Numerical Program Optimization by Automatic Improvement of the Accuracy of Computations

Nasrine Damouche* and Matthieu Martel

Université de Perpignan,
LAboratory of Mathematics and PhysicS, LAMPS
52 Avenue Paul Alduy,
66860 Perpignan, France
Fax: +33 04 68 66 22 34
E-mail: nasrine.damouche@univ-perp.fr
E-mail: matthieu.martel@univ-perp.fr
*Corresponding author

Alexandre Chapoutot

U2IS, ENSTA ParisTech,
Université Paris-Saclay,
828 bd des Maréchaux, Palaiseau, France, 91762.
Fax: (+33) 1 81 87 20 71
E-mail: chapoutot@ensta.fr

Abstract: Over the last decade, guaranteeing the accuracy of computations relying on the IEEE754 floating-point arithmetic has become increasingly complex. Failures, caused by small or large perturbations due to round-off errors, have been registered. To cope with this issue, we have developed a tool which corrects these errors by automatically transforming programs in a source to source manner. Our transformation, relying on static analysis by abstract abstraction, operates on pieces of code with assignments, conditionals and loops. By transforming programs, we can significantly optimize the numerical accuracy of computations by minimizing the error relatively to the exact result. In this article, we present two important desirable side-effects of our transformation. Firstly, we show that our transformed programs, executed in single precision, may compete with not transformed codes executed in double precision. Secondly, we show that optimizing the numerical accuracy of programs accelerates the convergence of numerical iterative methods. Both of these properties of our transformation are of great interest for numerical software.

Keywords: Program Transformation; Floating-Point Numbers; IEEE754 Standard; Data-Types Format Optimization; Convergence Acceleration.

Biographical notes:

Nasrine DAMOUCHE is Postdoc at the Laboratory of Mathematics and Physics (LAMPS) at the University of Perpignan Via Domitia (UPVD), France, since September 2016. She received her Ph.D. in Computer Science from the University of Perpignan in 2016 for her work on improving the numerical accuracy of floating-point programs with automatic code transformation methods. She received her master degree in Computer Science from University Paul Sabatier, Toulouse III in 2013. Formerly, she received a master degree in Computer Science from University Mouloud Mammeri, Tizi-Ouzou in 2011.

Her research mainly focuses on numerical accuracy, floating-point arithmetic, automatic transformation and static analysis by abstract interpretation. She is also interested by methods of verification, validation and compilation. Nasrine DAMOUCHE is the author of more than ten scientific articles. She is also the main developer of the Salsa tool to optimize numerical codes.

Matthieu MARTEL is Professor in computer science at the University of Perpignan Via Domitia (UPVD). He is also the head of the VENUS research group of the Laboratory of Mathematics and Physics (LAMPS), Vice-President in charge of technology transfers at UPVD and co-founder of the Numalis company. Matthieu MARTEL has been working for 15 years on static analysis and program transformation for numerical accuracy. Matthieu MARTEL is the author of more than 40 scientific articles in this domain, he is also the advisor of more than ten Ph.D. dissertations in this area. From 2000 to 2008, Matthieu MARTEL has been working as a researcher at the CEA, the French atomic energy agency and as part-time associate professor at Ecole Polytechnique.

Alexandre CHAPOUTOT is an Associated professor at ENSTA ParisTech since 2010. He received his Ph.D. in Computer Science from Ecole polytechnique in 2008 for his work on static analysis by abstract interpretation of hybrid systems. He received his master degree in Computer Science from University Pierre et Marie Curie (UPMC) in 2005. He worked as a research and teaching assistant at University Pierre et Marie Curie, at LIP6, where he studied the accuracy of floating-point computations in programs. His research activities are mainly focused on the definition and the application of set-based methods, for the analysis and the verification of properties of cyber-physical systems. He is also interested in the analysis and the improvement of floating-point accuracy in programs.

1 Introduction

Floating-point numbers, whose specification is given by the IEEE754 Standard (ieee, 2008; Muller et al., 2010), are more and more used in many industrial applications, including critical embedded software. In floating-point computations, an ubiquitous problem is that round-off errors limit the accuracy of the results. The approximation becomes problematic when accumulated errors cause damages whose gravity varies depending on the context of the application. We correct partly these errors by automatically transforming programs in a source to source manner. Our method not only transforms arithmetic expressions (Ioualalen and Martel, 2012; Panchekha et al., 2015) but also pieces of code containing assignments, conditionals, loops and sequences of commands. Basically, we generate large arithmetic expressions corresponding to the computations of the original program and further, we consider many expressions mathematically equivalent to the original ones in order to, finally, choose a more accurate one in polynomial time (Damouche et al., 2015b).

There exist several methods for validating (Bertrane et al., 2011; Darulova and Kuncak, 2014; Delmas et al., 2009; Goubault, 2013; Solovyev et al., 2015) and improving (Ioualalen and Martel, 2012; Panchekha et al., 2015) the accuracy of arithmetic expressions in order to avoid numerical failures. In this article, as in our previous work, we rely on static analysis by abstract interpretation (Cousot and Cousot, 1977) to compute variable ranges and round-off error bounds. We use a set of transformation rules for arithmetic expressions and commands (Damouche et al., 2015b). These rules, which are applied in a deterministic order, allow one to obtain a more accurate code among all the codes which are considered. We have shown in previous work (Damouche et al., 2015b) that the numerical accuracy of

programs is significantly improved (in most cases, the worst error on the result, for all the considered inputs, is decreased of about 20%.)

In this article, we present two important desirable side-effects of our transformation. Firstly, we compare the transformed programs obtained by our tool and executed in single precision to the initial programs executed in double precision. The basic idea is to show that the transformed programs, executed in single precision, compete in terms of accuracy with the original programs in double precision. This allows the programmer to use smaller data-type formats without loosing much information. It is then possible to save memory, reduce CPU usage and use less bandwidth for communications whenever distributed applications are concerned. In order to compare the effect of accuracy on the choice of formats, we take as an illustrative example Simpson's Rule (Atkinson, 1988) and we use it to compute the integral of a non-linear function.

Secondly, we show that by optimizing iterative numerical methods to be more accurate, we accelerate their convergence speed. In order to demonstrate the impact of the accuracy on the convergence time, we have chosen a set of four representative iterative methods which are Jacobi's and Newton-Raphson's method, a method to compute the largest Eigenvalue and Gram-Schmidt's method. Significant speedups are obtained in terms of number of iterations.

Some of the results presented here have been partly introduced in (Damouche et al., 2016) (concerning the reduction of the precision) and some others have been partly introduced in (Damouche et al., 2015a) (concerning the convergence acceleration). These experimentations are gathered in this article in order to demonstrate the impact of our transformation. In addition, we provide complementary benchmarks concerning speedups and the total number of floating-point operations (flops) performed by the original and transformed programs. Our objective is to check that the advantages of our techniques are not annealed by overheads in the execution time due to the transformation itself or by other side effects introduced for example by the compiler.

To ensure that our tool is useful in practice, all the source and transformed program have been implemented in the `C` programming language, compiled with `GCC 4.9.2`, and executed on an `Intel Core i7` in IEEE754 single precision in order to emphasize the effect of the finite precision. Programs are compiled with the default optimization level of the compiler `-o0`. We have tried other levels of optimization without observing significant changes in our results.

This article is organized as follows. We first introduce in Section 2 the floating-point arithmetic, how to compute the error bounds and we explain how to transform arithmetic expressions (Ioualalen and Martel, 2012). We introduce in Section 2.4 the basics of our transformation method and the different transformation rules that allow us to automatically transform programs. We demonstrate in Section 3 the interest of the comparison between original codes executed in double precision and transformed codes run in single precision. Section 4, we show that by improving the numerical accuracy of computations, we accelerate the convergence speed of numerical iterative methods. We discuss in Section 5 complementary experiments by studying the speedups of the numerical method in terms of number of iterations, time and total number of floating-point operations (flops). Finally, we give some concluding remarks and perspectives in Section 6.

2 Preliminaries

In this section, we give some background concerning the techniques used to achieve our program transformation. Section 2.1 briefly describes the floating-point arithmetic. Section 2.2 introduces the method that we use to compute the errors introduced by floating-point computations with respect to exact computations. Sections 2.3 and 2.4 respectively present the transformation techniques used to optimize arithmetic expressions (Ioualalen and Martel, 2012) and programs (Damouche et al., 2015b). these transformations are implemented in the tool that we use to optimize the programs of sections 3 and 4.

2.1 Floating-Point Arithmetic

Floating-point numbers are used to represent real numbers. Because of their finite representation, round-off errors arise during the computations which may cause damages in critical contexts. The IEEE754 Standard formalizes a binary floating-point number as a triplet of sign (s $\in \{-1, 1\}$), mantissa and exponent. We consider that a number x is written:

$$x = s \cdot (d_0.d_1 \dots d_{p-1}) \cdot b^e = s \cdot m \cdot b^{e-p+1} \ , \tag{1}$$

where,

- s is the sign $\in \{-1, 1\}$,
- b is the basis, $b = 2$,
- m is the mantissa, $m = d_0.d_1 \dots d_{p-1}$ with digits $0 \leq d_i < b$, $0 \leq i \leq p-1$,
- p is the precision,
- e is the exponent $\in [e_{min}, e_{max}]$.

A floating-point number x is *normalized* whenever $d_0 \neq 0$. Normalization avoids multiple representations of the same number. IEEE754 Standard specifies some particular values for p, e_{min} and e_{max}, which are summarized in Figure 1, as well as *denormalized numbers* which are floating-point numbers with $d_0 = d_1 = \ldots = d_k = 0$, $k < p-1$ and $e = e_{min}$. Denormalized numbers make underflow gradual (Goldberg, 1991). Finally, the following special values also are defined:

- `NaN` (Not a Number) result of an invalid operation,
- The values $\pm\infty$ corresponding to overflows,
- The values $+0$ and -0 (signed zeros).

The IEEE754 Standard defines five rounding modes for elementary operations over floating-point numbers. These modes are towards $-\infty$, towards $+\infty$, towards zero, to the nearest ties to even and to the nearest ties to zero respectively denoted by $\uparrow_{+\infty}$, $\uparrow_{-\infty}$, $\uparrow_0$, $\uparrow_{\sim_e}$ and $\uparrow_{\sim_0}$. The semantics of the elementary operations specified by the IEEE754 Standard is given by Equation (2).

$$x \circledast_r y = \uparrow_r (x * y) \ , \text{ with } \uparrow_r : \mathbb{R} \to \mathbb{F} \ , \tag{2}$$

Format	Name	p	e bits	e_{min}	e_{max}
Binary16	Half precision	11	5	-14	$+15$
Binary32	Single precision	24	8	-126	$+127$
Binary64	Double precision	53	11	-1122	$+1223$
Binary128	Quadratic precision	113	15	-16382	$+16383$

Figure 1 Basic IEEE754 formats.

where a floating-point operation, denoted by $\circledast_r$, is computed using the rounding mode $r \in \{\uparrow_{+\infty}, \uparrow_{-\infty}, \uparrow_0, \uparrow_{\sim_e}$ and $\uparrow_{\sim_0}\}$ and $* \in \{+, -, \times, /\}$ an exact operation. Obviously, the results of the computations are not exact because of the round-off errors. This is why, we use also the function $\downarrow_r$: $\mathbb{R} \to \mathbb{R}$ that returns the round-off error. We have

$$\downarrow_r (x) = x - \uparrow_r (x) \ . \tag{3}$$

2.2 Error Bound Computation

In order to compute the errors during the evaluation of arithmetic expressions (Martel, 2006) , we use values which are pairs $(x, \mu) \in \mathbb{F} \times \mathbb{R} \equiv \mathbb{E}$ where x denotes the floating-point number used by the machine and μ denotes the exact error attached to $\mathbb{F}$, i.e., the exact difference between the real and floating-point numbers as defined in Equation (3).

Example 2.1 *For example, let us consider the real number π which cannot be represented exactly on the machine. In single precision, the binary and decimal representations of π respectively are* $1.10010010000111111011011 \times 2^1$ *and* 3.1415928. *In our semantics, π is represented by the value* $v = (\uparrow_\sim (\pi), \downarrow_\sim (\pi)) = (3.1415928, (\pi - 3.1415928))$. ■

Our tool uses an abstract semantics (Cousot and Cousot, 1977) based on $\mathbb{E}$. The abstract values are represented by a pair of intervals. The first interval contains the range of the floating-point values of the program and the second one contains the range of the errors obtained by subtracting the floating-point values from the exact ones. In the abstract value denoted by $(x^\sharp, \mu^\sharp) \in \mathbb{E}^\sharp$, we have $x^\sharp$ the interval corresponding to the range of the values and $\mu^\sharp$ the interval of errors on $x^\sharp$. This value abstracts a set of concrete values $\{(x, \mu) : x \in x^\sharp$ and $\mu \in \mu^\sharp\}$ by intervals in a component-wise way. We now introduce the semantics of arithmetic expressions on $\mathbb{E}^\sharp$. We approximate an interval $x^\sharp$ with real bounds by an interval based on floating-point bounds, denoted by $\uparrow^\sharp (x^\sharp)$. Here bounds are rounded to the nearest, see Equation (4).

$$\uparrow^\sharp_\sim ([\underline{x}, \overline{x}]) = [\uparrow_\sim (\underline{x}), \uparrow_\sim (\overline{x})] \ . \tag{4}$$

We denote by $\downarrow^\sharp$ the function that abstracts the concrete function $\downarrow_\sim$. It over-approximates the set of exact values of the error $\downarrow_\sim (x) = x - \uparrow_\sim (x)$. Every error associated to $x \in [\underline{x}, \overline{x}]$ is included in $\downarrow^\sharp ([\underline{x}, \overline{x}])$. We also have for a rounding mode to the nearest

$$\downarrow^\sharp ([\underline{x}, \overline{x}]) = [-y, y] \quad \text{with} \quad y = \frac{1}{2}\text{ulp}\big(\max(|\underline{x}|, |\overline{x}|)\big) \ . \tag{5}$$

Formally, the *unit in the last place*, denoted by ulp(x), consists of the weight of the least significant digit of the floating-point number x. Equations (6) to (8) give the semantics of the addition, the substraction and multiplication over $\mathbb{E}^\sharp$, for other operations see (Martel,

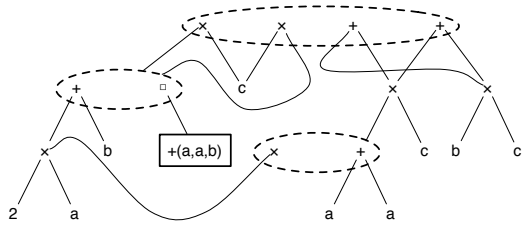

Figure 2 APEG for the expression $e = \big((a + a) + c\big) \times c$.

2006). If we sum two numbers, we must add errors on the operands to the error produced by the round-off of the result. When multiplying two numbers, the semantics is given by the development of $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$. Note that, the semantics of the elementary operations on $\mathbb{E}$ is defined in a former work in (Martel, 2006).

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = \big(\uparrow^\sharp (x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp + x_2^\sharp) \big) \ , \tag{6}$$

$$(x_1^\sharp, \mu_1^\sharp) - (x_2^\sharp, \mu_2^\sharp) = \big(\uparrow^\sharp (x_1^\sharp - x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp - x_2^\sharp) \big) \ , \tag{7}$$

$$(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) = \big(\uparrow^\sharp (x_1^\sharp \times x_2^\sharp), x_2^\sharp \times \mu_1^\sharp + x_1^\sharp \times \mu_2^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp \times x_2^\sharp) \big) \ . \tag{8}$$

2.3 Accuracy Improvement

To introduce the transformation of arithmetic expressions, we consider variables $id \in \mathcal{V}$ with $\mathcal{V}$ a finite set, constants $cst \in \mathbb{F}$ with $\mathbb{F}$ the set of floating-point numbers and the operators $+$, $-$, $\times$ and $/$. The syntax is

$$\text{Expr} \ni e ::= id \mid cst \mid e + e \mid e - e \mid e \times e \mid e/e \ . \tag{9}$$

Here, we briefly present former work (Ioualalen and Martel, 2012; Tate et al., 2011) to semantically transform (Cousot and Cousot, 2002) arithmetic expressions using Abstract Program Expression Graph (APEG). This data structure remains in polynomial size while dealing with an exponential number of equivalent expressions.

An APEG is defined inductively as follows: (1) A value v or a variable x is an APEG, (2) An expression $p_1 * p_2$ is an APEG, where p_1 and p_2 are APEGs and $*$ is a binary operator, (3) A box $\boxed{*(p_1, \ldots, p_n)}$ is an APEG, where $*$ is a commutative and associative operator and the p_i, $1 \leq i \leq n$, are APEGs and (4) A non-empty set $\{p_1, \ldots, p_n\}$, called equivalence class, of APEGs is an APEG where p_i, $1 \leq i \leq n$, is not a set of APEGs itself. An example of APEG is given in Figure 2. When an equivalence class (denoted by a dotted ellipse in Figure 2) contains many APEGs $p_1, \ldots, p_n$ then one of the p_i $1 \leq i \leq n$ may be selected in order to build an expression. A box $\boxed{*(p_1, \ldots, p_n)}$ represents any parsing of the expression $p_1 * \ldots * p_n$. From an implementation point of view, when several equivalent expressions share a common sub-expression, the latter is represented only once in the APEG. Then APEGs provide a compact representation of a set of equivalent expressions and make it possible to represent in an unique structure many equivalent expressions of very different shapes. For readability reasons, in Figure 2, the leafs corresponding to the variables a, b and c are duplicated while, it practice, they are defined only once in the structure. The set $\mathcal{A}(p)$ of expressions contained inside an APEG p is defined inductively as follows:

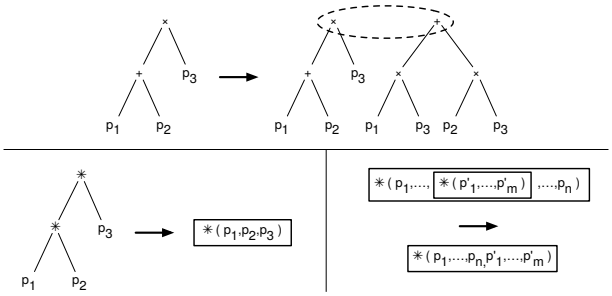

Figure 3 Some rules for APEG construction by pattern matching.

1. If p is a value v or a variable x then $\mathcal{A}(p) = \{v\}$ or $\mathcal{A}(p) = \{x\}$.
2. If p is an expression $p_1 * p_2$ then $\mathcal{A}(p) = \bigcup_{e_1} \in \mathcal{A}(p_1),\ e_2 \in \mathcal{A}(p_2) e_1 * e_2$
3. If p is a box $\boxed{*(p_1, \ldots, p_n)}$ then $\mathcal{A}(p)$ contains all the parsings of $e_1 * \ldots * e_n$, for all $e_1 \in \mathcal{A}(p_1), \ldots, e_n \in \mathcal{A}(p_n)$.
4. If p is an equivalence class $\{p_1, \ldots, p_n\}$ then $\mathcal{A}(p) = \bigcup_{1 \leq i \leq n} \mathcal{A}(p_i)$.

For instance, the APEG p of Figure 2 represents all the following expressions:

$$\mathcal{A}(p) = \left\{ \begin{array}{c} \big((a+a)+b\big) \times c,\ \big((a+b)+a\big) \times c,\ \big((b+a)+a\big) \times c, \\ \big((2 \times a)+b\big) \times c,\ c \times \big((a+a)+b\big),\ c \times \big((a+b)+a\big), \\ c \times \big((b+a)+a\big),\ c \times \big((2 \times a)+b\big),\ (a+a) \times c + b \times c, \\ (2 \times a) \times c + b \times c,\ b \times c + (a+a) \times c,\ b \times c + (2 \times a) \times c \end{array} \right\} \quad (10)$$

In their article on EPEGs, R. Tate *et al.* use rewriting rules to extend the structure up to saturation (Tate et al., 2011). In our context, such rules would consist of performing some pattern matching in an existing APEG p and then adding new nodes in p, once a pattern has been recognized. For example, the rules corresponding to distributivity and box construction are given in Figure 3. An alternative technique for APEG construction is to use dedicated algorithms. Such algorithms, working in polynomial time, have been proposed in (Ioualalen and Martel, 2012).

2.4 Transformation of Commands

In this section, we focus on the transformation of commands which is done using a set of rewriting rules. Our language is made of assignments, conditionals, loops and sequences of commands. The syntax is

$$\text{Com} \quad \ni \quad c ::= id = e \ \mid\ c_1 \ ;\ c_2 \ \mid \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \ \mid\ \mathsf{while}_\Phi\ e\ \mathsf{do}\ c \ \mid\ \mathsf{nop}. \quad (11)$$

The transformation relies on several hypotheses. First of all, programs are assumed to be in static single assignment form (SSA form) (Cytron and Gershbein, 1993). The principle of this intermediary representation is that every variable may be assigned only once in the source code and must be used before its use. To understand this intermediary representation, let us consider the example 2.1. In the original program, x is assigned several times. In the program in SSA form, a new variable x_1, x_2, etc. is used for each

assignment and at the junction of control paths (in conditionals or loops), a Φ-node $\Phi(x_1, x_2, x_3)$ indicates that we assign to x_1 the value of x_2 or x_3 depending on where we are coming from.

Example 2.1 *Initially, we give the original program in Equation* (12)*.*

$$\begin{array}{l} x = a\ ; \\ \textit{if}\ (x > 1.0)\ \textit{then} \\ \quad x = x \times 2.0\ ; \\ \textit{else} \\ \quad x = x\ /\ 2.0\ ; \\ \quad z = x \end{array} \tag{12}$$

The SSA form of the program given in Equation (12) *is*

$$\begin{array}{l} x_1 = a\ ; \\ \textit{if}\ (x_1 > 1.0)\ \textit{then} \\ \quad x_2 = x_1 \times 2.0\ ; \\ \textit{else} \\ \quad x_3 = x_1\ /\ 2.0\ ; \\ \quad \Phi(x_4, x_2, x_3)\ ; \\ \quad z = x_4 \end{array} \tag{13}$$

In the new program, given in Equation (13)*, the variables are assigned only once. Then x has been split into x_1, x_2, x_3 and x_4. The Φ-node $\Phi(x_4, x_2, x_3)$ states that x_4 is assigned to x_2 or x_3 depending on the branch taken by the control flow.* ■

The second hypothesis is that we optimize a reference variable defined by the user. Our transformation is defined by rules using states $\langle c, \delta, C, \nu, \beta \rangle$ where

- c is a command, as defined in Equation (11),
- δ is an environment $\delta : \mathcal{V} \to$ Expr which maps variables to expressions. Intuitively, this environment records the expressions assigned to variables in order to inline them later on in larger expressions,
- $C \in$ Ctx is a single hole context (Hankin, 1994). It records the program enclosing the current expression to be transformed,
- $\nu \in \mathcal{V}$ denotes the reference variable that we aim at optimizing,
- $\beta \subseteq \mathcal{V}$ is a list of assigned variables that should not be removed from the code. Initially, $\beta = \{\nu\}$, i.e., the target variable ν must not be removed.

The environment δ is used to discard assignments from programs and to re-insert the expressions when the variables are read, in order to build larger expressions.

Let us consider first assignments. If (i) the variable v of some assignment $v = e$ does not exist in the domain of δ, if (ii) $v \notin \beta$ and if (iii) $v \neq \nu$ then we memorize e in δ and we remove the assignment from the program. Otherwise, if one of the conditions (i), (ii) or (iii) is not satisfied then we rewrite this assignment by inlining the variables saved in δ in the concerned expression. Note that, when transforming programs by inlining expressions in variables, we get larger formulas. The basic idea, in our implementation, when dealing with too large expressions, is to create intermediary variables and to assign to them the sub-expressions obtained by slicing the global expression at a given level of the syntactic tree. The last step consists of re-inserting these intermediary variables into the main program.

Example 2.2 *For example, let us consider the program below in which three variables x, y and z are assigned. We assume that z is the variable that we aim at optimizing and $a = 0.1$, $b = 0.01$, $c = 0.001$ and $d = 0.0001$ are constants.*

$$\begin{array}{l} \langle \mathtt{x = a + b\ ;\ y = c + d\ ;\ z = x + y}\ ,\ \delta,\ [],\ [z]\rangle \\ \rightarrow_\nu \langle \mathit{nop}\ ;\ \mathtt{y = c + d\ ;\ z = x + y}, \delta' = \delta[\mathtt{x} \mapsto \mathtt{a + b}],\ x = a + b; [],\ [z]\rangle \\ \rightarrow_\nu \langle \mathit{nop}\ ;\ \mathit{nop}\ ;\ \mathtt{z = x + y}, \delta'' = \delta'[\mathtt{y} \mapsto \mathtt{c + d}],\ x = a + b; y = c + d; [],\ [z]\rangle \\ \rightarrow_\nu \langle \mathit{nop}\ ;\ \mathit{nop}\ ;\ \mathtt{z} = ((\mathtt{d + c}) + \mathtt{b}) + \mathtt{a},\ \delta'',\ x = a + b; y = c + d; [],\ [z]\rangle \end{array} \tag{14}$$

In Equation (14), the environment δ and the context C are initialy empty and the list β contains the reference variable z. We remove the variable x and memorize it in δ. So, the line corresponding to the variable discarded is replaced by nop *and the new environment is $\delta = [x \mapsto a + b]$. We then repeat the same process on the variable y. For the last step, we may not remove z because it is the reference variable. Instead, we substitute, in z, x and y by their values in δ and we transform the expression using the technique described in Section 2.2.* ■

Our tool also transforms conditionals. If a certain condition is always true or false, then we keep only the right branch, otherwise, we transform both branches of the conditional. When it is necessary, we re-inject variables that have been discarded from the main program.

Example 2.3 *Let us take another example to explain how we transform conditionals.*

$$\begin{array}{l} x_1 = a; \\ \mathit{if}_{\Phi(y_3,y_1,y_2)}\ x_1 > 1.0\ \mathit{then} \\ \quad y_1 = x_1 + 2.0; \\ \mathit{else} \\ \quad y_2 = x_1 - 1.0; \\ \quad \nu = y_3. \end{array} \tag{15}$$

First of all, x_1 is stored in δ. Then, we transform recursively the new program

$$\begin{array}{l} \mathit{if}_{\Phi(y_3,y_1,y_2)}\ x_1 > 1.0\ \mathit{then} \\ \quad y_1 = x_1 + 2.0; \\ \mathit{else} \\ \quad y_2 = x_1 - 1.0; \\ \quad \nu = y_3. \end{array} \tag{16}$$

■

This program is semantically incorrect since the test is undefined. So we re-inject the statement $x_1 = a$ in the program and add x_1 to the list β in order to avoid an infinite loop in the transformation.

For a sequence $c1;\ c2$, the first command c_1 is transformed into c_1' in the current environment δ, C, ν and β and a new context C' is built which inserts c_1' inside C. Then c_2 is transformed into c_2' using the context $C[c_1'; []]$, the formal environments δ' and the list β' resulting from the transformation of c_1. Finally, the state $\langle c_1'\ ;\ c_2', \delta'', \beta''\rangle$ is returned.

Other transformations have been defined for while loops. A first rule makes it possible to transform the body of the loop assuming that the variables of the condition have not been stored in δ. In this case, the body is transformed in the context $C[\mathsf{while}_\Phi\ e\ \mathsf{do}\ []]$ where C is the context of the loop. A second rule builds the list $V = Var(e) \cup Var(\Phi)$ where $Var(\Phi)$ is the list of variables read and written in the Φ nodes of the loop. The set V is used to achieve two tasks: firstly, it is used to build a new command c' corresponding to the sequence of assignments that must be re-inserted. Secondly, the variables of V are removed from the domain of δ and added to β. The resulting command is obtained by transforming $c'; \mathsf{while}_\Phi\ e\ \mathsf{do}\ c$ with δ' and $\beta \cup V$.

Listing 1 Listing of the initial Simpson's rule.

```
int main() {
  a = 1.9; b = 2.1; n = 100.0; i = 1.0; x = a; h = (b - a)/n;
  f = ((x * x * x * x * x * x * x * x * x) - 14.0 * (x * x * x * x * x * x * x)
    + 84.0 * (x * x * x * x * x * x) - 280.0 * (x * x * x * x * x)
    + 560.0 * (x * x * x) - 672.0 * (x * x) + 448.0 * x - 128.0);
  x = b ;
  g = ((x * x * x * x * x * x * x * x * x) - 14.0 * (x * x * x * x * x * x * x)
    + 84.0 * (x * x * x * x * x * x) - 280.0 * (x * x * x * x * x)
    + 560.0 * (x * x * x) - 672.0 * (x * x) + 448.0 * x - 128.0);
  s = f + g;
  while (i < n) {
    x = a + (i * h);
    f = ((x * x * x * x * x * x * x * x * x) - 14.0 * (x * x * x * x * x * x * x)
      + 84.0 * (x * x * x * x * x * x) - 280.0 * (x * x * x * x * x)
      + 560.0 * (x * x * x) - 672.0 * (x * x) + 448.0 * x - 128.0);
    s = s + 4.0 * f;
    i = i + 1.0;
  };
  i = 2.0 ;
  while (i < n-1) {
    x = a + (i * h) ;
    f = ((x * x * x * x * x * x * x * x * x) - 14.0 * (x * x * x * x * x * x * x)
      + 84.0 * (x * x * x * x * x * x) - 280.0 * (x * x * x * x * x)
      + 560.0 * (x * x * x) - 672.0 * (x * x) + 448.0 * x - 128.0);
    s = s + 2.0 * f;
    i = i + 1.0;
  };
  s = s * (h / 3.0);
}
```

3 Data-Types Optimization

We have shown in Section 2.4 how to optimize automatically intraprocedural programs (Damouche et al., 2015b) based on floating-point arithmetic. A tool named `Salsa` has been developed which implements our intraprocedural transformation rules. `Salsa` calls another tool, `Sardana`, to improve the numerical accuracy of arithmetic expression. `Sardana` uses the APEG introduced in Section 2.3. We have experimented `Salsa` to improve the numerical accuracy of controller algorithms, numerical methods, etc. This tool finds a more accurate program among all those equivalent.

This section gives the first application of our program transformation, i.e. the possibility to reduce the formats of the floating-point variables thanks to more accurate computations. We emphasize the efficiency of our implementation in terms of improving the data-types

Listing 2 Listing of the transformed Simpson Rule.

```
int main() {
  TMP_3 = 6.859 ;
  TMP_1 = ((((((TMP_3 × 1.9) × 1.9) × 1.9) × 1.9) − (14.0 × (((TMP_3 × 1.9) × 1.9)
        × 1.9))) + (84.0 × ((6.859 × 1.9) × 1.9)));
  TMP_2 = (1.9 × (280.0 × TMP_3));
  TMP_15 = 9.261000000000001;
  TMP_13 = ((((((TMP_15 × 2.1) × 2.1) × 2.1) × 2.1) − (14.0 × (((TMP_15 × 2.1)
        × 2.1) × 2.1))) + (84.0 × ((9.261000000000001 × 2.1) × 2.1)));
  TMP_14 = (2.1 × (280.0 × TMP_15));
  TMP_27 = 3.61;
  TMP_25 = ((((((TMP_27 × 1.9) × 1.9) × 1.9) × 1.9) × 1.9) − (14.0 × ((((TMP_27
        × 1.9) × 1.9) × 1.9) × 1.9)));
  TMP_26 = (1.9 × (159.599999999999994 × TMP_3));
  TMP_32 = (TMP_3 × 1.9);
  i = 1.0;
  s = ((((((TMP_1 − TMP_2) + (560.0 × TMP_3)) − 2425.920000000000073)
    + 851.1999999999999932) − 128.0) + (((((TMP_13 − TMP_14) + (560.0 × TMP_15))
    − 2963.5200000000000437) + 940.8000000000000068) − 128.0));
  f = ((((((TMP_25 + TMP_26) − (280.0 × TMP_32)) + (560.0 × (TMP_27 × 1.9)))
    − (672.0 × TMP_27)) + 851.1999999999999932) − 128.0);
  x = 2.1;
  while (i < 100.0) {
    x = (1.9 + (i × 0.002));
    TMP_37 = (x × x);
    TMP_35 = ((((((TMP_37 × x) × x) × x) × x) × x) − (14.0 × ((((TMP_37 × x) × x)
          × x) × x)));
    TMP_36 = (84.0 × (((TMP_37 × x) × x) × x));
    TMP_42 = (x × (TMP_37 × x));
    f = ((((((TMP_35 + TMP_36) − (280.0 × TMP_42)) + (560.0 × (TMP_37 × x)))
      − (672.0 × TMP_37)) + (448.0 × x)) − 128.0);
    s = (s + (4.0 × f));
    i = (i + 1.0);
  } ;
  i = 2.0 ;
  while (i < 99.) {
    x = (1.9 + (i × 0.002)) ;
    TMP_47 = (x × x) ;
    TMP_45 = ((((((TMP_47 × x) × x) × x) × x) × x)
          − (14.0 × ((((TMP_47 × x) × x) × x) × x))) ;
    TMP_46 = (84.0 × (((TMP_47 × x) × x) × x)) ;
    TMP_52 = (x × (TMP_47 × x)) ;
    f = ((((((TMP_45 + TMP_46) − (280.0 × TMP_52)) + (560.0 × (TMP_47 × x)))
      − (672.0 × TMP_47)) + (448.0 × x)) − 128.0) ;
    s = (s + (2.0 × f)) ;
    i = (i + 1.0) ;
  } ;
  s = (0.000666666666667 × s) ;
}
```

used by the programs. More precisely, we show that by using our tool, we approximate the results to be ever accurate and close to the results obtained under the double precision while using single precision.

In the light of these ideas, let us confirm our claims by means of a small examples such as Simpson's Rule. We start by briefly describing what our program computes, and then we give their listing before and after being transformed with our tool. Their accuracy is then discussed.

Simpson's Rule consists in a technique for numerical integration that approximates the computation of $\int_a^b f(x)\,\mathrm{d}x$. It uses a second order approximation of the function f by a quadratic polynomial P that takes three abscissa points a, b and m with $m = (a+b)/2$.

When integrating the polynomial, we approximate the integral of f on the interval $[x, x+h]$ ($h \in \mathbb{R}$ small) with the integral of P on the same interval. Formally, the smaller the interval is, the better the integral approximation is. Consequently, we divide the interval $[a, b]$ into subintervals $[a, a+h]$, $[a+h, a+2h]$, ... and then we sum the obtained values for each interval. We write:

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{3}\big[f(x_0) + 2\sum_{j=1}^{\frac{n}{2}-1} f(x_{2j}) + 4\sum_{j=1}^{\frac{n}{2}} f(x_{2j-1}) + f(x_n)\big], \quad (17)$$

where

- n is the number of subintervals of [a,b] with n is even,
- $h = (b-a)/n$ is the length of the subintervals,
- $x_i = a + i \times h$ for $i = 0, 1, \ldots, n-1, n$.

In our case, we have chosen the polynomial given in Equation (18). It is well-known by the specialists of floating-point arithmetic that the developed form of the polynomial evaluates very poorly close to a multiple root. This motivates our choice of the function f below for our experiments.

$$\begin{aligned} f &= (x-2.)^7 \\ &= x^7 - 14.\times x^6 + 84.\times x^5 - 280.\times x^4 + 560.\times x^3 - 672.\times x^2 + 448.\times x - 128. \end{aligned} \quad (18)$$

The listing corresponding of the implementation of the Simpson's Rule is described in Listing 1. Our tool optimizes the accuracy of the initial program described in Listing 1 by up to 99% depending on the entries. The transformed program is given in Listing 2. As detailed in Section 2.4, our tool has

- Created large expressions,
- Transformed them into more accurate expressions,
- Performed partial evaluation of the expressions,
- Split the transformed expressions,
- Assigned the transformed expressions to `TMP` variables.

This process has been applied inside and outside the loop.

The experimental results described hereafter compare the numerical accuracy of programs using single and double precision with a program transformed with our tool and running single precision only. Note that programs are compiled with the optimization level $-o0$ to avoid any optimization done by the compiler and additionally, we enforce the copy in the memory at each execution step by declaring all the variables as `volatile`, this avoids that values are kept in registers using more than 64 bits.

The results observed on Figure 4 when executing the initial program in single and double precision and the transformed program in single precision, demonstrate that our approach succeeds well to improve the accuracy. If we interest in the result of computations around the multiple root 2.0, we can see that the behavior of the transformed code is far

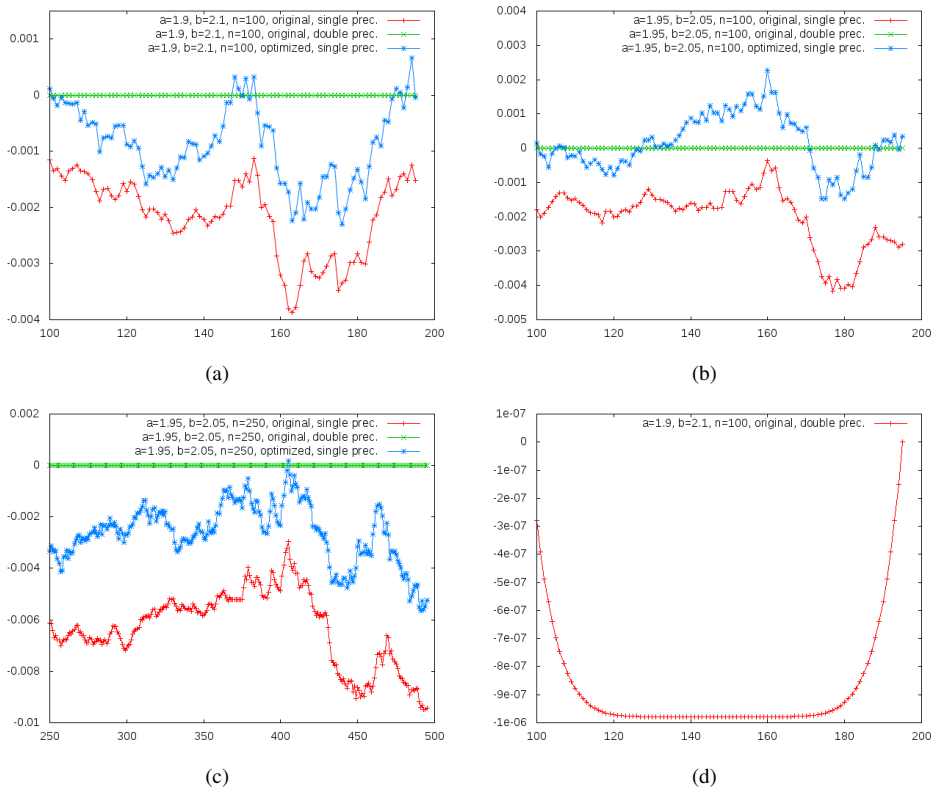

Figure 4 Simulation results of the Simpson's Rule with single, double precision and transformed program using our tool. The values of x and y axes correspond respectively to the value of n and s in Equation 19.

closer to the original program executed with a double precision than the single precision original program. This shows that single precision may suffices in many contexts.

In Figure 4, one can see the difference, for different values of the step $h > 0$, in the computation of

$$s = \int_a^{a+nh} f(x)\,\mathrm{d}x, \quad 0 \leq n < \frac{b-a}{h} \tag{19}$$

between the original program with both single and double precision and the transformed program (in single precision) in terms of numerical accuracy of computations. Obviously, the accuracy of the computations of the polynomial $f(x)$ depends on the values of x. The more the value of x is close to the multiple root, the worst the result is. For this reason, we make the interval $[a, b]$ vary by choosing $[a, b] \in \{[1.9, 2.1], [1.95, 2.05], [1.95, 2.05]\}$ and by choosing to split them in $n = 10$, $n = 100$ and $n = 250$ slices respectively for the application of Simpson's Rule. Concerning the results obtained, our tool states that the percentage of the optimization computed by the abstract semantics of Section 2.2 is up to 99.39%. This means that the bound (obtained by the technique of Section 2.2) on the numerical error of the computed values of the polynomial at any iteration is reduced by 99.39%.

Curve (d) of Figure 4 displays the function $f(x)$ at points $n = 1.9 + 0.02 \times i$, $100 \leq i \leq 200$. Next, if we take for example Curve (b) of Figure 4, we observe that our implementation is as accurate as the initial program in double precision for many values of n since it gives results very close to the double precision while working in single precision. Note that, for the $x - axis$ of Figure 4, we have chosen to observe only the interesting interval of n which is around the multiple root 2.0.

4 Convergence Acceleration

In this section, we study the impact of our program transformation on the convergence speed of classical iterative numerical methods. We consider four methods: Jacobi's method, Newton-Raphson Method, an iterative Gram-Schmidt method and an iterative method to compute the eigenvalues of a matrix. We give the listings of these methods before and after the transformation as well as the speed-ups obtained for each of them in terms of numbers of iteration saved.

4.1 Linear Systems of Equations

We start with a first case study concerning Jacobi's method (Atkinson, 1988) which consists of an iterative computation that solves linear systems of the form $\mathbf{Ax} = \mathbf{b}$. From this equation, we build a sequence of vectors ($\mathbf{x}^{(0)}$, $\mathbf{x}^{(1)}$, ... , $\mathbf{x}^{(k)}$, $\mathbf{x}^{(k+1)}$, ...) that converges towards the solution $\mathbf{x}^{(k)}$ of the system of linear equations. To compute $\mathbf{x}^{(k+1)}$, we use:

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)}}{a_{ii}}, \quad \mathbf{x}^{(k)} \text{ is known.} \tag{20}$$

The method iterates until $|x_i^{(k+1)} - x_i^{(k)}| < \epsilon$ for the desired x_i, $_{1\leq i\leq n}$. A sufficient condition for the stability of Jacobi's method is that

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}|. \tag{21}$$

Let us now examine how we can improve the convergence of Jacobi's method on the example given in Equation (22). This system is stable with respect to the sufficient condition of Equation (21) but it is close to be unstable in the sense that $\forall i$, $_{1\leq i\leq 4}$, $|a_{ii}| \approx \sum_{j=1, j\neq i}^{j=4} |a_{ij}|$.

$$\begin{pmatrix} 0.62 & 0.1 & 0.2 & -0.3 \\ 0.3 & 0.602 & -0.1 & 0.2 \\ 0.2 & -0.3 & 0.6006 & 0.1 \\ -0.1 & 0.2 & 0.3 & 0.601 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1.0/2.0 \\ 1.0/3.0 \\ 1.0/4.0 \\ 1.0/5.0 \end{pmatrix}. \tag{22}$$

To solve Equation (22) by Jacobi's method, we use the algorithm presented in Listing 3. This program is transformed with our tool by using the set of transformation rules described in Section 2.4. Note that, in the version of this program given to our tool, we have unfolded the body of the while loop twice. This makes it possible to rewrite more drastically the

Listing 3 Listing of the initial program of Jacobi's method.

```
eps = 10e-16; a11 = 0.61; a22 = 0.602; a33 = 0.6006; a44 = 0.601;
b1 = 0.5; b2 = 1.0/3.0; b3 = 0.25; b4 = 1.0/5.0;
while (e > eps) {
  x_n1 = (b1/a11) - (0.1/a11) × x2 - (0.2/a11) × x3 + (0.3/a11) × x4;
  x_n2 = (b2/a22) - (0.3/a22) × x1 + (0.1/a22) × x3 - (0.2/a22) × x4;
  x_n3 = (b3/a33) - (0.2/a33) × x1 + (0.3/a33) × x2 - (0.1/a33) × x4;
  x_n4 = (b4/a44) + (0.1/a44) × x1 - (0.2/a44) × x2 - (0.3/a44) × x3;
  e = x_n1 - x1;
  x1 = x_n1; x2 = x_n2;
  x3 = x_n3; x4 = x_n4;
}
```

Listing 4 Listing of the transformed program of Jacobi's method.

```
eps = 10e-16 ;
while (e > eps) {
  TMP_1 = (0.553709856035437 - (x1 × 0.498338870431894)) ;
  TMP_2 = (0.166112956810631 × x3) ;
  TMP_6 = (0.333000333000333 × x1) ;
  x_n1 = (((0.819672131147541 - (0.163934426229508 × ((TMP_1 + TMP_2)
       - (0.332225913621263 × x4)))) - (0.327868852459016 × (((0.416250416250416
       - TMP_6) + (0.4995004995005 × x2)) - (0.166500166500167 × x4))))
       + (0.491803278688525 × (((0.332778702163062 + (0.166389351081531 × x1))
       - (0.332778702163062 × x2)) - (0.499168053244592 × x3)))) ;
  x_n2 = (((0.553709856035437 - (0.498338870431894 × x_n1)) + (0.166112956810631
       × (((0.416250416250416 - TMP_6) + (0.4995004995005 × x2))
       - (0.166500166500167 × x4)))) - (0.332225913621263 × (((0.332778702163062
       + (0.166389351081531 × x1)) - (0.332778702163062 × x2))
       - (0.499168053244592 × x3)))) ;
  x_n3 = (((0.416250416250416 - (0.333000333000333 × x_n1)) + (0.4995004995005
       × x_n2)) - (0.166500166500167 × (((0.332778702163062 + (0.166389351081531
       × x1)) - (0.332778702163062 × x2)) - (0.499168053244592 × x3)))) ;
  x_n4 = (((0.332778702163062 + (0.166389351081531 × x_n1)) - (0.332778702163062
       × x_n2)) - (0.499168053244592 × x_n3)) ;
  e = (x_n4 - x4) ;
  x1 = x_n1 ;
  x2 = x_n2 ;
  x3 = x_n3 ;
  x4 = x_n4 ;
}
```

code by mixing the computations of both iterations. In this example, without unfolding, we win very few iterations and, obviously, if we unfold the body of the loop more than twice, our tool improves even more the accuracy at the price of a larger code. Note that in the examples of the next sections, we do not perform such an unfolding because our tool already optimizes significantly the original codes (results would be even better with unfolding).

The program corresponding to Jacobi's method after optimization is shown in Listing 4. Note that this code is rather not intuitive and could be very difficult to write by hand. Concerning the accuracy of the variables, our tool states that the percentage of the optimization computed by the abstract semantics of Section 2.2 is up to 44.5%. This means that the bound on the numerical error of the computed values of x_i, $_{1 \leq i \leq 4}$ at any iteration is reduced by 44.5%. The *target variable* at improving in this program is $\nu = \{x_i\}$, in other words, we improved the fourth variables (x_1, x_2, x_3, x_4) separately and we observed at each case the number of iterations required by the method to converge.

In Table 1, one can see the difference between the original and the transformed programs in term of the number of iterations needed to compute x_1, x_2, x_3 and x_4. Roughly speaking, about 15% less iterations are needed with the transformed code. Obviously, the fact that the body of the loop is unfolded twice, in the transformed code is taken into account in the computation of the number of iterations needed to converge.

4.2 Zero Finding

Newton-Raphson's Method (Atkinson, 1988) is a numerical method used to compute the successive approximations of the zeros of a real-valued function. In order to understand how this method works, let us start with the derivative $f'(x)$ of the function f which may be used to find the slope, and thus the equation of the tangent to the curve at a specified point. The method starts in an interval, for the equation $f(x) = 0$, in which there exists only one solution, the root a.

We choose a value u_0 close enough to a and then we build a sequence $(u_n)_{n\in\mathbb{N}}$ where u_{n+1} is obtained from u_n, as the abscissa of the meet point of the x-axis and the tangent at point $(u_n, f(u_n))$ to the function f. The final formula is given in Equation (23). Note that the computation stops when $|u_{n-1} - u_n| < \epsilon$.

$$u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}. \tag{23}$$

In general, Newton-Raphson's converges very quickly (quadratic convergence) but it may be slower if the computation of f or f' is inaccurate. For our case study, we have

x_i	Initial Number of iteration	Iterations Number after optimization	Difference	Percentage of Improvement
x_1	1891	1628	263	14.0
x_2	2068	1702	366	17.3
x_3	2019	1702	317	15.7
x_4	1953	1628	325	16.7

Table 1 Number of iterations of Jacobi's method before and after optimization to compute x_i, $1 \leq i \leq 4$.

Listing 5 Listing of the initial Newton-Raphson's program.

```
eps = 0.0005 ; e = 1.0 ; x = 0.0 ;
while (e > eps){
  f  = (x × x × x × x × x) − (10.0 × x × x × x × x) + (40.0 × x × x × x)
     − (80.0 × x × x) + (80.0 × x) − (32.0) ;
  ff = (5.0 × x × x × x × x) − (40.0 × x × x × x) + (120.0 × x × x)
     − (160.0 × x) + (80.0) ;
  x_n = x − (f / ff) ;
  e = (x − x_n) ;
  x = x_n ;
  if (e < 0.0) {
    e = (e × (−1.0)) ;
  }
  else {
    e = (e × 1.0) ;
  } ;
}
```

Listing 6 Listing of the transformed Newton-Raphson's program.

```
eps = 0.0005 ; e = 1.0 ; x = 0.0 ; x_n = 1.0 ;
while (e > eps){
  TMP_1 = (((((x × x) × x) × x) × x) − ((((10.0 × x) × x) × x) × x)) ;
  TMP_2 = ((x × x) × (40.0 × x)) ;
  TMP_3 = (80.0 × x) ;
  TMP_5 = (((5.0 × x) × x) × (x × x)) ;
  TMP_6 = ((x × x) × (40.0 × x)) ;
  TMP_7 = (120.0 × x) ;
  x_n = (x − (((((TMP_1 + TMP_2) − (TMP_3 × x)) + TMP_3) − 32.0)
      / ((((TMP_5 − TMP_6) + (TMP_7 × x)) − (160.0 × x)) + 80.0))) ;
  e = (x − x_n) ;
  x = x_n ;
  if (e < 0.0) {
    e = (e × (−1.0)) ;
  }
  else {
    e = (e × 1.0) ;
  } ;
}
```

chosen functions which are difficult to evaluate in the IEEE754 floating-point arithmetic. Let us consider the function $f(x) = (x-2)^5$. The developed formula of f and its derivative f' are:

$$f(x) = x^5 - 10.0 \times x^4 + 40x^3 - 80x^2 + 80x - 32, \tag{24}$$

$$f'(x) = 5x^4 - 40x^3 + 120x^2 - 160x + 80. \tag{25}$$

It is well-known from floating-point arithmetic experts that evaluating the developed form of a polynomial close to a multiple root may be quite inaccurate (Langlois and Louvet, 2007). Consequently, this example presents some numerical difficulties for Newton-Raphson's method which converges slowly in this case.

The algorithm corresponding to Equation (23) is given in Listing 5. We recognize the computation of $f(x)$ and its derivative $f'(x)$ called `ff`. When optimizing this program with our tool, we get the program of Listing 6. The accuracy of the x_n's is improved up to 1.53% following the semantics of Section 2.2.

The results given in Figure 5 show how much our tool optimizes the number of iterations needed to converge. Obviously, this number of iterations needed to converge to the solution with a given precision depends on the initial value x_0. We have experimented several initial values. We make x_0 go from 0 to 3 with a step of 0.1. The 30 results are presented in Figure 5. Due to the numerical inaccuracies, the number of iterations ranges from 10 to 1200, approximatively. It is always close to 10 with the transformed program.

4.3 Eigenvalue Computation

The Iterated Power Method is a method used to compute the largest eigenvalue of a matrix and the related eigenvector (Golub and Van Loan, 1996). We start by setting an arbitrary initial vector $\mathbf{x}^{(0)}$ containing a single non-zero element. Next, we build an intermediary vector $\mathbf{y}^{(1)}$ such that $\mathbf{A}\mathbf{x}^{(0)} = \mathbf{y}^{(1)}$. Then, we build $\mathbf{x}^{(1)}$ by re-normalizing $\mathbf{y}^{(1)}$ so that the selected component is again equal to 1. This process is repeated up to convergence. Optionally, we may change the reference vector if it converges to 0. Note that the renormalization factor converges to the largest eigenvalue and $\mathbf{x}$ converges to

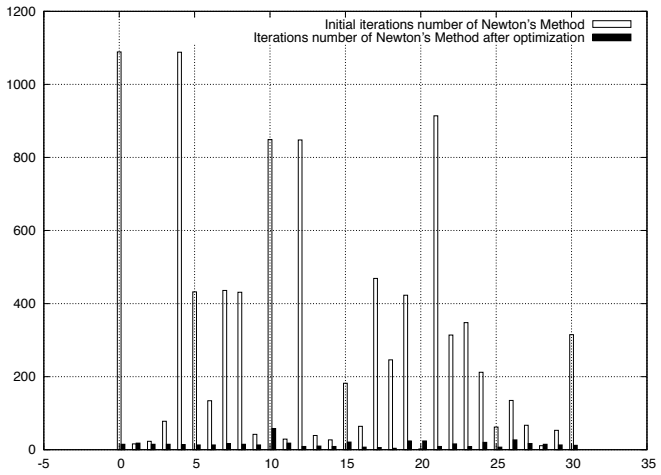

Figure 5 Number of iterations of the Newton-Raphson's Method before and after optimization for initial values ranging from 0 to 3 (30 runs with a step of 0.1).

the related eigenvector, under the conditions that the largest eigenvalue is unique and that all eigenvectors are independent. The convergence speed is proportional to the ratio between the two largest eigenvalues (in absolute value). For our experiments, let us take a square matrix $\mathbf{A}$ of dimension 4 with the eigenvector $(0.0\ \ 0.0\ \ 0.0\ \ 1.0)^T$ given on the Equation (22):

$$\mathbf{A} = \begin{pmatrix} \mathtt{d} & 0.01 & 0.01 & 0.01 \\ 0.01 & \mathtt{d} & 0.01 & 0.01 \\ 0.01 & 0.01 & \mathtt{d} & 0.01 \\ 0.01 & 0.01 & 0.01 & \mathtt{d} \end{pmatrix} \text{ with } \mathtt{d} \in [175.0, 200.0]. \tag{26}$$

By applying the Iterated Power Method, the first intermediary vector is

$$\begin{array}{l} \mathbf{A}\mathbf{x}^0 \quad = \mathbf{y}^1 \ , \\ \mathbf{A}{}^{y^1}\!/_{y_4^1} = \mathbf{y}^2 \ , \\ \mathbf{A}{}^{y^2}\!/_{y_4^2} = \mathbf{y}^3 \ , \\ \qquad \ldots \end{array} \tag{27}$$

To re-normalize this intermediary vector, we divide it by the last value $\mathtt{d}$, manner to have $y_4^{(1)}$ equal to 1.0. The new vector is: $(0.01/\mathtt{d}\ \ 0.01/\mathtt{d}\ \ 0.01/\mathtt{d}\ \ 1.0)^T$. We keep iterating with the new intermediary vector. We have to repeat the former operation on this new intermediary vector in order to re-normalize it. By repeating this process several times, the series converge to the eigenvector $(1.0\ \ 1.0\ \ 1.0\ \ 1.0)^T$.

The initial code of the iterated power method is given in Listing 7. Our tool has improved the error bounds computed by the semantics of Section 2.2 of up to 25.76%. The transformed code is given in Listing 8. The *target variable* at improving in this method is $\nu = \{v_1\}$.

When running this program, we observe significant improved results. In other words, the transformed implementation succeeds to reduce the numbers of iterations needed

to converge and accelerates the convergence speed of the iterative power method. The experimental results are summarized in Figure 6. The number of iterations is reduced by at least 475 iterations.

4.4 Iterative Gram-Schmidt Method

The Gram-Schmidt method is used to orthogonalize a set of non-zero vectors in a Euclidean or Hermitian space $\mathbb{R}^n$. This method takes as input a linear independent set of vectors $\mathbf{Q} = \{\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j\}$. The output is the orthogonal set of vectors $\mathbf{Q'} = \{\mathbf{q'}_1, \mathbf{q'}_2, \ldots, \mathbf{q'}_j\}$, with $1 \leq j \leq n$ (Abdelmalek, 1971; Golub and Van Loan, 1996; Hernandez et al., 2007). The process followed by Gram-Schmidt method starts by defining the projection:

$$proj_{\mathbf{q'}}(\mathbf{q}) = \frac{\langle \mathbf{q}, \mathbf{q'} \rangle}{\langle \mathbf{q'}, \mathbf{q'} \rangle} \mathbf{q'}. \tag{28}$$

In Equation (28), $\langle \mathbf{q}, \mathbf{q'} \rangle$ is the dot product of the vectors $\mathbf{q}$ and $\mathbf{q'}$. It means that the vector $\mathbf{q}$ is projected orthogonally onto the line spanned by the vector $\mathbf{q'}$. The normalized vectors are $\mathbf{e}_j = \frac{\mathbf{q'}_j}{||\mathbf{q'}_j||}$ where $||\mathbf{q'}_j||$ consists of the norm of the vector $\mathbf{q'}_j$. Explicitly, Gram-Schmidt process can be written as:

$$\begin{aligned}
\mathbf{q'}_1 &= \mathbf{q}_1, \\
\mathbf{q'}_2 &= \mathbf{q}_2 - proj_{\mathbf{q'}_1}(\mathbf{q}_2), \\
&\vdots \\
\mathbf{q'}_j &= \mathbf{q}_j - \sum_{j=1}^{j-1} proj_{\mathbf{q'}_j}(\mathbf{q}_j).
\end{aligned}$$

In general, Gram-Schmidt method is numerically stable and it is not necessary to use an iterative algorithm. However, important numerical errors may arise when the vectors become more and more linearly dependent. In this case iterative algorithms yield better

Listing 7 Listing of the initial iterated power method.

```
eps = 0.0005; d = 175.0; v1 = 0.0; v2 = 0.0; v3 = 0.0; v4 = 1.0; a41 = 0.01;
a44 = d; a11 = d; a12 = 0.01; a13 = 0.01; a14 = 0.01; a21 = 0.01; a22 = d;
a42 = 0.01; e = 1.0; a23 = 0.01; a24 = 0.01; a31 = 0.01; a32 =  0.01; a33 = d;
a34 = 0.01; a43 = 0.01;
while (e >  eps) {
   vx = a11 × v1 + a12 × v2 + a13 × v3 + a14 × v4 ;
   vy = a21 × v1 + a22 × v2 + a23 × v3 + a24 × v4 ;
   vz = a31 × v1 + a32 × v2 + a33 × v3 + a34 × v4 ;
   vw = a41 × v1 + a42 × v2 + a43 × v3 + a44 × v4 ;
   v1 = vx / vw ;
   v2 = vy / vw ;
   v3 = vz / vw ;
   v4 =  1.0 ;
   e = 1.0 − v1;
   if (v1 < 1.0) {
     e = 1.0 − v1 ;
   };
   else {
     e = v1 − 1.0;
   };
}
```

Listing 8 Listing of the transformed iterated power method.

```
eps = 0.0005; d = 175.0; v1 = 0.0; v2 = 0.0; v3 = 0.0; v4 = 1.0; e = 1.0;
while (e > eps) {
  vx = ((((0.01 × v4) + (0.01 × v2)) + (0.01 × v3)) + (d × v1)) ;
  vy = ((((0.01 × v1) + (0.01 × v4)) + (0.01 × v3)) + (d × v2)) ;
  vz = ((((0.01 × v4) + (0.01 × v2)) + (0.01 × v1)) + (d × v3)) ;
  vw = ((((0.01 × v2) + (0.01 × v1)) + (0.01 × v3)) + (d × v4)) ;
  v1 = (vx / vw) ;
  v2 = (vy / vw) ;
  v3 = (vz / vw) ;
  v4 = 1.0 ;
  e = (1.0 − v1) ;
  if (v1 < 1.0) {
    e = 1.0 − v1 ;}
  else { e = v1 − 1.0 ;}
}
```

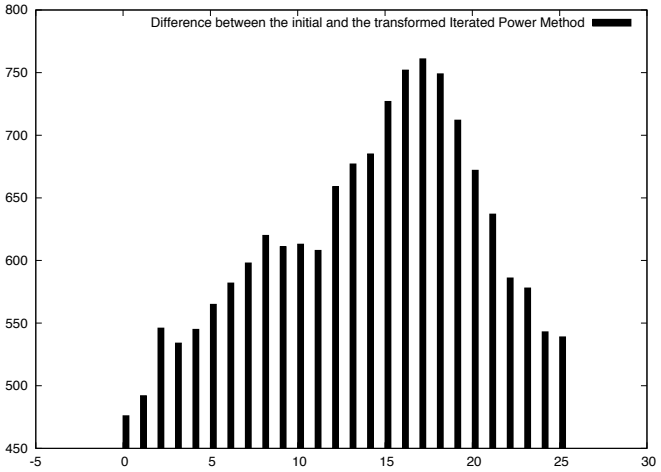

Figure 6 Difference between numbers of iterations of initial and transformed Iterated Power Method (tests done for $\mathtt{d} \in [175, 200]$ with a step of 1).

results, as for example the algorithm of Listing 9 which repeats the orthogonalization step until the ratio $\frac{||\mathbf{q'}_j||_2}{||\mathbf{q}_j||_2}$ becomes small enough (Hernandez et al., 2007). First, it starts by computing the orthogonal projection of span$\big(\{\mathbf{q_1}, \mathbf{q_2}, \mathbf{q_3}\}\big)$. Then, it substracts this projection from the original vector and then normalizes the result to obtain $\mathbf{q}_3$, *i.e.*, span$\big(\{\mathbf{q_1}, \mathbf{q_2}, \mathbf{q_3}\}\big)$ = span$\big(\{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}\}\big)$ and $\mathbf{q}_3$ is orthogonal to $\mathbf{q}_1$, $\mathbf{q}_2$. We assume that $r_{jj} > 0$.

To understand how this algorithm works, let us take for example a set of vectors in $\mathbb{R}^3$ that we aim at orthogonalizing.

$$Q_n = \left\{ \mathbf{q}_1 = \begin{pmatrix} 1/7\mathtt{n} \\ 0 \\ 0 \end{pmatrix}, \mathbf{q}_2 = \begin{pmatrix} 0 \\ 1/25\mathtt{n} \\ 0 \end{pmatrix}, \mathbf{q}_3 = \begin{pmatrix} 1/2592 \\ 1/2601 \\ 1/2583 \end{pmatrix} \right\}. \tag{29}$$

For our experiments, we have chosen the values of $\mathtt{n}$ ranging from 1 to 10.

In Listing 10, we give the transformed iterative Gram-Schmidt algorithm generated by our tool. By applying our techniques to the iterative Gram-Schmidt algorithm presented

Listing 9 Listing of the initial iterative Gram-Schmidt program.

```
Q11 = 1.0 / 7n; Q12 = 0.0; Q13 = 0.0; Q21 = 0.0;
Q22 = 1.0 / 25n; Q23 = 0.0; Q31 = 1.0 / 2592.0;
Q32 = 1.0 / 2601.0; Q33 = 1.0 / 2583.0;
e = 10.0; eps = 0.000005; qj1 = Q31; qj2 = Q32;
qj3 = Q33;  r1 = 0.0; r2 = 0.0; r3 = 0.0;
r = qj1 × qj1 + qj2 × qj2 + qj3 × qj3;
rold = sqrt(r);
while( e >  eps)  {
  h1 = Q11 × qj1 + Q21 × qj2 + Q31 × qj3;
  h2 = Q12 × qj1 + Q22 × qj2 + Q32 × qj3;
  h3 = Q13 × qj1 + Q23 × qj2 + Q33 × qj3;
  qj1 = qj1 − (Q11 × h1 + Q12 × h2 + Q13 × h3);
  qj2 = qj2 − (Q21 × h1 + Q22 × h2 + Q23 × h3);
  qj3 = qj3 − (Q31 × h1 + Q32 × h2 + Q33 × h3);
  r1 = r1 + h1 ;
  r2 = r2 + h2 ;
  r3 = r3 + h3 ;
  r = qj1 × qj1 + qj2 × qj2 + qj3 × qj3 ;
  rjj = sqrt(r);
  e =  1.0 − (rjj / rold) ;
  if (e < 0.0) {
    e = −e ;
  };
  rold = rjj ;
}
```

Listing 10 Listing of the transformed iterative Gram-Schmidt program.

```
Q11 = 1.0 / 7n; Q12 = 0.0; Q13 = 0.0; Q21 = 0.0; Q22 = 1.0/25n; Q23 = 0.0;
Q31 = 1.0 / 2592.0; Q32 = 1.0 / 2601.0; Q33 = 1.0 / 2583.0; eps = 0.000005;
qj1 = Q31; qj2 = Q32; qj3 = Q33; r1 = 0.0; r2 = 0.0; r3 = 0.0; e = 10.0;
r = qj1 × qj1 + qj2 × qj2 + qj3 × qj3;

rold = sqrt(r);
while ( e > eps)  {
  TMP_6 = (qj1 × qj3);
  TMP_14 = (qj2 × qj3);
  qj1 = (qj1 − ((0.142857142857 × (((qj1 × qj3)) + (0.142857142857 × qj1)))));
  qj2 = (qj2 − (((0.04 × (((0.0 × qj1) + (qj2 × qj3)) + (0.04 × qj2))))));
  qj3 = (qj3 − (((qj2 × ((TMP_14) + (0.04 × qj2))) + (qj3 + (qj3 × qj3))))
      + (qj1 × (((qj1 × qj3)) + (0.142857142857 × qj1)))));
  r1 = (r1 + ((TMP_6) + (0.142857142857 × qj1)));
  r2 = (r2 + ((TMP_14) + (0.04 × qj2)));
  r3 = (r3 + ((qj3 × qj3)));
  r = qj1 × qj1 + qj2 × qj2 + qj3 × qj3;
  rjj = sqrt(r);
  e =  1.0 − (rjj / rold);
  if (e < 0.0) {
    e = −e ;
  };
  rold = rjj ;
}
```

previously in Listing 9, we show in Figure 7 that the transformed algorithm converges faster than the initial one by up to 14.5%. Note that in this method, the variable at optimizing is $\nu = \{r\}$.

Method	Original Code Execution Time (s)	Transformed Code Execution Time (s)	Percentage Improvement	Mean on n Runs
Simpson	$8.40 \cdot 10^{-2}$	$2.40 \cdot 10^{-2}$	13.3%	10^2
Jacobi	$1.49 \cdot 10^{-4}$	$0.38 \cdot 10^{-4}$	74.5%	10^4
Newton-Raphson	$1.34 \cdot 10^{-3}$	$0.02 \cdot 10^{-3}$	98.4%	10^4
Eigenvalue	$4.50 \cdot 10^{-2}$	$3.07 \cdot 10^{-2}$	31.6%	10^3
Gram-Schmidt	$1.99 \cdot 10^{-1}$	$1.70 \cdot 10^{-1}$	14.5%	10^2

Table 2 Execution time measurements of programs of Section 3 and 4.

5 Speedups and Flops

We have shown in the former sections that we optimize the data-type formats of Simpson's Rule as well as the number of iterations of our four iterative numerical algorithms. In this section, we provide complementary benchmarks concerning speedups and the number of floating-point operations. Our objective is to check that the gains in the number of iterations are not annealed by overheads in the execution time or by other side effects for example due to the compiler.

We have chosen to observe the speedups to compute s in Simpson's Rule, x_4 for Jacobi's method, and for $x_0 = 3$ for Newton-Raphson's method. We have taken $d = 200$ for the iterated power method and $\mathbf{q}_{11} = \frac{1}{63}$ and $\mathbf{q}_{22} = \frac{1}{225}$ for iterative Gram-Schmidt method.

If we focus on measuring the execution time of the four programs before and after optimization, we observe that the percentage of improvement is rather important. If we take for example Simpson's method, we remark that we reduce its execution time by 13.3%. We give in Table 2 the speedups results obtained for the four methods. These results are very interesting to emphasize the usefulness of our tool and its ability to improve accuracy and execution time simultaneously.

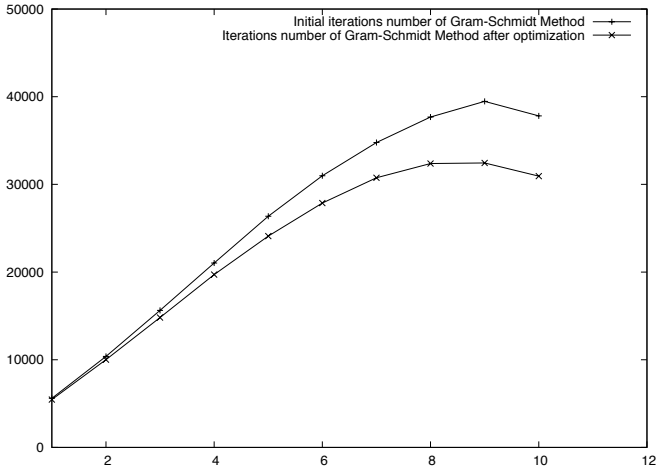

Figure 7 Iterations number of initial and transformed iterative Gram-Schmidt Method for the family $(Q_n)_n$ of vectors, $1 \leq n \leq 10$.

We have also counted the number of floating-point operations (flops) in the original and transformed codes. For each method, we count the number of additions and subtractions as well as the number of products and divisions for a single iteration and for the total number of iterations required in each case to converge. In Table 3, we give:

- $\oplus_o$, is the number of additions and subtractions per iteration of the original program,
- $\oplus_t$, is the number of additions and subtractions per iteration of the transformed program,
- $\bigoplus_o$, is the total number of additions and subtractions for all the iterations of the original program,
- $\bigoplus_t$, is the total number of additions and subtractions for all the iterations of the transformed program,
- $\otimes_o$, is the number of multiplications and divisions per iteration of the original program,
- $\otimes_t$, is the number of multiplications and divisions per iteration of the transformed program,
- $\bigotimes_o$, is the number of multiplications and divisions of the total number for all the iterations of the original program,
- $\bigotimes_t$, is the number of multiplications and divisions of the total number for all the iterations of the transformed program,
- $\%$, is the percentage of improvement of the number of iterations needed by method to converge.

The results show that by transforming programs, we improve not only their numerical accuracy but we reduce the number of iterations of methods needed to converge. The original `Simpson's` methods requires 249 iterations to converge while the transformed `Simpson's` program needes only 99 iterations. More precisely, if we take the `Simpson's` method program, we shown in Table 3 that the convergence speed of the transformed program is accelerated by 53.61% for the addition and the substraction and by 67.78% for the multiplication and the division. These results are coherent with the observed execution times.

Method	$\oplus_o$	$\oplus_t$	$\bigoplus_o$	$\bigoplus_t$	$\%$
Simpson	36	42	8964	4158	53.61
Jacobi	13	15	25389	24420	3.81
Newton-Raphson	11	11	3465	132	96.19
Eigenvalue	15	15	694080	685995	1.16
Gram-Schmidt	21	19	791364	715996	9.52
Method	$\otimes_o$	$\otimes_t$	$\bigotimes_o$	$\bigotimes_t$	$\%$
Simpson	116	94	28884	9306	67.78
Jacobi	28	14	54684	22792	58.32
Newton-Raphson	27	26	8505	312	96.33
Eigenvalue	19	19	879168	868927	1.16
Gram-Schmidt	22	20	712316	647560	9.09

Table 3 Floating-point operations needed by programs of Section 4 to converge.

6 Conclusion

In this article, we have shown the usefulness side-effects of our program transformation to improve the accuracy of programs. The first one allows one to work in a lower precision and obtain results close to the higher precision when transforming programs using our tool. The second side-effect concern the acceleration of the convergence of numerical iterative methods.

A significant interest would be to extend the current work with a case study concerning real size numerical applications. The study described in former work (Damouche et al., 2015c) is a first step in this direction. We are currently working on real-size problems coming from the domain of non-smooth contact mechanics. It is well-known that floating-point computations may impact the numerical quality of the results of simulations as well as their execution times, especially when dealing with sensitive functions such as the ones that we find in non-smooth contact mechanics. In this direction, we aim at optimizing the accuracy of numerical simulations on the time required by their iterative methods to converge. Indeed, we will apply `Salsa` on selected examples coming from non-smooth and non-linear problems and on algorithms for linear systems. For our case study, we have taken an example of a linear contact between a deformable body and a foundation.

Another interesting perspective consists in extending our work to perform the high performance computing programs. In this direction, we aim at solving problems due to the numerical accuracy like the order of operation of a distributed system. We are interesting also in studying the compromise execution time, computation performances, numerical accuracy and the convergence acceleration of numerical methods. A key issue is to study the impact of accuracy optimization on the convergence time of distributed numerical algorithms like the ones used usually for high performance computing. In addition, still about distributed systems, an important issue concerns the reproducibility of the results: different runs of the same application yield different results due to the variations in the order of evaluation of the mathematical expression. We would like to study how our technique could improve reproducibility.

References

Abdelmalek, N. (1971) *Roundoff Error Analysis for Gram-Schmidt Method and Solution of Linear Least Squares Problem*, BIT, Volume 11, pp.345–368.

ANSI/IEEE, (2008) *IEEE Standard for Binary Floating-point Arithmetic*, Std 754-2008, ANSI/IEEE.

Atkinson, K.-E. (1988) *An Introduction to Numerical Analysis*, John Wiley & Sons, Second Edition, USA, 0-471-62489-6.

Bertrane, J. and Cousot, P. and Cousot, R. and Feret, J. and Mauborgne, L. and Miné, A. and Rival, X. (2011) *Static analysis by abstract interpretation of embedded critical software*, ACM SIGSOFT Software Engineering Notes, Volume 36, number 1, pp.1–8.

Cousot, P. and Cousot, R. (1977) *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, POPL'77, ACM, pp.238–252.

Cousot, P. and Cousot, R. (2002) *Systematic Design of Program Transformation Frameworks by Abstract Interpretation*, Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, ACM Press, New York, NY, pp.178–190.

Cytron, R. and Gershbein, R. (1993) *Efficient Accomodation of May-Alias Information in SSA Form*, PLDI'93, ACM, 0-89791-598-4, pp.36–45.

Damouche, N. and Martel, M. and Chapoutot, A. (2015) *Impact of Accuracy Optimization on the Convergence of Numerical Iterative Methods*, LOPSTR'15, LNCS, 9527, Springer, pp.1–18.

Damouche, N. and Martel, M. and Chapoutot, A. (2015) *Intra-procedural Optimization of the Numerical Accuracy of Programs*, FMICS'15, LNCS, Volume 9128, Springer, pp.31–46.

Damouche, N. and Martel, M. and Chapoutot, A. (2015) *Optimizing the Accuracy of a Rocket Trajectory Simulation by Program Transformation* , CF'15, ACM, pp.40:1–40:2.

Damouche, N. and Martel, M. and Chapoutot, A. (2016) *Data-Types Optimization for Floating-Point Formats by Program Transformation*, CoDIT, IEEE, pp.576–581.

Darulova, E. and Kuncak, V. (2014) *Sound compilation of reals*, POPL'14, ACM, pp.235–248.

Delmas, D. and Goubault, E. and Putot, S. and Souyris, J. and Tekkal, K. and Védrine, F. (2009) *Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software*, FMICS'09, pp.53–69.

Goldberg, D. (1991) *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Comput. Surv., Volume 23, number 1, pp.5–48.

Golub, G.-H. and Van Loan, C.-F. (1996) *Matrix computations (3. ed.)*, Johns Hopkins University Press, Baltimore.

Goubault, E. (2013) *Static Analysis by Abstract Interpretation of Numerical Programs and Systems, and FLUCTUAT*, SAS'13, LNCS, Volume 7935, Springer, pp.1–3.

Hankin, E. (1994) *Lambda Calculi A Guide For Computer Scientists*, Clarendon Press, Oxford, 0-19-853841-3.

Hernandez, V. and Roman, J.-E. and Tomas, A. and Vidal, V. (2007) *Orthogonalization routine in SLEPc Technical Report STR-1*, Polytechnic University of Valencia, STR1.

Ioualalen, A. and Martel, M. (2012) *A New Abstract Domain for the Representation of Mathematically Equivalent Expressions*, Volume 7460, SAS'12, LNCS, Springer, pp.75–93.

Langlois, Ph. and Louvet, N. (2007) *How to Ensure a Faithful Polynomial Evaluation with the Compensated Horner Algorithm*, ARITH-18, IEEE Computer Society, pp.141–149.

Martel, M. (2006) *Semantics of roundoff error propagation in finite precision calculations*, Higher-Order and Symbolic Computing, Volume 19, number 1, pp.7–30.

Muller, J.-M. and Brisebarre, N. and De Dinechin, F. and Jeannerod, C.-P. and Lefèvre, V. and Melquiond, G. and Revol, N. and Stehlé, D. and Torres, S. (2010) *Handbook of Floating-Point Arithmetic*, Birkhäuser Boston , 978-0-8176-4704-9.

Panchekha, P. and Sanchez-Stern, A. and Wilcox, J.-R. and Tatlock, Z. (2015) *Automatically improving accuracy for floating point expressions*, PLDI'15, ACM, pp.1–11.

Solovyev, A. and Jacobsen, C. and Rakamaric, Z. and Gopalakrishnan, G. (2015) *Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions*, FM'15, LNCS, Volume 9109, Springer, pp.532–550.

Tate, R. and Stepp, M. and Tatlock, Z. and Lerner, S. (2011) *Equality Saturation: A New Approach to Optimization*, Logic Methods in Computer Science, Volume 7, number 1, pp.264–276.