Toward the synthesis of fixed-point code for matrix inversion based on Cholesky decomposition

Matthieu Martel[1,2,3] Amine Najahi[1,2,3] Guillaume Revy[1,2,3]
[1] Univ. Perpignan Via Domitia, DALI, F-66860, Perpignan, France
[2] Univ. Montpellier II, LIRMM, UMR 5506, F-34095, Montpellier, France
[3] CNRS, LIRMM, UMR 5506, F-34095, Montpellier, France

***Abstract*—Matrix inversion is a computationally intensive basic block of many digital signal processing algorithms. To decrease the cost of their implementations, programmers often prefer the fixed-point arithmetic. This arithmetic requires less resources and runs faster than the floating-point arithmetic, but all the arithmetical details must be handled by the programmer. In this article, we overcome this drawback by presenting an automated approach to synthesize fixed-point code for matrix inversion based on Cholesky decomposition. First we rigorously define the square root and division operators especially in terms of rounding error, and we implement them in the CGPE library. This allows us to provide accuracy certificates for the generated code. Second we propose a workflow based on Cholesky decomposition that carefully uses these operators to produce accurate code for the basic blocks of matrix inversion. Finally we illustrate the efficiency of our approach on some benchmarks, and show how it allows us to synthesize accurate code in a few seconds and thus to reduce the development time of fixed-point matrix inversion.**

Keywords: **Cholesky decomposition, matrix inversion, fixed-point arithmetic, automated code synthesis, certified numerical accuracy**

I. Introduction

Matrix inversion is known to be numerically unstable. As a consequence, numerical analysts advise against using it for a large set of problems. Yet, as stated by Higham [1, Ch. 14], cases exist where the inverse conveys useful information. In the wireless communication field, matrix inversion is used in equalization algorithms [2] as well as detection estimation algorithms in space-time coding [3]. This article does not discuss these cases and leaves it to the programmer to decide whether computing the inverse is indeed justified. The goal of this article is to introduce techniques that help the programmer in the process of writing fixed-point code for matrix inversion. The originality of our work relies on the fact that certified error bounds are also produced together with the code. In a typical design, DSP programmers prototype and simulate their algorithms in high level environments like MATLAB. These environments work with the floating-point arithmetic [4] to ease and speedup the prototyping phase. However, when mapping the design to hardware, constraints on silicon area, power consumption, or throughput frequently force the implementer to convert this design to the more efficient fixed-point arithmetic [5]. This conversion is known to be a tedious and time consuming process [6] that may be split into two phases:

1) Range analysis: This phase allows to find the integer wordlength of each variable in the design. In a finite wordlength environment, minimizing the integer word length allows one to allocate more digits for the fractional part, thus obtaining more accuracy.
2) Precision analysis: In this phase, the number of bits to allocate to the fractional part is decided. This phase must take into account the precision needs of the application.

Over the last years, authors have suggested different strategies to tackle these conversion phases. These contributions fit into two categories:

1) Simulation based strategies [7], [8]: The information that allows to estimate the required range and precision are inferred from intensive simulations carried out using a precise arithmetic, typically floating-point arithmetic.
2) Analytic strategies [9], [10]: The information is obtained using formal methods such as interval arithmetic, affine arithmetic, and norm computation for digital filters. The precision analysis phase relies on optimization techniques.

In this work, we use an analytic approach based on interval arithmetic to bound the range of the variables in our design and to give strict bounds on the rounding errors. Previous works applied similar techniques but only to a small range of problems involving only additions and multiplications. This article describes algorithms to compute square roots and divisions, and the means to bound their rounding errors. By adding these operators to the CGPE tool, we are able to treat problems, previously considered intractable in fixed-point arithmetic, like Cholesky decomposition and triangular matrix inversion.

This article is organized as follows: Section II gives some insights on matrix inversion algorithms. Section III introduces some basics on fixed-point arithmetic, while Section IV details our new square root and division operators. Section V details the workflow we propose to map matrix inversion into the fixed-point arithmetic. Then some experimental results are exhibited in Section VI, before concluding in Section VII.

II. Background on matrix inversion

A survey of floating-point matrix inversion and linear systems solving shows that there are many algorithms in use: LU and QR decompositions, Cramer's rule, ... [11]. A common pattern to the efficient algorithms is the decomposition of the input matrix into a product of easy to invert matrices (triangular or orthogonal matrices). LU decomposition for instance, proceeds by Gaussian elimination to decompose an input matrix A into two triangular matrices L and U such that $A = LU$. Inverting triangular matrices being straightforward, solving the associated linear system is equally simple and so is obtaining the inverse A^{-1} by the formula $A^{-1} = U^{-1}L^{-1}$. Almost the same chain of reasoning is applicable to QR decomposition. Clearly, the computationally intensive step of these algorithms is the decomposition part [11].

The decomposition part is also the missing link in fixed-point arithmetic. Indeed, some works exist on fixed-point code synthesis for matrix multiplication [12], but, to the authors knowledge, no published material suggests a rigorous methodology of code synthesis for matrix decomposition or triangular matrix inversion. The authors of the Gusto tool [13] do provide benchmarks for the accuracy of their matrix inversion algorithms. However, the methodology for generating the fixed-point implementation is not explicit in the article and the treated matrices do not exceed size 8. Besides, the evaluation of the rounding error is an *a posteriori* estimation process. Therefore, no absolute error bounds are provided. In [7], Frantz *et al.* use an approach based on the ideas introduced by [8] to study the mapping of many linear algebra routines to fixed-point DSPs. They treat matrices of sizes up to 35, but uniquely with simulation based methods and without providing certificates on the error bounds. In introducing a rigorous error model, our work is analogous to [9] and [10], where affine arithmetic is used to assert properties on the error bounds. However, both approaches target discrete transformation algorithms and only [9] treats briefly of small size matrix multiplication. In our approach, we suggest an interval arithmetic based formalization of all the arithmetic operators that intervene in the computation of classical linear algebra decompositions. This allows us to compute explicit bounds on the rounding errors of the codes synthesized by our tools. In the following of this section, we recall the formulas for triangular matrix inversion and Cholesky decomposition, which are useful to solve linear systems and invert matrices.

A. Triangular matrix inversion

Using the so called backward and forward substitution techniques, inverting a triangular matrix is a straightforward process in floating-point arithmetic. Indeed, for a lower triangular matrix A, its inverse N is given by the following equations:

$$n_{i,j} = \begin{cases} \dfrac{1}{a_{i,i}} & \text{if } i = j \\ \dfrac{-c_{i,j}}{a_{i,i}} & \text{if } i \neq j \end{cases} \quad \text{where } \; c_{i,j} = \sum_{k=j}^{i-1} a_{i,k} \cdot n_{k,j}. \quad (1)$$

In floating-point arithmetic, these equations are easy to implement. It is less so in fixed-point arithmetic, since the coefficient $n_{i,j}$ depends on other coefficients of N, namely all the $n_{k,j}$ with $k \in \{j, \ldots, i-1\}$. This implies that the synthesis tool, when generating code that computes $n_{i,j}$ must know the ranges and formats of all the $n_{k,j}$ with $k \in \{j, \ldots, i-1\}$. It is clear that such a tool must follow a determined order in synthesizing code and that it must keep track of the formats and ranges of the computed coefficients so as to reuse them. Besides, similarly to the work in [12], this process involves multiple trade-offs between code size and accuracy.

B. Cholesky decomposition

If A is a symmetric positive definite matrix, its Cholesky decomposition is defined. Since this method exploits the structure of the matrix, it is more efficient than Gaussian elimination. It is also known for its numerical stability. The aim of Cholesky's method is to find a lower triangular matrix L such that

$$A = L \cdot L^T. \quad (2)$$

By equating the coefficients in (2) and using the properties of A, the following two formulas for the general term of L are deduced:

$$\ell_{i,j} = \begin{cases} \sqrt{c_{i,i}} & \text{if } i = j \\ \dfrac{c_{i,j}}{\ell_{j,j}} & \text{if } i \neq j \end{cases} \quad \text{with } \; c_{i,j} = a_{i,j} - \sum_{k=0}^{j-1} \ell_{i,k} \cdot \ell_{j,k}. \quad (3)$$

Generating code for $c_{i,j}$ is easily achievable in fixed-point arithmetic since it involves a subtraction and a size-j dot product. Synthesizing code that computes $\ell_{i,j}$ from $c_{i,j}$ is less obvious since the square root and division operators are not well studied in fixed-point arithmetic. To conclude this section, note that Cholesky decomposition can be used to invert any non symmetric positive definite matrix A by decomposing the following matrix: $M = AA^T$ which is guaranteed to be symmetric to obtain $M = LL^T$. From the decomposition, A^{-1} can be recovered thanks to the formula: $A^{-1} = A^T L^{-T} L^{-1}$.

III. Basics on fixed-point arithmetic

This section presents some basics on fixed-point arithmetic. It explains the means to implement and bound the error of fixed-point addition/subtraction, multiplication, and shift, the most useful operators in DSP algorithms. The reader is encouraged to refer to [5] or [14] for a more detailed introduction to fixed-point arithmetic.

A. Fixed-point number representation

Fixed-point arithmetic allows to represent a real value by means of an integer associated to an *implicit scaling factor*. Let X be a k-bit signed integer in radix 2, encoded using two's complement notation. Combined with a factor $f \in \mathbb{Z}$, it represents the real value x defined as follows:

$$x = X \cdot 2^{-f}.$$

In the sequel of this article, $\mathbf{Q}_{i.f}$ denotes the *format* of a given fixed-point variable v represented using a k-bit integer associated with a scaling factor f, with $k = i + f$. Here i and f denote the number of bits in the *integer* and *fraction* part of v, respectively, while k represents its *wordlength*. Hence v is such that:

$$v \in \{V \cdot 2^{-f}\} \quad \text{with} \quad V \in \mathbb{Z} \cap [-2^{k-1}, 2^{k-1} - 1]$$

by step of 2^{-f}.

B. Arithmetic and error model

Given an arithmetic operation $\diamond \in \{+, -, \times, <<, >>\}$, to bound the range of the resulting value and the rounding error entailed by its computation, we keep track for each fixed-point variable v, of the three following intervals:

1) $\mathbf{Val}(v)$ enclosing the values of v computed at run-time with finite precision,
2) $\mathbf{Math}(v)$ enclosing the values taken by v if computations were done with infinite precision,
3) $\mathbf{Err}(v)$ enclosing the rounding errors occurring while computing v,

such that:

$$\mathbf{Err}(v) = \mathbf{Math}(v) - \mathbf{Val}(v).$$

Notice that all the computations involving $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$ are done using interval arithmetic [15]. And thanks to the formula above, keeping track of $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$ suffices to deduce $\mathbf{Math}(v)$.

To show how $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$ are computed, let us define three fixed-point variables v, v_1, and v_2 along with their formats $\mathbf{Q}_{i.f}$, $\mathbf{Q}_{i_1.f_1}$, and $\mathbf{Q}_{i_2.f_2}$, respectively. When $\diamond \in \{+, -, \times\}$ and $v = v_1 \diamond v_2$, we have:

$$\mathbf{Val}(v) = \mathbf{Val}(v_1) \diamond \mathbf{Val}(v_2) - \mathbf{Err}_\diamond.$$

For physical shifts, that is, for $\diamond \in \{<<, >>\}$, we simply have:

$$\mathbf{Val}(v) = \mathbf{Val}(v_1) - \mathbf{Err}_\diamond,$$

since these operators do not modify the value of the input v_1 but both the bit string of V_1 and its fixed-point format. Interpreting v_1 in a different fixed-point format without modifying the bit string of V_1 is called virtual shift[1] in [14]: this operation is error-free, and it is the only shift operation that modify the value of v_1:

$$\mathbf{Val}(v) = \mathbf{Val}(v_1) \times 2^{f_1 - f}.$$

Addition and subtraction. In absence of overflow, addition and subtraction are error-free. Hence for $\diamond \in \{+, -\}$ we have:

$$\mathbf{Err}(v) = \mathbf{Err}(v_1) \diamond \mathbf{Err}(v_2) \quad \text{and,}$$

$$i = \max(i_1, i_2) + 1 \quad \text{and} \quad f = \max(f_1, f_2).$$

Multiplication. If $\diamond$ is a multiplication, we have:

$$\begin{aligned} \mathbf{Err}(v) &= \mathbf{Err}_\times + \mathbf{Err}(v_1) \cdot \mathbf{Err}(v_2) \\ &+ \mathbf{Err}(v_1) \cdot \mathbf{Val}(v_2) + \mathbf{Val}(v_1) \cdot \mathbf{Err}(v_2), \end{aligned}$$

where $\mathbf{Err}_\times$ is the error entailed by the multiplication itself. Remark that exact fixed-point multiplication results in a number having a fraction of $f_1 + f_2$ bits. And usually in fixed-point arithmetic the error $\mathbf{Err}_\times$ is due to the truncation of the exact result of the multiplication to fit in a smaller format with $f < f_1 + f_2$ fraction bits. Hence we have:

$$\mathbf{Err}_\times = [0, 2^{-f} - 2^{-(f_1+f_2)}].$$

Most 32-bit DSP processors provide a 32×32 multiplier that returns the 32 most significant bits of the exact result, which is the multiplier considered in this work. In that case,

$$i = i_1 + i_2 \quad \text{and} \quad f = 32 - i.$$

Left and right shift. If $\diamond \in \{<<, >>\}$, we have:

$$\mathbf{Err}(v) = \mathbf{Err}(v_1) + \mathbf{Err}_\diamond.$$

Left shifts of s bits entail no error but only a possible overflow: $\mathbf{Err}_{<<} = [0, 0]$ and $(i, f) = (i_1 - s, f_1 + s)$. However right shifts of s bits may be followed by a truncation to fit the result in a smaller format with $f < f_1$ fraction bits. Thus, we have:

$$(i, f) = (i_1 + s, f_1 - s) \quad \text{and} \quad \mathbf{Err}_{>>} = [0, 2^{-f_1+s} - 2^{-f_1}].$$

[1]This operation is not detailed in our article, but we encourage the reader to refer to [14] for more details.

IV. Fixed-point square root and division

Many research articles have been published that deal with the four fixed-point operations presented in Section III [14], [16], [17], [18]. Square root and division did not receive as much attention, and prior to this work, we are not aware of any published material that formalizes rigorously square root and division in fixed-point arithmetic, by suggesting an implementation together with an error model. Yet square root is useful when computing euclidean norms. In a linear algebra context, Cholesky decomposition and triangular inverse involve square roots and divisions, as shown in Section II. Division may also be used to evaluate rational polynomial approximants [19]. The following of the section presents our new fixed-point square root and division operators.

A. Fixed-point square root

This first part presents the operator of fixed-point square root we have implemented. Assuming $v_1 \geq 0$, for $v = \sqrt{v_1}$, we have:

$$\mathbf{Val}(v) = \sqrt{\mathbf{Val}(v_1)} - \mathbf{Err}_\surd$$

since the computed value is truncated, while $\mathbf{Err}(v)$ is:

$$\mathbf{Err}(v) = \sqrt{\mathbf{Math}(v_1)} - \sqrt{\mathbf{Val}(v_1)} + \mathbf{Err}_\surd,$$

where $\mathbf{Err}_\surd$ is the error entailed by the square root operation itself. The error term is given by the following formula:

$$\begin{aligned} \mathbf{Err}(v) &= \sqrt{\mathbf{Val}(v_1) + \mathbf{Err}(v_1)} - \sqrt{\mathbf{Val}(v_1)} + \mathbf{Err}_\surd \quad (4) \\ &= \sqrt{\mathbf{Val}(v_1)} \cdot \left(\sqrt{1 + \frac{\mathbf{Err}(v_1)}{\mathbf{Val}(v_1)}} - 1 \right) + \mathbf{Err}_\surd. \end{aligned}$$

Here the factorization is used to remedy the *interval dependency* phenomenon inherent to interval arithmetic.

Notice that this formula does not yield tight error bounds as soon as $\mathbf{Val}(v_1)$ smallest elements are of the same order of magnitude than $\mathbf{Err}(v_1)$. To overcome this issue, we may use the subadditivity property of the square root function, which holds as long as x and $x + y$ are both positive:

$$\sqrt{x} - \sqrt{|y|} \leq \sqrt{x + y} \leq \sqrt{x} + \sqrt{|y|}.$$

Hence we deduce the following bounds on $\mathbf{Err}(v)$:

$$\mathbf{Err}_\surd - \sqrt{|\mathbf{Err}(v_1)|} \leq \mathbf{Err}(v) \leq \mathbf{Err}_\surd + \sqrt{|\mathbf{Err}(v_1)|}. \quad (5)$$

In practice, we compute the intersection of the enclosures (4) and (5).

As for $\mathbf{Err}_\surd$, it depends on the square root algorithm. To explicit such an algorithm, let us remember that we have $v_1 = V_1 \cdot 2^{-f_1}$. In the following, we propose two different algorithms together with a piece of C code to perform fixed-point square root, and we compare them by exhibiting their error bounds.

Algorithm 1. A first attempt to implement square root is to start from the following rewriting:

$$\sqrt{v_1} = \begin{cases} \sqrt{V_1} \cdot 2^{-f_1/2} & \text{if } f_1 \text{ is even,} \\ \sqrt{V_1/2} \cdot 2^{-(f_1-1)/2} & \text{if } f_1 \text{ is odd.} \end{cases}$$

An implementation of this approach would compute $\sqrt{v_1}$ using one of the following:

$$\lfloor\sqrt{V_1}\rfloor \cdot 2^{-f_1/2} \quad \text{or} \quad \lfloor\sqrt{V_1/2}\rfloor \cdot 2^{-(f_1-1)/2}$$

where $\lfloor\sqrt{\cdot}\rfloor$ is the *integer square root* operation. This operation may be implemented in hardware or in software using multiple techniques such as digit-recurrence, and Newton-Raphson or Goldschmidt iteration [20], [21]. We deduce that the value $\sqrt{v_1}$ has $\approx f_1/2$ fraction bits, and that

$$\mathbf{Err}_{\sqrt{}} = [0, 2^{-\frac{f_1}{2}}] \quad \text{or} \quad \mathbf{Err}_{\sqrt{}} = [0, 2^{-\frac{f_1-1}{2}}]$$

depending on the parity of f_1. Compared to the error bound of the previous operations, an error bound of $\approx 2^{-f_1/2}$ for the square root is not acceptable in practice.

Algorithm 2. To overcome the accuracy issue of Algorithm 1 above, let use rewrite v_1 as

$$v_1 = 2^{\eta} \cdot V_1 \cdot 2^{-(f_1+\eta)}$$

with the integer η being a parameter of the algorithm chosen at synthesis-time such as $f_1 + \eta$ is even. Using this scaling factor, it follows that

$$\sqrt{v_1} = \sqrt{2^{\eta} \cdot V_1} \cdot 2^{\frac{-(f_1+\eta)}{2}}. \tag{6}$$

The cases $\eta = 0$ and $\eta = -1$ correspond to the even and odd cases of Algorithm 1, respectively. An algorithm that exploits (6) shifts the integer representation V_1 of v_1 by η bits to the left and computes its integer square root. The result of this algorithm is a fixed-point variable with $(f_1+\eta)/2$ bits of fraction part. Hence using this approach, we conclude that

$$i = \lceil i_1/2 \rceil, \quad f = \frac{f_1+\eta}{2}, \quad \text{and} \quad \sqrt{v_1} = \left\lfloor \sqrt{2^{\eta} \cdot V_1} \right\rfloor \cdot 2^{\frac{-(f_1+\eta)}{2}},$$

where $\left\lfloor \sqrt{2^{\eta} \cdot V_1} \right\rfloor$ is computed using an integer square root operation. It is clear now that with such an algorithm, we obtain the following bound on $\mathbf{Err}_{\sqrt{}}$:

$$\mathbf{Err}_{\sqrt{}} = \left[0, 2^{-\frac{(f_1+\eta)}{2}}\right].$$

Notice that it would not make sense to choose $\eta < 0$, since it would result in an increase of $\mathbf{Err}_{\sqrt{}}$. Hence in the following of the section, we assume $\eta \geq 0$.

Listing 1 gives a C implementation for a $32 \rightarrow 32$-bit square root operation using Algorithm 2 based on digit-recurrence iteration when $0 \leq \eta < 32$. This is by far the most frequent case in practice. In this code, the 32 least significant bits of the variable `V` are returned. Hence the parameter η must be carefully chosen, to ensure that $f_1 + \eta$ is even, and that the wordlength of the result is at most 32, otherwise an overflow may occur.

```
uint32_t isqrt32hu(uint32_t V1, uint16_t eta)
{
  uint64_t V1_extended = ((uint64_t)V1) << eta; // eta >= 0
  uint64_t V = 0;
  int64_t  one = 0x4000000000000000ll;          // 2^(62)

  while (one != 0){
    if (V1_extended >= V + one){
      V1_extended = V1_extended - (V + one);
      V = V + (one << 1);
    }
    V >>= 1;
    one >>= 2;
  }
  return (uint32_t)V;
}
```

Listing 1. C code of $32 \rightarrow 32$-bit fixed-point square root operation.

B. Fixed-point division

This second part presents our fixed-point division operator. Let $v = v_1/v_2$. When the quotient is defined, *i.e.* when $v_2 \neq 0$, that is, $0 \notin \mathbf{Val}(v_2)$, we have:

$$\mathbf{Val}(v) = \frac{\mathbf{Val}(v_1)}{\mathbf{Val}(v_2)} - \mathbf{Err}_{/}$$

while $\mathbf{Err}(v)$ is defined as:

$$\mathbf{Err}(v) = \frac{\mathbf{Math}(v_1)}{\mathbf{Math}(v_2)} - \frac{\mathbf{Val}(v_1)}{\mathbf{Val}(v_2)} + \mathbf{Err}_{/},$$

where $\mathbf{Err}_{/}$ is the error entailed by the division itself. It follows that the error term is defined as:

$$\mathbf{Err}(v) = \frac{\mathbf{Val}(v_2) \cdot \mathbf{Err}(v_1) - \mathbf{Val}(v_1) \cdot \mathbf{Err}(v_2)}{\mathbf{Val}(v_2) \cdot (\mathbf{Val}(v_2) + \mathbf{Err}(v_2))} + \mathbf{Err}_{/}.$$

From a theoretical point of view, the quotient v_1/v_2 when defined must be a fixed-point variable in the format $\mathbf{Q}_{i.f}$ with

$$i = i_1 + f_2 \quad \text{and} \quad f = f_1 + i_2$$

since:

1) The largest possible dividend is -2^{i_1-1} while the smallest divisor is 2^{-f_2}. Thus the quotient could be as large as $-2^{i_1+f_2-1}$.
2) As for the opposite case, the smallest dividend is 2^{-f_1} while the largest divisor is -2^{i_2-1}. To be precise, the fractional part must be of size $f_1 + i_2$.

Remark that a special care must be taken when $0 \in \mathbf{Val}(v_2)$ to avoid division by zero. In this case, we first compute both error and value bounds twice, using the two intervals $\underline{\mathbf{Val}(v_2)}$ and $\overline{\mathbf{Val}(v_2)}$, such that:

$$\underline{\mathbf{Val}(v_2)} \cup \overline{\mathbf{Val}(v_2)} = \mathbf{Val}(v_2) \setminus \{0\}.$$

Then we compute the union of the resulting intervals.

As for the square root, the fixed-point format of v depends on the algorithm implemented. In the following, we suggest two different algorithms to perform fixed-point division together with a piece of C code, and we analyze the difference between these algorithms by exhibiting their error bounds. To do this, let us remember that we have $v_1 = V_1 \cdot 2^{-f_1}$ and $v_2 = V_2 \cdot 2^{-f_2}$.

Algorithm 1. A first attempt to implement division is to start from the following rewriting:

$$\frac{v_1}{v_2} = \frac{V_1 \cdot 2^{-f_1}}{V_2 \cdot 2^{-f_2}} = \frac{V_1}{V_2} \cdot 2^{-(f_1-f_2)}.$$

This algorithm computes the quotient of the two integers V_1 and V_2, and then considers $f_1 - f_2$ as the new implicit scale factor. The C standard requires integer division to be computed by discarding the fractional part of the exact division result, even for nonpositive results, that is, by rounding the exact result toward zero [22, § 6.5.5]. Since our implementations are intended to be compliant with the C language, we chose to implement a division operator satisfying this requirement. We denote by $\texttt{trunc}(\cdot)$ this operation. Hence:

$$\frac{v_1}{v_2} = \texttt{trunc}\left(\frac{V_1}{V_2}\right) \cdot 2^{-(f_1-f_2)}.$$

Below we give an implementation of this method using the C standard integer division. If this option is not available or is too costly, this operation may also be implemented in hardware or in software using digit-recurrence, and Newton-Raphson or Goldschmidt iteration. In this case, the error $\mathbf{Err}_/$ is as follows:

$$\mathbf{Err}_/ = [-2^{-(f_1-f_2)}, 2^{-(f_1-f_2)}].$$

It follows that the result has no more fraction bits, as soon as v_1 and v_2 have the same fixed-point format. This leads to a huge loss of accuracy since $\mathbf{Err}_/ = [-1, 1]$.

Algorithm 2. To overcome the accuracy issue of Algorithm 1 above, we use a technique similar to the one presented for square root, by introducing a scaling factor η. Hence

$$\frac{v_1}{v_2} = \frac{V_1 \cdot 2^{-f_1}}{V_2 \cdot 2^{-f_2}} = \frac{V_1 \cdot 2^{\eta}}{V_2} \cdot 2^{-(f_1-f_2+\eta)}.$$

Usually, an implementation of this algorithm computes

$$\frac{v_1}{v_2} = \mathtt{trunc}\left(\frac{V_1 \cdot 2^{\eta}}{V_2}\right) \cdot 2^{-(f_1-f_2+\eta)}$$

which results in a variable having the fractional part

$$f = f_1 - f_2 + \eta.$$

In our context, we consider that all the variables have the same wordlength, particularly $i + f = i_1 + f_1$. It follows that the integer part of the result is:

$$i = i_1 + f_2 - \eta. \tag{7}$$

Then we deduce that the error $\mathbf{Err}_/$ is as follows:

$$\mathbf{Err}_/ = [-2^{-(f_1-f_2+\eta)}, 2^{-(f_1-f_2+\eta)}].$$

Remark that even when v_1 and v_2 have the same format, $\mathbf{Err}_/ = [-2^{-\eta}, 2^{-\eta}]$ remains tight as long as η is large enough. Also as for square root, it would not make sense to choose $\eta < 0$, since it would result in an increase of $\mathbf{Err}_/$. Again in the following of the section, we assume $\eta \geq 0$. Listing 2 gives a C implementation for a $32 \times 32 \rightarrow 32$-bit division operation using Algorithm 2 when $0 \leq \eta < 64$. This code returns the

```
int32_t div32hs(int32_t V1, int32_t V2, uint16_t eta)
{
  int64_t t1 = ((int64_t)V1) << min(32, eta);  // eta >= 0
  int64_t t2 = ((int64_t)V2);
  int64_t V  = (t1 / t2) << max(0, eta-32);
  return (int32_t) V;
}
```

Listing 2. C code of $32 \times 32 \rightarrow 32$-bit fixed-point division operation.

32 least significant bits of the variable V. It must be clear to the reader that here again the parameter η must be chosen carefully, since it greatly influences the result by impacting its integer part i. Indeed picking a large η leads to a smaller value i than the theoretical one and it minimizes the error bound and ensures more accuracy on the result. However, by doing so, we suppose that the result is not large enough. More precisely, this means that the largest values in magnitude eventually taken by the result are ignored and discarded. This approach is equivalent to discarding the smallest values eventually taken by the variable v_2 in $\mathbf{Val}(v_2)$, that is, the values around 0 in $\mathbf{Val}(v_2)$. This may also be taken in consideration when computing $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$. In our experiments, we have implemented various heuristics to decide the parameter η.

V. Synthesis tool for matrix inversion

In this section, we detail the synthesis tool we have implemented for the particular case of matrix inversion.

A. Cholesky decomposition based approach

As mentioned earlier, we choose to implement matrix inversion through Cholesky decomposition. Given an input matrix A, it is done in three steps:

1) matrix decomposition: L such as $A = LL^T$,
2) triangular matrix inversion: L^{-1},
3) and matrix multiplication: $A^{-1} = L^{-T}L^{-1}$.

In fixed-point arithmetic, we can frequently reduce the ranges of the inputs to a range included in $[-1, 1]$. Indeed, for Cholesky decomposition, if a matrix A does not satisfy this condition, instead of decomposing A, we can decompose

$$B = 2^{-k} \cdot A \text{ with } k \in \mathbb{Z}, \tag{8}$$

to obtain $B = RR^T$, where B is a matrix with coefficients in $[-1, 1]$. It follows that $A = 2^k \cdot RR^T = LL^T$ where

$$L = 2^{k/2} \cdot R.$$

Notice that k must be chosen even. For triangular matrix inversion, we can still compute A^{-1} as follows:

$$A^{-1} = 2^{-k} \cdot B^{-1}, \text{ where } B \text{ is as in (8).}$$

In fixed-point arithmetic, these scalings are just a matter of manipulating the fixed-point format of the coefficients.

We have automated this in the FPLA tool[2] which relies on the CGPE library.[3] These tools are detailed below.

CGPE (Code Generation for Polynomial Evaluation). Developed by Revy and Mouilleron [16], CGPE is dedicated to the automated synthesis of fast and accurate fixed-point codes to evaluate mathematical expressions, like polynomials, dot products, and summations. It takes as input intervals that bound the range of each variable that appears in the expression, and produces a code which is able to evaluate the input expression for any instance of its arguments as long as they are in the original interval of values. Initially devoted to evaluating expressions involving only addition/subtraction, multiplication, and shift, CGPE has been here enhanced by adding the square root and division operators presented in Section IV.

FPLA (Fixed-Point Linear Algebra). We developed this tool with the aim of generating fixed-point code for the most frequently used linear algebra routines. It handles the aspects peculiar to each class of input problems and relies on the CGPE library for the low level and code synthesis details. For instance, when prompted to generate code for matrix multiplication, it is capable of finding good trade-offs between code size and numerical accuracy as in [12], by using the same code for output coefficients obtained from input rows and columns with elements having close enough fixed-point formats. For triangular matrix inversion and Cholesky decomposition, since some of the output coefficients are dependant on previously generated code, this strategy is harder to apply. Hence in these

[2] An archive containing the code of the FPLA tool is available upon request.
[3] CGPE is freely available at http://cgpe.gforge.inria.fr/.

cases, FPLA takes as input the intervals that bound the ranges of the coefficients of the input matrix and works by making successive calls to CGPE with the appropriate parameters. It iterates on the output matrix and asks for a code to compute a given output coefficient, defined by a specific expression as in (1) or (3). Once this code generated by CGPE, FPLA stores the range and error of its output. Note that FPLA also takes care of correctly ordering the calls to CGPE in such a way that each coefficient's code is generated only after all the information on which it depends has been collected. In this sense, FPLA's mode of operation greatly depends on the linear algebra problem to be solved.

B. Order of code synthesis in FPLA

In triangular inversion, the diagonal elements do not depend on any generated code. Therefore they may be computed in any order. The non diagonal coefficients depend only on the coefficients that precede them on the same column. FPLA must therefore follow a row major, column major or even a diagonal major approach. The latter consists in generating the elements on the diagonal, followed by those on the first sub-diagonal and so on. The last code generated in this fashion would be that of the bottom left coefficient $\ell_{n-1,0}$. For Cholesky decomposition, a diagonal element $\ell_{i,i}$ depends on the generated coefficients that precede it on row i. A non diagonal element $\ell_{i,j}$ depends on the first j elements of row i as well as the first $j+1$ elements of row j. FPLA may satisfy these dependencies by following either a row major or column major synthesis strategy but not a diagonal major strategy.

Listing 3 gives the global code produced by FPLA for a size-3 triangular matrix inversion. The coefficients of the upper triangular part are explicitly set to zero. Then `compute_i_i` computes the coefficient $n_{i,i}$ as $n_{i,i} = 1/d$ while `compute_i_j` computes the coefficient $n_{i,j}$ as

$$n_{i,j} = (a_0 \cdot b_0 + \cdots + a_{i-j} \cdot b_{i-j})/d$$

where a_0 =`A[i][j]`, b_0 =`N[j][j]`, a_{i-j} =`A[i][i-1]`, b_{i-j} =`N[i-1][j]`, and d =`A[i][i]`. The six `compute_x_y` functions of this example are each generated by CGPE in a specific C file. Listing 4 shows the code of `compute_2_0` in Listing 3.

```
// A: input matrix  --  N: inverse matrix of A
N[0][0] =  compute_0_0( A[0][0] );
N[0][1] =  0;
N[0][2] =  0;

N[1][0] = -compute_1_0( A[1][0], N[0][0], A[1][1] );
N[1][1] =  compute_1_1( A[1][1] );
N[1][2] =  0;

N[2][0] = -compute_2_0( A[2][0], A[2][1],
                        N[0][0], N[1][0], A[2][2] );
N[2][1] = -compute_2_1( A[2][1], N[1][1], A[2][2] );
N[2][2] =  compute_2_2( A[2][2] );
```

Listing 3. FPLA ouput code for a 3×3 triangular matrix inversion.

```
int32_t compute_2_0( int32_t a0 /* Q1.31 in [-1,1] */ ,
                     int32_t a1 /* Q2.30 in [-2,2] */ ,
                     int32_t b0 /* Q1.31 in [-1,1] */ ,
                     int32_t b1 /* Q2.30 in [-2,2] */ ,
                   int32_t d  /* Q1.31 in [0.88,0.99] */ ){
  int32_t r0  = mul(a0, b0);          // Q2.30 in [-1,1]
  int32_t r1  = r0 >> 2;              // Q4.28 in [-1,1]
  int32_t r2  = mul(a1, b1);          // Q4.28 in [-4,4]
  int32_t r3  = r1 + r2;              // Q4.28 in [-5,5]
  int32_t r4  = div32hs(r3, d, 31); // Q4.28 in [-8,8]
  return r4;
}
```

Listing 4. C code of the `compute_2_0` function.

C. How to use correctly fixed-point division?

Division in Section IV is the trickiest among the arithmetic operators treated. It requires one to provide a parameter η that impacts the format of the output. In practice, instead of choosing the parameter η, we compute it according to an expected acceptable output integer part using (7). This integer part can be set using multiple ways: 1) Set to a constant, 2) Using a function of the formats of the operands. For instance, if we want division results to have an integer part two bits larger than the integer part of its left operand denoted i_1, we will use the function $f(i_1) = i_1 + 2$ to compute i. In this example, using (7), we deduce that the parameter $\eta = f_2 - 2$.

At first sight, the first solution seems to be either too restrictive or too unsafe. Indeed with long chains of computations, the format of the intermediate operands tend to grow and choosing a small enough output integer part is a good idea to bring the result coefficient to a manageable range. Conversely choosing a small output integer part increases the chance that overflows occur at run-time. In Section VI, we illustrate the interest of using the first solution on some cases, and we show experimental evidence of the problems caused by the different methods of deciding η.

VI. Experimental results

In this section, we conduct some experiments to show the interest of using the operators specified in Section IV as well as the Cholesky decomposition and triangular inversion basic blocks of Section V. First we investigate the impact of the output format of division. Second we study the speed of the generation and the sharpness of the error bounds of the result. Finally we show the impact of the matrix condition number on the accuracy of the generated code.

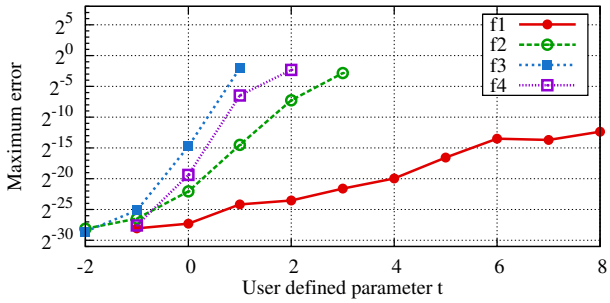
A. Impact of the output format of division on accuracy

As mentioned in Section V-C, we need to explicitly fix the output format of each division. More particularly, we define each output integer part using a function of the formats of the operands, the output fraction part being determined so as each result fits on 32 bits. This first experiment shows the impact of this function on the output accuracy. To do so, we defined four functions:

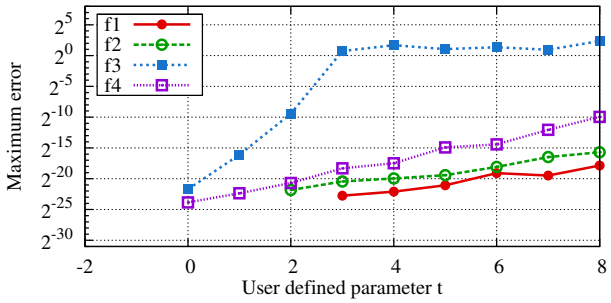
$$f_1(i_1, i_2) = t,\ f_2(i_1, i_2) = \min(i_1, i_2) + t,$$

$$f_3(i_1, i_2) = \max(i_1, i_2) + t, \text{ and } f_4(i_1, i_2) = \lfloor (i_1 + i_2)/2 \rfloor + t,$$

where $t \in \mathbb{Z}$ is a user defined parameter, and i_1 and i_2 are the integer parts of the numerator and denominator, respectively. The function f_1 consists in fixing all the division results to the same fixed-point format. The experiment consists in computing the Cholesky decomposition and the triangular inversion of matrices of size 5 and 10, respectively. Using FPLA, we synthesize codes for each problem and each function in $\{f_1, f_2, f_3, f_4\}$, for t ranging from -2 to 8. Then for each synthesized code, 10000 example instances are generated and solved both in fixed and floating-point arithmetics. Each example input is a matrix having 32-bits coefficients ranging from -1 to 1. Then the error considered is obtained by comparing the results to floating-point computations and by considering the maximum errors among the 10000 samples. The results are shown in Figure 1. No curve in the figures means that all of the examples overflew, and none succeeded.



(a) Cholesky 5×5.



(b) Triangular 10×10.

Figure 1. Maximum error of Cholesky decomposition and triangular inverse with various functions used to determine output formats of division.

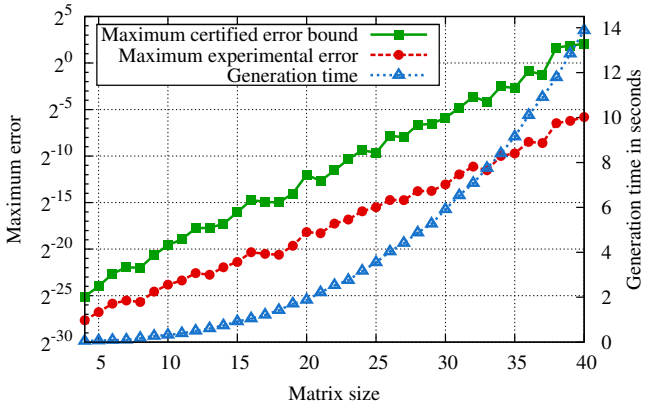


Figure 2. Comparison of the error bounds and experimental errors together with generation time, for the inversion of triangular matrices of size 4 to 40.

Obviously, we can observe that the function used to determine the output format of division has a great impact on the accuracy of the generated code. For example, if we consider the case $t = 0$ on 5×5 Cholesky decomposition on Figure1(a), using f_1 leads to an error of $\approx 2^{-28}$, while using f_3 gives an error $\approx 2^{-15}$, that is, twice larger than f_1. More particularly, we can observe that a good function choice is one that minimizes the output integer part. Indeed, as long as $t \geq -1$, using the function f_1 always leads to better maximum error than using function f_3. In addition surprisingly, as long as $t \geq -1$, the function that gives the best results is $f_1(i_1, i_2) = t$, namely the function that fixes explicitly all the division results of a resulting code to the same fixed-point format independently of the input formats. Indeed the problem of using a function that depends on the input formats comes from the fact that it quickly leads to a growth of the integer part of each computed coefficient, since it relies on the previously computed coefficient themselves. Hence the interest of f_1 is that it avoids this fast growth, and leads to result coefficients having a fixed and relatively small integer part, thus to tighter errors than the other functions. This remark is particularly true on 5×5 Cholesky decomposition on Figure1(a) when $t \geq 0$ where f_1 is the only function that leads to successful results. This phenomenon becomes obvious as the matrix size increases.

However we cannot restrain ourselves to function f_1 when implementing matrix inversion basic blocks. Indeed, cases occur where f_1 leads to unsuccessful results. This occurs when the result of some division has a fixed-point format with an integer part greater than t. This is the case on 10×10 triangular inversion on Figure1(b), where the diagonal coefficients of the inverse are $1/a_{i,j}$. Since $a_{i,j}$ may be arbitrarily small, then $1/a_{i,j}$ may be too large to fit into a format with an integer part of t bit. In these cases, other functions should be preferred.

B. *Sharpness of the error bounds and generation time*

The originality of our approach is the automatic generation of certified error bounds along with the synthesized code. This enables the generated code to be used in critical and precision sensitive applications. However, it is equally important that these bounds be sharp, at least for a large class of matrices. To investigate their sharpness, we compare in this second experiment the error bounds for the case of triangular matrix inversion with the experimental errors obtained from inverting 10000 sample matrices. This experiment is carried out using the f_4 function introduced in the previous experiment with $t = 1$. For each matrix size from 4 to 40, C code and error bounds are obtained by running FPLA. The generation time for each size is shown by the third curve with triangle-shaped dots and the right ordinate of Figure 2. It does not exceed 14 seconds on an Intel Core i7-870 2.93 GHz and for 40×40 matrices, an improvement of several orders of magnitude over a hand written fixed-point code.

The remaining two curves show the evolution of the error bounds and the experimental errors. The bounds vary from 2^{-26} to 2^{2} while the experimental errors vary from 2^{-28} to 2^{-6}. The difference between the error bounds and experimental errors is less than 2 bits for size-4 matrices and is inferior to 5 bits for size-15 matrices, and it grows as the size of the input matrices grows. Although the bounds obtained for matrices of size larger than 35 are too large to be useful in practice, the experimental errors are still tight enough and do not exceed 2^{-6}. These issues may be tackled by considering other means to handle division that are more suited to large matrices. Indeed, our experiments tend to show that the output format of division impacts heavily the accuracy of the result and that there is no way to determine this format that is adapted to all matrix sizes. We also argue that a bound of 2^{-12} on the inversion of size-20 matrices is satisfying for a broad range of applications, and this is a large improvement over hand-written fixed-point codes or codes whose numerical quality is asserted uniquely by simulations and a posteriori processes.

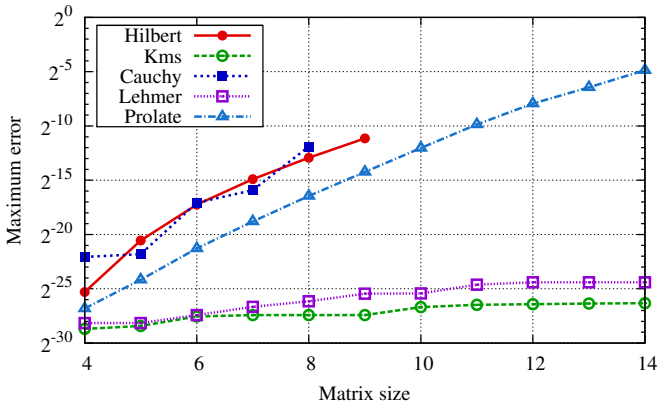


Figure 3. Maximum errors measured when computing the Cholesky decomposition of various kinds of matrices for sizes varying from 4 to 14.

C. Impact of the matrix condition number on accuracy

The sample matrices considered in the previous experiments were randomly drawn in the input intervals. In this third experiment, we consider some rather known matrices namely, Hilbert, Cauchy, Kms, Lehmer, and Prolate matrices. Among these, Hilbert and Cauchy matrices and to a lower extent Prolate are ill conditioned. Nonetheless, with a fixed-point code generated for matrices in the input format $\mathbf{Q}_{1.31}$, we were able to check that the fixed-point results, whenever computable, are quite accurate as shown in Figure 3. For sizes larger than 8 and 9, respectively, overflows occur when computing the decompositions of Cauchy and Hilbert matrices. But this fact does not invalidate our approach. Indeed, these matrices are very ill conditioned and are difficult to decompose accurately even in floating-point arithmetic.

On the other hand, Lehmer and Kms matrices have a linearly growing condition number and are therefore very well suited to our approach. Indeed as shown by the two bottom curves, the code generated by FPLA decomposes these matrices with a precision of up to 25 bits.

VII. Conclusion

In this article, we presented an automated approach to help in writing codes in fixed-point arithmetic for the particular case of matrix inversion based on Cholesky decomposition. First we formalized rigorously the fixed-point square root and division, especially in terms of error bounds, and implemented both of them in the CGPE tool. Second we introduced FPLA, dedicated to the synthesis of linear algebra subroutines. Our work focused particularly on the correct handling of fixed-point division, which is not as straightforward as the other operators and for which the output format must be carefully chosen to ensure a certain accuracy on the result. Enhancing our tool chain with these operators allowed us to tackle Cholesky decomposition and triangular matrix inversion. We finally showed that accurate fixed-point codes accompanied by bounds on the rounding errors can be automatically generated in a few seconds to invert and to decompose matrices of sizes up to 40.

In addition our further research direction is twofold: As a first direction, we intend to investigate the synthesis of other linear algebra basic blocks, like LU or QR decompositions, that may also be used to implement matrix inversion, and to implement them in FPLA. As a second direction, we aim at investigating, similarly to the work done in [12], the different trade-offs involved in the code synthesis process and especially the one between code size and accuracy.

References

[1] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Society for Industrial and Applied Mathematics, 2002.

[2] L. Zhou, L. Qiu, and J. Zhu, "A novel adaptive equalization algorithm for MIMO communication system," in *Proc. of the IEEE 62nd Vehicular Technology Conference (VTC-2005-Fall)*, vol. 4, 2005, pp. 2408–2412.

[3] H. Chen, X. Deng, and A. Haimovich, "Layered turbo space-time coded mimo-ofdm systems for time varying channels," in *Proc. of the 2003 IEEE Global Telecommunications Conference (GLOBECOM'03)*, vol. 4, 2003, pp. 1831–1836 vol.4.

[4] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, 2008.

[5] R. Yates, *Fixed-Point Arithmetic: An Introduction*, Digital Signal Labs, 2013.

[6] D. Menard, D. Chillet, and O. Sentieys, "Floating-to-fixed-point conversion for digital signal processors," in *EURASIP Journal on Applied Signal Processing*, 2006, pp. 1–15.

[7] Z. Nikolic, H. T. Nguyen, and G. Frantz, "Design and Implementation of Numerical Linear Algebra Algorithms on Fixed-Point DSPs," *EURASIP J. Adv. Sig. Proceedings*, vol. 2007, 2007.

[8] W. Sung and K.-I. Kum, "Simulation-based word-length optimization method for fixed-point digital signal processing systems," *IEEE Trans. Signal Processing*, vol. 43, no. 12, pp. 3087–3090, 1995.

[9] D.-U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, "Accuracy-guaranteed bit-width optimization," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1990–2000, 2006.

[10] C. F. Fang, R. A. Rutenbar, and T. Chen, "Fast, Accurate Static Analysis for Fixed-Point Finite-Precision Effects in DSP Designs," in *Proc. of the 2003 IEEE/ACM International Conf. on Computer-aided Design (ICCAD'03)*. IEEE Computer Society, 2003, pp. 275–282.

[11] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[12] M. Martel, A. Najahi, and G. Revy, "Code Size and Accuracy-Aware Synthesis of Fixed-Point Programs for Matrix Multiplication," in *Proc. of the 4th International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS'14)*, 2014, pp. 204–214.

[13] A. Irturk, B. Benson, S. Mirzaei, and R. Kastner, "GUSTO: An Automatic Generation and Optimization Tool for Matrix Inversion Architectures," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 4, pp. 32:1–32:21, 2010.

[14] C. Mouilleron, A. Najahi, and G. Revy, "Automated Synthesis of Target-Dependent Programs for Polynomial Evaluation in Fixed-Point Arithmetic," UPVD/LIRMM, Tech. Rep., 2013.

[15] R. E. Moore, *Interval Analysis*. Prentice-Hall, 1966.

[16] C. Mouilleron and G. Revy, "Automatic Generation of Fast and Certified Code for Polynomial Evaluation," in *Proc. of the 20th IEEE Symposium on Computer Arithmetic (ARITH'20)*, 2011, pp. 95–103.

[17] B. Lopez, T. Hilaire, and L.-S. Didier, "Sum-of-products evaluation schemes with fixed-point arithmetic, and their application to IIR filter implementation," in *Proc. of the Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2012, pp. 160–167.

[18] D.-U. Lee and J. D. Villasenor, "Optimized Custom Precision Function Evaluation for Embedded Processors," *IEEE Trans. Computers*, vol. 58, no. 1, pp. 46–59, 2009.

[19] R. C. C. Cheung, D.-U. Lee, O. Mencer, W. Luk, and P. Y. K. Cheung, "Automating custom-precision function evaluation for embedded processors," in *Proc. of the Inter. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05)*. ACM, 2005, pp. 22–31.

[20] M. D. Ercegovac, L. Imbert, D. W. Matula, J.-M. Muller, and G. Wei, "Improving Goldschmidt division, square root, and square root reciprocal," *IEEE Trans. Computers*, vol. 49, no. 7, pp. 759–763, 2000.

[21] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.

[22] International Organization for Standardization, *Programming Languages – C*. Geneva, Switzerland: ISO/IEC Standard 9899:2010, 2010.