical object). This final code may be executed on a device (software implementation) or describing hardware operations to be performed (hardware implementation). This problem is a difficult one, that can be decomposed in several steps [4] as illustrated in Figure 1. First, one should consider the various equivalent realizations to implement the filter. In addition to the classical direct forms or the state-space, some other interesting realizations can be considered, like the ones with the δ-operator (defined by δ ≜ \frac{x_{q+1}}{2x_q}, where q is the classical shift operator, and Δ a strictly positive constant [5]), the ρ-Direct Form II transposed (ρDFII) [6], the wave lattice filter, warped filter, and a lot of other specific realizations (LGC, LCW-structures [7], etc.).

They are all equivalent in infinite precision, but they are not anymore equivalent in finite precision, and their robustness to the finite precision implementation differs from one realization to another. Moreover, they do not imply the same number of coefficients. Criteria based on transfer function sensitivity, or pole sensitivity [1, 8] can be used as indicators of the Finite Word-Length (FWL) effects, in order to compare them and find the most suitable one. Then, for a given realization, the fixed-point conversion and the accurate error analysis must be done, according to the target. This fixed-point algorithm could then be processed to generate C code for software implementation, or used to generate dedicated hardware operators for FPGA or ASIC targets.

This paper focuses on step 3 that transforms a realization into a fixed-point algorithm, and performs an error analysis (i.e. bounds the difference between the real output and the implemented one). A complete formalization of the sum-of-products in fixed-point arithmetic is proposed here.

For that purpose, the framework used to describe the realization (SIF) is reminded in section 2. Section 3 formalizes the fixed-point arithmetic and its propagation through an Abstract Syntax Tree with the use of interval arithmetic. Then the error analysis is applied to the SIF in section 4. Finally, an illustrative example is given in section 5, before the conclusion in section 6.

Notations: Throughout this paper, real numbers are in lowercase, column vectors in lowercase boldface and matrices in uppercase boldface. \([\ell; \bar{\eta}]\) with square brackets corresponds to an inf-sup representation of an interval with \(\ell\) as the infimum and \(\bar{\eta}\) as the supremum. \((x_m; x_r)\) with angular brackets correspond to a mid-rad representation of an interval, with \(x_m\) as the middle point and \(x_r\) as the radius (obviously \(x_m = \frac{x_m + x_r}{2}\) and \(x_r = \frac{x_r - x_m}{2}\)).

2. SPECIALIZED IMPLICIT FRAMEWORK

2.1. Formalism

In order to encompass all these possibilities, the Specialized Implicit Framework (SIF) has been proposed in [9]. It can be used as a unifying framework to describe all the filter realizations and allows the study and comparison of the FWL effects. It is an extension of the state-space framework, modified in order to allow a detailed (but still macroscopic) description of the inline computations to be performed. All the linear realizations, with delays, multiplications by
constants and additions can be represented. Multiple Input Multiple Output filters or controllers are also considered.

So various classical realizations, like $q$ (shift) or $\delta$-state-spaces, classical Direct Forms I and II, cascade or parallel decompositions, mixed structures, etc. may be then described in a single unifying form (see [9] for details and examples).

Definition 1 (Specialized Implicit Framework) Equation (1) recalls the specialized implicit form that explicitly expresses the parametrization and the intermediate variables used:

$$
\begin{pmatrix}
J & 0 & 0 \\
-K & I_x & 0 \\
-L & 0 & I_y
\end{pmatrix}
\begin{pmatrix}
x(k + 1) \\
y(k)
\end{pmatrix}
=
\begin{pmatrix}
0 & M & N \\
0 & P & Q \\
0 & R & S
\end{pmatrix}
\begin{pmatrix}
x(k) \\
y(k)
\end{pmatrix}
$$

where $u(k)$ represents the $n$ inputs, $y(k)$ the $p$ outputs, $x(k)$ is the $n$ stored states (one state per delay) and $t(k+1)$ is the $l$ intermediate variables in the calculations of step $k$ (the column of $0$ in the second matrix shows that $t(k)$ is not used for the calculation at step $k$: that characterizes the concept of intermediate variables).

It is implicitly considered throughout the paper that the computations associated to (1) are done according to the Algorithm 1.

```
begin
  // Intermediate variables
  J.t(k + 1) ← M.x(k) + N.u(k)
  // state-vector update
  x(k + 1) ← K.t(k + 1) + P.x(k) + Q.u(k)
  // outputs computation
  y(k) ← L.t(k + 1) + R.x(k) + S.u(k)
end
```

Algorithm 1: General algorithm associated to the SIF

Note that $J$ is lower triangular with $1$'s on its diagonal, so the first value of $t(k+1)$ is first calculated, and then its second value is computed using its first and so on (the computation of $J^{-1}$ is then not necessary).

The coefficients of the realization can be written in a compact form

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

The equivalent transfer function is given by

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

with

$$
A_Z = KJ^{-1}M + P, \quad B_Z = KJ^{-1}N + Q, \\
C_Z = LJ^{-1}M + R, \quad D_Z = LJ^{-1}N + S.
$$

2.2. Example

Li and Hao [6, 10] have presented a new sparse structure called $\rho$DFIIt. This is a generalized of the transposed direct-form II structure with the conventional shift operator. It is a sparse realization (with $3n + 1$ parameters when $n$ is the order of the filter), leading so to an economic (few computations) implementation that is provably be very numerically efficient with sensitive filters. Let us define

$$
\rho_i : z \mapsto \frac{z-\gamma_i}{\Delta_i}, \quad \text{and} \quad g_i : z \mapsto \prod_{j=1}^{i} \rho_j(z), \quad 1 \leq i \leq n.
$$

It can be shown that any transfer function $H(z)$ can be reparametrized with $\{\alpha_i\}$ and $\{\beta_i\}$ as follows:

$$
H(z) = \frac{\beta_0 + \beta_1 g_1^{-1}(z) + \ldots + \beta_n g_n^{-1}(z)}{1 + \alpha_1 g_1^{-1}(z) + \ldots + \alpha_n g_n^{-1}(z)}, \quad \forall z \in \mathbb{C}.
$$

Equation (6) can be, for example, implemented with a transposed direct form II (see Figure 2) with $\rho_i^{-1}$ operators, that is denoted the $\rho$-Direct Form II transposed ($\rho$DFIIt).

Within the SIF Framework, the $\rho$DFIIt form is described by [8]:

$$
Z = \begin{pmatrix}
-1 & \Delta_1 & \cdots & \Delta_n & \beta_0 \\
-\alpha_1 1 & \gamma_1 & \cdots & \gamma_n & \beta_1 \\
-\alpha_2 0 & 1 & \cdots & \gamma_n & \beta_2 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
-\alpha_n 0 0 & \cdots & 1 & \gamma_n & \beta_n
\end{pmatrix}
$$

3. FIXED-POINT ARITHMETIC

3.1. Formalism and notations

In this article, the signed fixed-point arithmetic is used, with $2^w$ complement representation. Let $x$ be such a fixed-point number with $w$ bits. It can be written as

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

where $\ell$ and $m$ are the position of the least significant and most significant bits, respectively, and $x_i \in \{0, 1\}$ is the $i$th bit of $x$ as shown in Figure 3. They satisfy $m > \ell$ and $w = m - \ell + 1$ where $w$ is the word-length.

Usually $m > 0$ and $\ell < 0$, but this is not mandatory. $2^\ell$ is the quantization step of the representation.

$x$ is represented in machine by the integer $X$, such that $X = x \cdot 2^{-\ell}$. $X$ belongs to $[-2^{w-1}; 2^{w-1} - 1] \cap \mathbb{Z}$.

In the article, the notation $(m, \ell)$ is used to denote the Fixed-Point Format (FPF) of such a fixed-point number.

Remark 1 It is also common to decompose a fixed-point representation in its integer part (bits $x_m$ to $x_0$) and fractional part (bits $x_{-1}$ to $x_{-\ell}$).
3.2. Constant in fixed-point arithmetic

Let \( c \) be a real constant. It cannot necessary be exactly represented with finite word-length \( w \), and will be approximated by the closest fixed-point number \( \tilde{c} \). The following proposition gives the conversion algorithm.

**Proposition 1 (Fixed-point conversion)** Let \( c \in \mathbb{R} \) a number to be represented in signed fixed-point arithmetic with \( w \) bits as word-length. The following steps are used for the conversion and the determination of the FPF \((m, \ell)\):

1. **Step 1:** Compute \( m = \left\lfloor \log_2 |c| \right\rfloor + 1 \) if \( c > 0 \)
   \( m = \left\lceil \log_2 |c| \right\rceil \) if \( c < 0 \)

2. **Step 2:** Compute the integer \( C = \left\lfloor c \cdot 2^{w-m-1} \right\rfloor \)

3. **Step 3:** Check for particular cases:
   - If \( C = 2^w-1 \) then \( m = m + 1 \) \( C = \left\lfloor c \cdot 2^{w-m-1} \right\rfloor \)
   - If \( C = -2^{w-2} \) then \( m = m - 1 \) \( C = \left\lfloor c \cdot 2^{w-m-1} \right\rfloor \)

4. **Step 4:** Deduce \( \ell = m + 1 - w \) and \( \tilde{c} = C2^\ell \).

The operators \([\cdot]_+\) and \([\cdot]_-\) are the round towards minus infinity, round towards plus infinity and round to the nearest integer operators.

**Proof:** Step 1 finds the appropriate most significant bit \( m \) so as to represent \( c \) without overflow. \( m \) is such that \( c \in [-2^m;2^m-1] \) and \( c \notin [-2^{m-1};2^{m-1}-1] \). Equation on step 1 is based on the dissymmetry of the 2’s complement representation around 0.

Step 2 computes the associated integer, rounded to the nearest. And finally, step 3 considers some very special cases, where the rounded-to-the-nearest make \( C \) goes out of the range. This is the case, for example, for \( c = 127.9 \) and \( w = 8 \): in step 1, \( m \) is found to be equal to 7, and \( c \) is quantized to \( \tilde{c} = 128 \) \( (C = 128 \cdot 2^h) \), that is out of the range \([-128,127] \cap \mathbb{Z} \) for the FPF \((7,0)\). The same problem occurs with \( c = -128.1 \) that is rounded to \(-128 \) but is lower than \(-128 \) \((m \) is first set to 8, and \( c \) is approximated by \( \tilde{c} = -128 \) and \( C = -64 \cdot 2^h \), whereas the FPF \((7,0)\) is sufficient to represent -128).

**Proposition 2** If a variable \( x \) is known to be in the interval \([x;\tilde{x}]\), then the appropriate FPF \((m, \ell)\) to represent it with \( w \) bits is given by:

\[
m = \max(m, \tilde{m}), \quad \ell = m + 1 - w
\]

where \( m \) and \( \tilde{m} \) are the most significant bits of \( x \) and \( \tilde{x} \), respectively, obtained from proposition 1.

**Proof:** \( m \) is chosen to represent numbers in \([x;\tilde{x}]\), so \( m \) should be high enough to represent \( x \) and \( \tilde{x} \).

3.3. Sum-of-Products and fixed-point operations

As seen in section 2, Algorithm 1 gives the expression to compute the output(s) \( y(k) \) at each step \( k \) from the input(s) \( u(k) \), the intermediate variables \( \ell(k) \) and the states \( x(k) \). These matrix operations can be decomposed in several Sum-Of-Products (SoP), with all the null terms removed.

Let us consider the following Sum-Of-Products to be evaluated in fixed-point arithmetic:

\[
s = \sum_{k=1}^{n} c_i \cdot x_i
\]

where \( c_i \) are some given non-null constants and \( \{x_i\} \) some variables, only known to be in the intervals \([x;\tilde{x}]\).

To transform this expression in a fixed-point version, an Abstract Syntax Tree (AST) is used to describe the operations to be done and their order. The tree consists of internal nodes that are the operations to perform (only additions and multiplications here) and the leaves are some constants or variables. See Figure 4 for an AST representing \(12.2x + (y + 3.11z)\).

In order to provide a fixed-point version of this AST (with same operations performing on integers, and some shifts used to align the binary-point position), it is first required to determine the FPF of each node in the AST. For that purpose, some propagation rules are recursively applied in a bottom-up way. Unlike what was proposed in [11], where the FPF of a result of an operation was derived from the FPF of the two operands (so as to avoid overflow), a more precise method is proposed here, based on interval analysis arithmetic so as to avoid the possible over-estimation of the FPF.

An interval is associated to each variable and indicates in which range the variable belongs to. Then, by applying interval arithmetic rules [12], the intervals are propagated (from bottom-to-top) through the AST, in order to determine the interval of each result and their FPF (deduced from proposition 1).

The addition and multiplication operations are now considered:

\[ z \leftarrow x \odot y, \quad \odot \in \{+, \cdot\}, \quad x \text{ with format } (m_x, \ell_x) \text{ and values in } [x;\tilde{x}] \text{ and } y \text{ with } [y;\tilde{y}] \text{ with } (m_y, \ell_y) \text{ as FPF.} \]

**Multiplication** \( z \leftarrow x \cdot y; \quad z \) will be in the interval \([z;\tilde{z}]\) with:

\[
\tilde{z} = \min(xy, \tilde{x}y, xy, \tilde{y}), \quad \bar{z} = \max(\tilde{z}, xy, \tilde{x}y, \tilde{y}). \quad (11)
\]

Its FPF \((m_z, \ell_z)\) is given by \( m_z = m_x + m_y + 1 \) and \( \ell_z = \ell_x + \ell_y \). Note that if the multiplier hardware operator is a \(w_y\)-bit multiplier \((w_y\)-bit operands and \(2w_y\) bit result) with \( w_x > w_y \) (or \( w_y > w_x \)), then \( x \) (or \( y \)) should be first right shifted of \( w_x - w_y \) bits \((w_y - w_x \) bits respectively) before performing the operation.

**Addition** \( z \leftarrow x + y; \) the addition is performed with a \(w_y\)-bit adder \((i.e. \text{ the operands are } w_x \text{ bits and the result is on } w_y \text{ bits})\) plus a carry bit that is not used here since there will be no overflow by construction), as follows:

1. the interval of the result is given by \( \tilde{z} = \tilde{x} + \tilde{y}, \quad \bar{z} = z + \bar{y} \)
2. its FPF \((m_z, \ell_z)\) with \(w_y\) bits is deduced from proposition 2 (so the addition is performed on format \((m_z, \ell_z)\) );
3. before the addition, \( x \) must be right shifted of \( \ell_x - \ell_z \) bits (or left shifted of \( \ell_z - \ell_x \) bits) and \( y \) of \( \ell_z - \ell_y \) bits.

Figure 5 summarizes the fixed-point addition.
Remark 2 Moreover, if the FPF of the final result $s$ is known by an other mean (like the one presented in Proposition 4), then the most significant bit of the partial result can be limited to the most significant bit of the final result. Even if an intermediate overflow occurs, it will be compensated in a next addition, thanks to Jackson’s Rule [13].

These two rules may be propagated from bottom-to-up in the AST, in order to obtain a fixed-point version of the initial AST.

3.4. Error analysis

In fixed-point arithmetic, any right shift produces an error, called 
roundoff error.

Proposition 3 (Error model) Let $x(k)$ be a signal with $(m, \ell)$ as FPF. Right shifting $x(k)$ of $d$ bits is equivalent to add an interval error $[e] = [e; \overline{e}]$ with:

$$
\begin{array}{c|c|c}
\text{Truncation} & \text{Round to the nearest} & \text{Range} \\
\hline
[e, \overline{e}] & [-2^{m+d} + 2^{-\ell}; 0] & [-2^{m+d-1} + 2^{-\ell}; 2^{m+d-1}] \\
\end{array}
$$

Proof: The interval error contains all the possible values formed with bits $x_{k+d-1}$ to $x_{\ell}$.

So, when FPF have been propagated with preceding rules through the whole SoP, all shift values are known, so as the interval errors added for each right shift. Indeed, all these interval errors can be regrouped in one unique error, with interval addition rule [12]. The error is propagated from bottom-to-top through the SoP, as well as the FPF are.

Finally, a Sum-of-Products implemented in fixed-point arithmetic is equivalent to the original infinite precision sum-of-products, corrupted by the addition of a roundoff error, that is the sum of all the roundoff errors occurring during the fixed-point evaluation.

Remark 3 The fixed-point format and roundoff error depends on the order the operations (additions here) are performed. There are $\prod_{i=1}^{n}(2t - 1)$ possible ordering for sum of $n$ terms. In [11], an heuristic has been built so as to deal with all these possible ordered SoP. In the following, the considered SoP are the optimal ordered SoP, according to the roundoff error and degree of parallelism.

4. APPLICATION TO THE SIF

This work on Sum-of-Products is now applied to linear filters/controllers with input-output algorithms expressed by the SIF.

In order to determine the FPF of the input(s), output(s), states and intermediate variables, the following definition and lemma are required:

Definition 2 (DC-gain and $\ell^\infty$-norm) Let $H$ be a Simple-Input-Simple-Output Bounded-Input-Bounded-Output filter, characterized by its transfer function $H(z)$. Denote $h(k)$ its impulse response, and $(A, b, c, d)$ a state-space realization of $H$.

Its DC-gain, denoted $|H|_{DC}$, is the low-frequency gain of the system, i.e. $\lim_{w \to 0} |H(e^{j\omega})|$. It can be computed by

$$
|H|_{DC} = c(I - A)^{-1}b + d \quad \text{or} \quad |H|_{DC} = H(1)
$$

Its worst-case peak gain, denoted $\|H\|_{\ell^\infty}$ is the largest possible peak value of the output $y(k)$ over all possible inputs $u(k)$:

$$
\|H\|_{\ell^\infty} = \sup_{k \geq 0} |y(k)|
$$

It can be computed with the $\ell^\infty$-norm of its impulse response: $\|H\|_{\ell^\infty} = \sum_{k \geq 0} |h(k)|$. It can also be computed from its state-space realization: $\|H\|_{\ell^\infty} = \sum_{k \geq 0} \|CA^kB + d\|_{\ell^\infty}$. This worst-case peak gain can be achieved with $u(k) = U \text{sign}(h(k))$, where $U$ is a constant and $\text{sign}$ the sign function returning $\pm 1$ or 0.

Remark 4 The infinite sum required to compute the $\ell^\infty$-norm can be bounded by two convergent finite sums, so that pratically $\|H\|_{\ell^\infty}$ can be evaluated at any required precision [14].

Lemma 1 (Interval through a linear filter) Let $c(k)$ be a scalar input going through a filter $H$, and denote $f(k)$ the resulting output. If $\forall k, c(k) \in (e_m, e_r)$, then $\forall k, f(k) \in (f_m, f_r)$ with:

$$
f_m = |H|_{DC} e_m \quad \text{and} \quad f_r = \|H\|_{\ell^\infty} e_r.
$$

Proof: Since $H$ is linear, then the output $f(k)$ is the sum of the constant term $e_m$ through the filter and a variable term in $[-e_r; e_r]$ through the filter. The constant term is scaled by the DC-gain of the filter whereas the bound on the second term is scaled by the worst-case gain using (14).

By extension, if $H$ is a Multiple Input Multiple Output filter ($p$ inputs, $q$ outputs, $(A, B, C, D)$ as state-space representation), then $\langle H \rangle$ denotes the worst-case peak gain matrix, i.e. each component of these matrix is the worst-case peak gain of the sub-system associated. $\langle H \rangle \in \mathbb{R}^{p \times q}$ such as $\|H\|_{\ell^\infty} = \sum_{k \geq 0} \|CA^kB + D\|_{\ell^\infty}$ with component-wise absolute value (i.e. $A|_{i,j} \triangleq |A_{i,j}|$).

Then, if the input vector $e(k)$ is component-wisely in $\langle e_m, e_r \rangle$, then $f(k)$ is component-wisely in $\langle f_m, f_r \rangle$ with

$$
f_m = |H|_{DC} e_m \quad \text{and} \quad f_r = \|H\|_{\ell^\infty} e_r.
$$

Proposition 4 Let the input $v(k)$ of the system be (component-wisely) in $\langle v_m, v_r \rangle$, then the output $y(k)$, states $x(k)$ and the intermediate variables $t(k)$ are (component-wisely) in $\langle y_m, y_r \rangle$, $\langle x_m, x_r \rangle$ and $\langle t_m, t_r \rangle$ respectively, with

$$
\begin{align*}
\begin{bmatrix} t_m \\ x_m \\ y_m \end{bmatrix} &= |H|_{DC} \begin{bmatrix} u_m \\ x_r \\ y_r \end{bmatrix} = \langle H \rangle \begin{bmatrix} u_m \\ x_r \\ y_r \end{bmatrix} \\
H_u : z &\to N_1 (zI_n - A_Z)^{-1} B_Z + N_2,
\end{align*}
$$
Fig. 6. Equivalent system, with errors separated

\[
N_1 \triangleq \begin{pmatrix} J^{-1} M \\ A_z \\ C_z \end{pmatrix}, \quad N_2 \triangleq \begin{pmatrix} J^{-1} N \\ B_z \\ D_z \end{pmatrix}.
\]

Then, Proposition 2 can be applied to deduce the FPF of each output, state and intermediate variable.

Proof: Eq (1) can be written as following:

\[
\begin{pmatrix} x(k+1) \\ y(k) \end{pmatrix} = \begin{pmatrix} J^{-1} M & J^{-1} N \\ A_z & B_z \\ C_z & D_z \end{pmatrix} \begin{pmatrix} x(k) \\ u(k) \end{pmatrix}
\]

so \( H_u \) defined in (18) is the transfer function from \( u(k) \) to \( \begin{pmatrix} x(k+1) \\ y(k) \end{pmatrix} \). Then Lemma 1 is applied to get (17).

As seen in section 3.3, the evaluation for each SoP in fixed-point arithmetic may provide an additional error. When implemented, the algorithm 1 is changed in

\[
J \cdot t(k+1) \leftarrow M \cdot x(k) + N \cdot u(k) + e_t(k) \\
x(k+1) \leftarrow K \cdot t(k+1) + P \cdot x(k) + Q \cdot u(k) + e_x(k) \\
y(k) \leftarrow L \cdot t(k+1) + R \cdot x(k) + S \cdot u(k) + e_y(k)
\]

Denote \( e(k) \) the vector with all the added errors \( e(k) \triangleq \begin{pmatrix} e_t(k) \\ e_x(k) \\ e_y(k) \end{pmatrix} \).

Proposition 5 It is then possible to express the implemented system as the initial system with an error \( e'(k) \) added on the output(s), as shown in Figure 6. \( e'(k) \) is the result of the error \( e(k) \) through the transfer function \( H_e \) defined by:

\[
H_e : z \rightarrow C_z (zI_n - A_z)^{-1} M_1 + M_2
\]

with

\[
M_1 \triangleq \begin{pmatrix} K & J^{-1} \\ I_n & 0 \end{pmatrix}, \quad M_2 \triangleq \begin{pmatrix} L & J^{-1} \\ 0 & I_p \end{pmatrix}.
\]

The error added on the output(s) is finally known to be (component-wisely) in the intervals \( (e'_m, e'_u) \), with

\[
e'_m = |H_e| \cdot D \cdot e_m, \quad e'_u = \langle H_e \rangle \cdot e_r
\]

Proof: \( H_e \) comes from a reformulation of equation (21), with \( e(k) \) as one input of the system. Then separability principle is applied, since the system \( H \) is linear, to get system decomposition shown in Figure 6.

5. EXAMPLE

As an illustrative example, the following transfer function to implement has been randomly chosen (with the Matlab drrs command):

\[
H(z) = \frac{0.4679z^4 - 1.535z^3 + 0.9085z^2 + 0.6147z - 0.6206}{z^4 - 0.3843z^3 - 0.7342z^2 + 0.1943z + 0.0589}
\]

The \( \rho \text{DFIIt} \) structure presented section 2.2 has been chosen. The coefficients \( \{ \gamma_i \} \) have been chosen so as to minimize the final output error (this optimization process is not detailed here, due to lack of place. With few coefficients, exhaustive search can also be done [6]):

\[
\gamma = ( -0.1224 \quad 0.3881 \quad -0.7620 \quad 0.8808 )^T.
\]

Algorithm 2 exhibits the computations to perform.

Input: \( u \): the input at step \( k \)
Output: \( y \): the output at step \( k \)
begin

// Intermediate variables
\( t \leftarrow x_0 + 0.467892u \);
// Output
\( y \leftarrow t \);
// states
\( x_0 \leftarrow -0.122366x_0 - 1.35548u - 0.00029146t + x_1 \); \( x_1 \leftarrow 0.388137x_1 + 0.542843u + 0.046928t + x_2 \);
\( x_2 \leftarrow -0.762002x_2 - 0.254215u - 0.00485693t + x_3 \);
\( x_3 \leftarrow 0.886823x_3 - 0.141993u + 0.00271706t \);
\end

The input \( u \) is supposed to be bounded in \([-10; 10]\) and represented with 16 bits, so Proposition 4 proposes to determine the magnitude of each involved variable in the system, and thus their FPF, assuming 16-bit words. Here, the worst-case matrix of \( H_u \) is given by

\[
\| H_u \|_{\infty} = ( 3.775 \quad 3.307 \quad 1.780 \quad 0.991 \quad 1.191 \quad 3.775 )^T
\]

so the input \( u \), the output \( y \) and the intermediate variable \( t \) have \((4, -11), (6, -9)\) and \((6, -9)\) as FPF respectively. The four states have the following FPF: \((6, -9), (5, -10), (4, -11)\) and \((4, -11)\).

The FPF propagation and error analysis shown in section 3.3 have been applied to convert the four SoP in fixed-point arithmetic, assuming \(16 \times 16\) bits multiplications and 32 bits additions. Figure 7 exhibits the fixed-point AST for the computation of \( x_2 \), and Algorithm 3 is the fixed-point version of Algorithm 2 (truncation has been chosen as rounding mode).

From the five SoPs, the following error can be deduced:

\[
\xi = \begin{pmatrix} -0.0019531 \\ -0.0019531 \\ -0.0009765 \\ -0.0004882 \\ -0.0004882 \end{pmatrix}, \quad \bar{\xi} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.
\]
so, with Proposition 5, the output error \( e' \) can be known to be in 
\(-0.0105, 0.000928\).

On Figure 8 is plotted the error \( e'(k) = y'(k) - y(k) \) for a white noise input \( u(k) \) in \([-10, 10]\). The error between the double floating result and the fixed-point one is relatively small and in the bound predicted by the theory. The mean of the error is around 0.05 that is around the middle of error interval. Of course, the \( \epsilon_\infty \)-norm is a conservative norm (but still achievable), but it gives the worst case error performed by the computations. More specifically, it can give us how many bits of the results are reliable (with faithful rounding).

**Algorithm 3: Fixed-point algorithm**

All these results have been obtained with our tool named Fi-PoGen (as Fixed-Point Generator) written in Python. Its code will be soon released under open-source licence.

---

### 6. CONCLUSION

Throughout this paper, a more precise method than the one previously introduced in [11] has been proposed to describe automatic fixed-point conversion of linear signal processing or control algorithms. Based on interval arithmetic and linear filter properties, the error analysis shown for Sum-of-Products have been extended to every structure of linear filters/controllers. Additional work includes code generation for software implementation (algorithm-to-architecture mapping) and hardware implementation (word-length optimization so as to minimize implementation cost such as area under constraints on the output error).

### 7. REFERENCES


