Numerical program optimisation 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
Email: nasrine.damouche@univ-perp.fr
Email: matthieu.martel@univ-perp.fr
*Corresponding author

Alexandre Chapoutot

ENSTA ParisTech, U2IS,
Université Paris-Saclay,
828 bd des Maréchaux,
Palaiseau, 91762, France
Fax: +33-1-81-87-20-71
Email: 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 optimise the numerical accuracy of computations by minimising 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 optimising 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 optimisation; convergence acceleration.


Copyright © 2018 Inderscience Enterprises Ltd.
1 Introduction

Floating-point numbers, whose specification is given by the IEEE754 standard (ANSI/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 nonlinear function.

Secondly, we show that by optimising 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 emphasise the effect of the finite precision. Programs are compiled with the default optimisation level of the compiler –o0. We have tried other levels of optimisation without observing significant changes in our results.
This article is organised 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 optimise arithmetic expressions (Ioualalen and Martel, 2012) and programs (Damouche et al., 2015b). These transformations are implemented in the tool that we use to optimise 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 formalises 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:

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

where

- \( s \) is the sign \( \in \{-1, 1\} \)
- \( b \) is the basis, \( b = 2 \)
- \( m \) is the mantissa, \( m = d_0.d_1...d_{p-1} \) 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 normalised whenever \( d_0 \neq 0 \). Normalisation avoids multiple representations of the same number. IEEE754 standard specifies some particular values for \( p, e_{\text{min}} \) and \( e_{\text{max}} \), which are summarised in Table 1, as well as denormalised numbers which are floating-point numbers with \( d_0 = d_1 = \ldots = d_k = 0, k < p - 1 \) and \( e = e_{\text{min}} \). Denormalised numbers make underflow gradual (Goldberg, 1991). Finally, the following special values also are defined:

- Not a number (NaN) 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 \( \uparrow_{-\infty}, \uparrow_{+\infty}, \uparrow_0, \uparrow_{\text{e}} \) and \( \uparrow_{\text{o}} \). The semantics of the elementary operations specified by the IEEE754 standard is given by equation (2):

\[
x \circ_{r} y = \uparrow_{r} (x \cdot y), \quad \text{with} \quad \uparrow_{r} : \mathbb{R} \rightarrow \mathbb{F},
\]

where a floating-point operation, denoted by \( \circ_{r} \), is computed using the rounding mode \( r \in \{ \uparrow_{-\infty}, \uparrow_{+\infty}, \uparrow_0, \uparrow_{\text{e}} \} \) and \( \circ \in \{ +, -, \times, / \} \) 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 \( \downarrow_{r} : \mathbb{R} \rightarrow \mathbb{R} \) that returns the round-off error. We have:

\[
\downarrow_{r} (x) = x - \uparrow_{r} (x).
\]

### 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 \((x, \mu) \in \mathbb{F} \times \mathbb{R} = \mathbb{E}\) where \( x \) denotes the floating-point number used by the machine and \( \mu \) denotes the exact error attached to \( F \), 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 \( \pi \) which cannot be represented exactly on the machine. In single precision, the binary and decimal representations of \( \pi \) respectively are \( 1.10010100001111011101011 \times 2^{1} \) and \( 3.1415928 \). In our semantics, \( \pi \) is represented by the value \( v = (\uparrow_{\text{o}} (\pi), \downarrow_{\text{o}} (\pi)) = (3.1415928, (\pi - 3.1415928)) \).
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 $(x^\varepsilon, \mu^\varepsilon) \in \mathbb{E}^2$, we have $x^\varepsilon$ the interval corresponding to the range of the values and $\mu^\varepsilon$ the interval of errors on $x^\varepsilon$. This value abstracts a set of concrete values $\{(x, \mu) : x \in x^\varepsilon \text{ and } \mu \in \mu^\varepsilon\}$ by intervals in a component-wise way. We now introduce the semantics of arithmetic expressions on $\mathbb{E}^2$. We approximate an interval $x^\varepsilon$ with real bounds by an interval based on floating-point bounds, denoted by $(\uparrow x^\varepsilon)$. Here bounds are rounded to the nearest, see equation (4):

$$\uparrow: ([x, x]) = \left[\uparrow- (x), \uparrow+ (x)\right]. \quad (4)$$

We denote by $\downarrow$ the function that abstracts the concrete function $\downarrow$. It over approximates the set of exact values of the error $\downarrow(x) = x - \uparrow(x)$. Every error associated to $x \in ([x, x])$ is included in $\downarrow([x, x])$. We also have for a rounding mode to the nearest:

$$\downarrow([x, x]) = [-y, y] \text{ with } y = \frac{1}{2} \upulp(\max(|x|, |x|)). \quad (5)$$

Formally, the unit in the last place, denoted by $\upulp(x)$, 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 subtraction and multiplication over $\mathbb{E}^2$, for other operations see Martel (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 $(x_1^\varepsilon, \mu_1^\varepsilon) \times (x_2^\varepsilon, \mu_2^\varepsilon)$. Note that, the semantics of the elementary operations on $\mathbb{E}$ is defined in a former work in Martel (2006):

$$\begin{align*}
(x_1^\varepsilon, \mu_1^\varepsilon) + (x_2^\varepsilon, \mu_2^\varepsilon) &= \left(\uparrow (x_1^\varepsilon + x_2^\varepsilon), \mu_1^\varepsilon + \mu_2^\varepsilon + \downarrow (x_1^\varepsilon + x_2^\varepsilon)\right), \quad (6) \\
(x_1^\varepsilon, \mu_1^\varepsilon) - (x_2^\varepsilon, \mu_2^\varepsilon) &= \left(\uparrow (x_1^\varepsilon - x_2^\varepsilon), \mu_1^\varepsilon + \mu_2^\varepsilon + \downarrow (x_1^\varepsilon - x_2^\varepsilon)\right), \quad (7) \\
(x_1^\varepsilon, \mu_1^\varepsilon) \times (x_2^\varepsilon, \mu_2^\varepsilon) &= \left(\uparrow (x_1^\varepsilon \times x_2^\varepsilon), x_1^\varepsilon \times \mu_2^\varepsilon + x_2^\varepsilon \times \mu_1^\varepsilon + \mu_1^\varepsilon \times \mu_2^\varepsilon + \downarrow (x_1^\varepsilon \times x_2^\varepsilon)\right). \quad (8)
\end{align*}$$

### 2.3 Accuracy improvement

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

$$\text{Expr} \in e ::= id | cst | e + e | e - e | e \times e | e \div e. \quad (9)$$

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 \( p_1 \times p_2 \) is an APEG, where \( p_1 \) and \( p_2 \) are APEGs and \( \times \) is a binary operator.
3. A box \( \{ p_1, ..., p_n \} \) is an APEG, where \( \times \) is a commutative and associative operator and the \( p_i \), \( 1 \leq i \leq n \), are APEGs.
4. A non-empty set \( \{ p_1, ..., p_n \} \), 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 3) contains many APEGs \( p_1, ..., p_n \) then one of the \( p_i \), \( 1 \leq i \leq n \), may be selected in order to build an expression. A box \( \{ p_1, ..., p_n \} \) represents any parsing of the expression \( p_1 \times ... \times p_n \). 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 a 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, in 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:

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 \( p_1 \times p_2 \) then \( \mathcal{A}(p) = \bigcup_{e_1 \in \mathcal{A}(p_1), e_2 \in \mathcal{A}(p_2)} e_1 \times e_2 \).
3. If \( p \) is a box \( \{ p_1, ..., p_n \} \) then \( \mathcal{A}(p) \) contains all the parsings of \( e_1 \times ... \times e_n \), for all \( e_1 \in \mathcal{A}(p_1), ..., e_n \in \mathcal{A}(p_n) \).
4. If \( p \) is an equivalence class \( \{ p_1, ..., p_n \} \) 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:

\[
\mathcal{A}(p) = \left\{ ((a + a) + b) \times c, ((a + b) + a) \times c, ((b + a) + a) \times c, \right.
\]
\[
((2 + a) + b) \times c, c \times ((a + a) + b), c \times ((a + b) + a),
\]
\[
c \times ((b + a) + a), c \times (2 + a) + b, (a + a) \times c + b \times c,
\]
\[
\left. (2 \times a) \times c + b \times c, b \times c + (a + a) \times c, b \times c + (2 \times a) \times c \right\}
\] (10)

In their article on EPEGs, Tate et al. (2011) use rewriting rules to extend the structure up to saturation. 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 recognised. 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).

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

![APEG diagram](image)

Figure 3 Some rules for APEG construction by pattern matching

![Pattern matching rules](image)

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:

\[
\text{Com} \ni e ::= \text{id} = e \mid c_1 ; c_2 \mid \text{if}_e \ e \ \text{then} \ c_1 \ \text{else} \ c_2 \mid \text{while}_e \ e \ \text{do} \ c_1 \ \text{nop}.
\]  

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 \( \Phi \)-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):

\[ x = a; \]
\[ \text{if}(x > 1.0) \text{then} \]
\[ x = x \times 2.0; \]
\[ \text{else} \]
\[ x = x / 2.0; \]
\[ z = x \]

The SSA form of the program given in equation (12) is:

\[ x_1 = a; \]
\[ \text{if} (x_1 > 1.0) \text{then} \]
\[ x_2 = x_1 \times 2.0; \]
\[ \text{else} \]
\[ x_3 = x_1 / 2.0; \]
\[ \Phi(x_4, x_2, x_3); \]
\[ z = x_4 \]

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 \( \Phi \)-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 optimise 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)
- \( \delta \) is an environment \( \delta : \mathcal{V} \rightarrow \text{Expr} \) which maps variables to expressions. Intuitively, this environment records the expressions assigned to variables in order to inline them later on in larger expressions
- \( C \in \text{Ctx} \) 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 optimising
- \( \beta \subseteq \mathcal{V} \) is a list of assigned variables that should not be removed from the code.

Initially, \( \beta = \{ \nu \} \), i.e., the target variable \( \nu \) must not be removed.

The environment \( \delta \) 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 (1) the variable \( \nu \) of some assignment \( \nu = e \) does not exist in the domain of \( \delta \) if (2) \( \nu \notin \beta \) and if (3) \( \nu \neq \nu \) then we memorise \( e \) in \( \delta \) and we remove the assignment from the program. Otherwise, if one of the conditions (1), (2) or (3) is not satisfied then we rewrite this assignment by inlining the variables saved in \( \delta \) 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 optimising and \(a = 0.1, b = 0.01, c = 0.001\) and \(d = 0.0001\) are constants.

\[
\langle x = a + b; y = c + d; z = x + y, \delta, [], [z] \rangle
\]

\[
\rightarrow, \langle \text{nop}; y = c + d; z = x + y, \delta' = \delta[x \mapsto a + b], x = a + b; [], [z] \rangle
\]

\[
\rightarrow, \langle \text{nop}; \text{nop}; z = x + y, \delta'' = \delta'[y \mapsto c + d], x = a + b; y = c + d; [], [z] \rangle
\]

\[
\rightarrow, \langle \text{nop}; \text{nop}; z = ((d + c)b) + a, \delta'', x = a + b; y = c + d; [], [z] \rangle
\]

In equation (14), the environment \(\delta\) and the context \(C\) are initialy empty and the list \(\beta\) contains the reference variable \(z\). We remove the variable \(x\) and memorise it in \(\delta\). So, the line corresponding to the variable discarded is replaced by \(\text{nop}\) 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 \(\delta\) 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:

\[
\begin{align*}
  & x_1 = a; \\
  & \text{if}_0(y_1, y_2, x_2) x_1 > 1.0 \text{ then} \\
  & \quad y_1 = x_1 + 2.0; \\
  & \text{else} \\
  & \quad y_2 = x_1 - 1.0; \\
  & \quad \nu = y_3.
\end{align*}
\]

(15)

First of all, \(x_1\) is stored in \(\delta\). Then, we transform recursively the new program:

\[
\begin{align*}
  & \text{if}_0(y_1, y_2, x_2) x_1 > 1.0 \text{ then} \\
  & \quad y_1 = x_1 + 2.0; \\
  & \text{else} \\
  & \quad y_2 = x_1 - 1.0; \\
  & \quad \nu = y_3.
\end{align*}
\]

(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 \(\beta\) in order to avoid an infinite loop in the transformation.
Numerical program optimisation by automatic improvement

For a sequence $c_1; c_2$, the first command $c_1$ is transformed into $c_1'$ in the current environment $\delta$, $C$, $\nu$ and $\beta$ 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 $\delta'$ and the list $\beta'$ 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 $\delta$. In this case, the body is transformed in the context $C[\text{while } \Phi e \text{ do } []]$ where $C$ is the context of the loop. A second rule builds the list $V = \text{Var}(e) \cup \text{Var}(\Phi)$ where $\text{Var}(\Phi)$ is the list of variables read and written in the $\Phi$ 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 $\delta$ and added to $\beta$. The resulting command is obtained by transforming $c'$; while $\Phi e$ do $c$ with $\delta'$ and $\beta \cup V$.

3 Data-types optimisation

We have shown in Section 2.4 how to optimise 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.

Listing 1 Listing of the initial Simpson’s rule

```c
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) - 14.0 * (x * x * x * x * x * x))
        + 84.0 * (x * x * x * x * x * x) - 280.0 * (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 * x * x))
        + 84.0 * (x * x * x * x * x * x * x * x * x) - 280.0 * (x * x * x * x * x * x * x * x * x)
        + 560.0 * (x * x * x * x * x * x * x * x * x) - 672.0 * (x * x * x * x * x * x * x * 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 * x * x))
            + 84.0 * (x * x * x * x * x * x * x * x * x) - 280.0 * (x * x * x * x * x * x * x * x * x)
\begin{align*}
    +560.0 \times (x \times x \times x) - 672.0 \times (x \times x) + 448.0 \times x - 128.0; \\
    s = s + 4.0 \times f; \\
    i = i + 1.0; \\
\end{align*}

\}

\i = 2.0;

\} \text{ while } (i < n - 1) \{ 

\begin{align*}
    x &= a + (i \times h); \\
    f &= ((x \times x \times x \times x \times x \times x \times x \times x) \\
    &\quad + 14.0 \times (x \times x \times x \times x \times x) \\
    &\quad + 84.0 \times (x \times x \times x \times x \times x) \\
    &\quad + 560.0 \times (x \times x \times x) - 14.0 \times (x \times x \times x \times x \times x) \\
    &\quad + 672.0 \times (x \times x) + 448.0 \times x - 128.0; \\
    s &= s + 2.0 \times f; \\
    i &= i + 1.0; \\
\end{align*}

\}

\begin{align*}
    s &= s \times (h / 3.0); \\
\end{align*}

We emphasise the efficiency of our implementation in terms of improving the data-types 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 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 \( \int_a^b f(x)dx \). 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 \([a, a + h], [a + h, a + 2h], \ldots\) and then we sum the obtained values for each interval. We write:

\begin{equation}
\int_a^b f(x)dx \approx \frac{h}{3} \left[ f(x_0) + 2 \sum_{j=1}^{n-1} f(x_{2j}) + 4 \sum_{j=1}^{n-1} f(x_{2j-1}) + f(x_n) \right],
\end{equation}

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 \( i = 0, 1, \ldots, n - 1, n \).
Listing 2  Listing of the transformed Simpson rule

int main () { 
  TMP_3 = 6.859;
  TMP_1 = (((((((TMP_3 \times 1.9) \times 1.9) \times 1.9) \times 1.9) \times 1.9) + (84.0 \times ((6.859 \times 1.9) \times 1.9)))));
  TMP_2 = (1.9 \times (280.0 \times TMP_3));
  TMP_15 = 9.261000000000001;
  TMP_13 = (((((((TMP_15 \times 2.1) \times 2.1) \times 2.1) \times 2.1) – (14.0 \times ((((TMP_15 \times 2.1) \times 2.1) \times 2.1)))) + (84.0 \times ((9.261000000000001 \times 2.1) \times 2.1))));
  TMP_14 = (2.1 \times (280.0 \times TMP_15));
  TMP_27 = 3.61;
  TMP_25 = (((((((TMP_27 \times 1.9) \times 1.9) \times 1.9) \times 1.9) \times 1.9) – (14.0 \times (((((TMP_27 \times 1.9) \times 1.9) \times 1.9) \times 1.9)))));
  TMP_26 = (1.9 \times (159.599999999999994 \times TMP_3));
  TMP_32 = ((TMP_3 \times 1.9);
  i = 1.0;
  s = ((((((TMP_1 – TMP_2) + (560.0 \times TMP_3)) – 2425.920000000000073) + 851.199999999999932) – 128.0) + ((((((TMP_13 – TMP_14) + (560.0 \times TMP_15)) – 2963.520000000000437) + 940.800000000000068) – 128.0));
  f = ((((((TMP_25 + TMP_26) – (280.0 \times TMP_32)) + (560.0 \times (TMP_27 \times 1.9))) – (672.0 \times (TMP_27))) + 851.199999999999932) – 128.0);
  x = 2.1;
  while (i < 100.0) {
    x = (1.9 + (i \times 0.002));
    TMP_37 = (x \times x);
    TMP_35 = ((((((TMP_37 \times x) \times x) \times x) \times x) \times x) – (14.0 \times (((((TMP_37 \times x) \times x) \times x) \times x))));
    TMP_36 = (84.0 \times (((((TMP_37 \times x) \times x) \times x) \times x)));
    TMP_42 = (x \times (TMP_37 \times x));
    f = ((((((TMP_35 + TMP_36) – (280.0 \times TMP_42)) + (560.0 \times (TMP_37 \times x))) – (672.0 \times (TMP_37))) + (448.0 \times x)) – 128.0);
    s = (s + (4.0 \times f));
    i = (i + 1.0);
  };
  i = 2.0;
  while (i < 99.) {
    x = (1.9 + (i \times 0.002));
    TMP_47 = (x \times x);
    TMP_45 = ((((((TMP_47 \times x) \times x) \times x) \times x) \times x) \times x) – (14.0 \times (((((TMP_47 \times x) \times x) \times x) \times x)))};
\[
\begin{align*}
\text{TMP_{46}} &= (84.0 \times (((\text{TMP}_{47} \times x) \times x) \times x)); \\
\text{TMP_{52}} &= (x \times (\text{TMP}_{47} \times x)); \\
f &= (((((\text{TMP}_{45} + \text{TMP}_{46}) - (280.0 \times \text{TMP}_{52})) + (560.0 \times (\text{TMP}_{47} \times x))) \\
& \quad - (672.0 \times \text{TMP}_{47})) + (448.0 \times x)) - 128.0); \\
s &= (s + (2.0 \times f)); \\
i &= (i + 1.0); \\
\} \\
\text{s} &= (0.000666666666667 \times s); \\
\}
\end{align*}
\]

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:

\[
f = (x - 2.0)^7 \\
= x^7 - 14.0 \times x^6 + 84.0 \times x^5 - 280.0 \times x^4 + 560.0 \times x^3 - 672.0 \times x^2 + 448.0 \times x - 128.
\]

The listing corresponding of the implementation of the Simpson’s rule is described in Listing 1. Our tool optimises 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 \( \text{TMP} \) 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 optimisation level \(-o0\) to avoid any optimisation done by the compiler and additionally, we enforce the copy in the memory at each execution step by declaring all the variables as \textit{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 behaviour of the transformed code is far 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.
Figure 4  Simulation results of the Simpson’s rule with single, double precision and transformed program using our tool (see online version for colours)

(a)

(b)

Note: The values of $x$ and $y$ axes correspond respectively to the value of $n$ and $s$ in equation (19).
Figure 4  Simulation results of the Simpson’s rule with single, double precision and transformed program using our tool (continued) (see online version for colours)

Note: The values of $x$ and $y$ axes correspond respectively to the value of $n$ and $s$ in equation (19).
In Figure 4, one can see the difference, for different values of the step $h > 0$, in the computation of:

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

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 optimisation 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 $Ax = b$. From this equation, we build a sequence of vectors $(x^{(0)}, x^{(1)}, \ldots, x^{(k)}, x^{(k+1)}, \ldots)$ that converges towards the solution $x^{(k)}$ of the system of linear equations. To compute $x^{(k+1)}$, we use:

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

$$a_{ij}$$
The method iterates until \( |x_i^{(k+1)} - x_i^{(k)}| < \epsilon \) for the desired \( x_{i \in S_n} \). A sufficient condition for the stability of Jacobi’s method is that:

\[
|a| = \sum_{j \neq i} |a_{ij}| > \sum_{j = 1, j \neq i} |a_{ij}|. \tag{21}
\]

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:

\[
\forall i \in \{1, 4\}, |a| = \sum_{j \neq i} |a_{ij}| = \begin{bmatrix} 0.62 & 0.1 & 0.2 & -0.3 \\ 0.3 & 0.602 & -0.1 & 0.2 \\ 0.2 & -0.3 & 0.6006 & 0.1 \\ -0.1 & 0.2 & 0.3 & 0.601 \end{bmatrix}.
\tag{22}
\]

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 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 optimises significantly the original codes (results would be even better with unfolding).
Numerical program optimisation by automatic improvement

Table 1  Number of iterations of Jacobi’s method before and after optimisation to compute $x_{i\leq 4}$

<table>
<thead>
<tr>
<th>$x_i$</th>
<th>Initial number of iteration</th>
<th>Iterations number after optimisation</th>
<th>Difference</th>
<th>Percentage of improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>1,891</td>
<td>1,628</td>
<td>263</td>
<td>14.0</td>
</tr>
<tr>
<td>$x_2$</td>
<td>2,068</td>
<td>1,702</td>
<td>366</td>
<td>17.3</td>
</tr>
<tr>
<td>$x_3$</td>
<td>2,019</td>
<td>1,702</td>
<td>317</td>
<td>15.7</td>
</tr>
<tr>
<td>$x_4$</td>
<td>1,953</td>
<td>1,628</td>
<td>325</td>
<td>16.7</td>
</tr>
</tbody>
</table>

Listing 4  Listing of the transformed program of Jacobi’s method

```c
eps = 10e – 16;
while (e > eps) {
    TMP_1 = (0.553709856035437 – (x1 × 0.4 98338 8704 31894));
    TMP_2 = (0.166112956810631 × x3);
    TMP_6 = (0.333000333000333 × x1);
    x_n1 = (((0.819672131147541 – (0.163934426229508 × ((TMP_1 + TMP_2)
    – (0.32225913621263 × x4))) – (0.32786885245916 × (((0.416250416250416
    – TMP_6) + (0.49950049995005 × 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) + (0.49950049995005 × 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)) + (0.49950049995005
    × 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;
}
```

The program corresponding to Jacobi’s method after optimisation 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 optimisation 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 $x_{1,15054}$ 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 $|u_{n+1} - u_n| < \epsilon$:

$$u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}.$$  \hspace{1cm} (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 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:

$$f(x) = x^5 - 10.0 \times x^4 + 40.0 \times x^3 - 80.0 \times x^2 + 80.0 \times x - 32,$$

$$f'(x) = 5x^4 - 40.0 \times x^3 + 120.0 \times x^2 - 160.0 \times x + 80.$$  \hspace{1cm} (24) \hspace{1cm} (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 recognise the computation of $f(x)$ and its derivative $f'(x)$ called $ff$. When optimising 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 optimises the number of iterations needed to converge. Obviously, this number of iterations needed to converge to
Numerical program optimisation by automatic improvement

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 1,200, approximatively. It is always close to ten with the transformed program.

**Listing 5** Listing of the initial Newton-Raphson’s program

```plaintext
code
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 * 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

```plaintext
code
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)));
    TMP_2 = ((x * x) * (40.0 * x));
    TMP_3 = (80.0 * x);
    TMP_4 = (((5.0 * x) * x) * (x * x));
    TMP_5 = ((x * x) * (40.0 * x));
    TMP_6 = (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);
    };
}
```
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 $x^{(0)}$ containing a single non-zero element. Next, we build an intermediary vector $y^{(1)}$ such that $Ax^{(0)} = y^{(1)}$. Then, we build $x^{(1)}$ by re-normalising $y^{(1)}$ so that the selected component is again equal to one. This process is repeated up to convergence. Optionally, we may change the reference vector if it converges to 0. Note that the renormalisation factor converges to the largest eigenvalue and $x$ converges to 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 $A$ of dimension four with the eigenvector $(0.0 \ 0.0 \ 0.0 \ 1.0)^T$ given on the equation (22):

$$A = \begin{pmatrix} d & 0.01 & 0.01 & 0.01 \\ 0.01 & d & 0.01 & 0.01 \\ 0.01 & 0.01 & d & 0.01 \\ 0.01 & 0.01 & 0.01 & d \end{pmatrix} \quad \text{with } d \in [175.0, 200.0]. \quad (26)$$

By applying the iterated power method, the first intermediary vector is:
Numerical program optimisation by automatic improvement

\[ \mathbf{A} \mathbf{x}^0 = \mathbf{y}^1, \]
\[ \mathbf{A} \mathbf{y}_1^1 / \mathbf{y}_3^1 = \mathbf{y}^2, \]
\[ \mathbf{A} \mathbf{y}_2^2 / \mathbf{y}_5^2 = \mathbf{y}^3, \]

(27)

To re-normalise this intermediary vector, we divide it by the last value \( d \), manner to have \( y_j^{(1)} \) equal to 1.0. The new vector is: \((0.01/d 0.01/d 0.01/d 1.0)^T\). We keep iterating with the new intermediary vector. We have to repeat the former operation on this new intermediary vector in order to re-normalise 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 \( v = \{v_i\} \).

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 summarised in Figure 6. The number of iterations is reduced by at least 475 iterations.

**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;
vz = a21 × v1 + a22 × v2 + a23 × v3 + a24 × v4;
vw = a31 × v1 + a32 × v2 + a33 × v3 + a34 × 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

\[ \text{eps} = 0.0005; \, d = 175.0; \, v1 = 0.0; \, v2 = 0.0; \, v3 = 0.0; \, v4 = 1.0; \, e = 1.0; \]

\[ \text{while } (e > \text{eps}) \{ \]
\[ v_x = (((0.01 \times v_4) + (0.01 \times v_2)) + (0.01 \times v_3)) + (d \times v_1)); \]
\[ v_y = (((0.01 \times v_1) + (0.01 \times v_4)) + (0.01 \times v_3)) + (d \times v_2)); \]
\[ v_z = (((0.01 \times v_4) + (0.01 \times v_2)) + (0.01 \times v_1)) + (d \times v_3)); \]
\[ v_w = (((0.01 \times v_2) + (0.01 \times v_1)) + (0.01 \times v_3)) + (d \times v_4)); \]
\[ v_1 = (v_x / v_w); \]
\[ v_2 = (v_y / v_w); \]
\[ v_3 = (v_z / v_w); \]
\[ v_4 = 1.0; \]
\[ e = (1.0 - v_1); \]
\[ \text{if } (v_1 < 1.0) \{ \]
\[ e = 1.0 - v_1; \} \]
\[ \text{else } \{e = v_1 - 1.0 ;\} \]
\[ \} \]

Figure 6  Difference between numbers of iterations of initial and transformed iterated power method (tests done for \( d \in [175, 200] \) with a step of 1)
4.4 Iterative Gram-Schmidt method

The Gram-Schmidt method is used to orthogonalise 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 $Q = \{q_1, q_2, ..., q_j\}$. The output is the orthogonal set of vectors $Q' = \{q'_1, q'_2, ..., q'_j\}$, 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:

$$\text{proj}_q(q) = \frac{\langle q, q' \rangle}{\langle q', q \rangle} q'.$$

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

$$q'_1 = q_1,$nQ
$$q'_2 = q_2 - \text{proj}_{q_1}(q_2),$$
$$\vdots$$
$$q'_j = q_j - \sum_{i=1}^{j-1} \text{proj}_{q_i}(q_j).$$

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 results, as for example the algorithm of Listing 9 which repeats the orthogonalisation step until the ratio $\|q'_j\|/\|q_j\|$ becomes small enough (Hernandez et al., 2007). First, it starts by computing the orthogonal projection of span ($\{q_1, q_2, q_3\}$). Then, it subtracts this projection from the original vector and then normalises the result to obtain $q_3$, i.e., span ($\{q_1, q_2, q_3\}$) = span ($\{x_1, x_2, x_3\}$) and $q_3$ is orthogonal to $q_1, q_2$. We assume that $rij > 0$.

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

$$Q_n = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} 1/7n \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} 1/25n \\ 0 \end{bmatrix}, q_3 = \begin{bmatrix} 1/2592 \\ 1/2601 \\ 1/2583 \end{bmatrix}.$$  \hspace{1cm} (29)

For our experiments, we have chosen the values of $n$ ranging from 1 to 10.
Listing 9  Listing of the initial iterative Gram-Schmidt program

\[ Q_{11} = 1.0 / 7n; Q_{12} = 0.0; Q_{13} = 0.0; Q_{21} = 0.0; \]
\[ Q_{22} = 1.0 / 25n; Q_{23} = 0.0; Q_{31} = 1.0 / 2592.0; \]
\[ Q_{32} = 1.0 / 2601.0; Q_{33} = 1.0 / 2583.0; \]
\[ e = 10.0; eps = 0.000005; qj1 = Q_{31}; qj2 = Q_{32}; \]
\[ qj3 = Q_{33}; r1 = 0.0; r2 = 0.0; r3 = 0.0; \]
\[ r = qj1 \times qj1 + qj2 \times qj2 + qj3 \times qj3; \]
\[ rold = \sqrt{r}; \]
\[ \text{while } (e > eps) \{ \]
\[ h1 = Q_{11} \times qj1 + Q_{21} \times qj2 + Q_{31} \times qj3; \]
\[ h2 = Q_{12} \times qj1 + Q_{22} \times qj2 + Q_{32} \times qj3; \]
\[ h3 = Q_{13} \times qj1 + Q_{23} \times qj2 + Q_{33} \times qj3; \]
\[ qj1 = qj1 - (Q_{11} \times h1 + Q_{21} \times h2 + Q_{31} \times h3); \]
\[ qj2 = qj2 - (Q_{12} \times h1 + Q_{22} \times h2 + Q_{32} \times h3); \]
\[ qj3 = qj3 - (Q_{13} \times h1 + Q_{23} \times h2 + Q_{33} \times h3); \]
\[ r1 = r1 + h1; \]
\[ r2 = r2 + h2; \]
\[ r3 = r3 + h3; \]
\[ r = qj1 \times qj1 + qj2 \times qj2 + qj3 \times qj3; \]
\[ rjj = \sqrt{r}; \]
\[ e = 1.0 - (rjj / rold); \]
\[ \text{if } (e < 0.0) \{ \]
\[ e = -e; \]
\[ \}; \]
\[ \text{rold = rjj; } \]

Listing 10  Listing of the transformed iterative Gram-Schmidt program

\[ Q_{11} = 1.0 / 7n; Q_{12} = 0.0; Q_{13} = 0.0; Q_{21} = 0.0; Q_{22} = 1.0 / 25n; Q_{23} = 0.0; \]
\[ Q_{31} = 1.0 / 2592.0; Q_{32} = 1.0 / 2601.0; Q_{33} = 1.0 / 2583.0; eps = 0.000005; \]
\[ qj1 = Q_{31}; qj2 = Q_{32}; qj3 = Q_{33}; r1 = 0.0; r2 = 0.0; r3 = 0.0; e = 10.0; \]
\[ r = qj1 \times qj1 + qj2 \times qj2 + qj3 \times qj3; \]
\[ \text{rold = } \sqrt{r}; \]
\[ \text{while } (e > eps) \{ \]
\[ \text{TMP.6 = } (qj1 \times qj3); \]
\[ \text{TMP.14 = } (qj2 \times qj3); \]
\[ qj1 = qj1 - ((0.14285714285 \times (((qj1 \times qj3)) + (0.14285714285 \times qj1)))); \]
\[ qj2 = qj2 - ((0.04 \times (((0.0 \times qj1) + (qj2 \times qj3)) + (0.04 \times qj2)))); \]
\[ qj3 = qj3 - (((qj2 \times ((\text{TMP.14}) + (0.04 \times qj2))) + (qj3 + (qj3 \times qj3))) \]
\[ + ((qj1 \times (((qj1 \times qj3)) + (0.14285714285 \times qj1)))); \]
\[ r1 = (r1 + ((\text{TMP.6}) + (0.14285714285 \times qj1))); \]
Numerical program optimisation by automatic improvement

\[ r_2 = (r_2 + (\text{TMP}_{14}) + (0.04 \times qj_2)); \]
\[ r_3 = (r_3 + (qj_3 \times qj_3)); \]
\[ r = qj_1 \times qj_1 + qj_2 \times qj_2 + qj_3 \times qj_3; \]
\[ rjj = \text{sqrt} (r); \]
\[ e = 1.0 - (rjj / rold); \]
\[ \text{if (} e < 0.0 \text{)} { \]
\[ e = -e; \]
\[ } \]
\[ rold = rjj; \]
\}

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 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 optimising is \( v = \{r\} \).

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Original code execution time (s)</th>
<th>Transformed code execution time (s)</th>
<th>Percentage Improvement</th>
<th>Mean on ( n ) runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simpson</td>
<td>( 8.40 \cdot 10^{-2} )</td>
<td>( 2.40 \cdot 10^{-2} )</td>
<td>13.3%</td>
<td>( 10^2 )</td>
</tr>
<tr>
<td>Jacobi</td>
<td>( 1.49 \cdot 10^{-4} )</td>
<td>( 0.38 \cdot 10^{-4} )</td>
<td>74.5%</td>
<td>( 10^4 )</td>
</tr>
<tr>
<td>Newton-Raphson</td>
<td>( 1.34 \cdot 10^{-3} )</td>
<td>( 0.02 \cdot 10^{-3} )</td>
<td>98.4%</td>
<td>( 10^4 )</td>
</tr>
<tr>
<td>Eigenvalue</td>
<td>( 4.50 \cdot 10^{-2} )</td>
<td>( 3.07 \cdot 10^{-2} )</td>
<td>31.6%</td>
<td>( 10^3 )</td>
</tr>
<tr>
<td>Gram-Schmidt</td>
<td>( 1.99 \cdot 10^{-1} )</td>
<td>( 1.70 \cdot 10^{-1} )</td>
<td>14.5%</td>
<td>( 10^2 )</td>
</tr>
</tbody>
</table>
5 Speedups and flops

We have shown in the former sections that we optimise 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 $q_{11} \frac{1}{63}$ and $q_{22} \frac{1}{225}$ for iterative Gram-Schmidt method.

If we focus on measuring the execution time of the four programs before and after optimisation, 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 emphasise the usefulness of our tool and its ability to improve accuracy and execution time simultaneously.

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
- $\oplus_o$, is the total number of additions and subtractions for all the iterations of the original program
- $\oplus_t$, 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
- $\otimes_o$, is the number of multiplications and divisions of the total number for all the iterations of the original program
- $\otimes_t$, 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
Table 3 Floating-point operations needed by programs of Section 4 to converge

<table>
<thead>
<tr>
<th>Method</th>
<th>$\oplus_o$</th>
<th>$\oplus_i$</th>
<th>$\otimes_o$</th>
<th>$\otimes_i$</th>
<th>$%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simpson</td>
<td>36</td>
<td>42</td>
<td>8,964</td>
<td>4,158</td>
<td>53.61</td>
</tr>
<tr>
<td>Jacobi</td>
<td>13</td>
<td>15</td>
<td>25,389</td>
<td>24,420</td>
<td>3.81</td>
</tr>
<tr>
<td>Newton-Raphson</td>
<td>11</td>
<td>11</td>
<td>3,465</td>
<td>132</td>
<td>96.19</td>
</tr>
<tr>
<td>Eigenvalue</td>
<td>15</td>
<td>15</td>
<td>694,080</td>
<td>685,995</td>
<td>1.16</td>
</tr>
<tr>
<td>Gram-Schmidt</td>
<td>21</td>
<td>19</td>
<td>791,364</td>
<td>715,996</td>
<td>9.52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>$\otimes_o$</th>
<th>$\otimes_i$</th>
<th>$%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simpson</td>
<td>116</td>
<td>94</td>
<td>67.78</td>
</tr>
<tr>
<td>Jacobi</td>
<td>28</td>
<td>14</td>
<td>58.32</td>
</tr>
<tr>
<td>Newton-Raphson</td>
<td>27</td>
<td>26</td>
<td>96.33</td>
</tr>
<tr>
<td>Eigenvalue</td>
<td>19</td>
<td>19</td>
<td>1.16</td>
</tr>
<tr>
<td>Gram-Schmidt</td>
<td>22</td>
<td>20</td>
<td>9.09</td>
</tr>
</tbody>
</table>

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 needed 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 subtraction and by 67.78% for the multiplication and the division. These results are coherent with the observed execution times.

6 Conclusions

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 concerns 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 optimising 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 nonlinear 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 optimisation 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
Numerical program optimisation by automatic improvement


