Impact of Accuracy Optimization on the Convergence of Numerical Iterative Methods[‡]

Nasrine Damouche[1], Matthieu Martel[1], Alexandre Chapoutot[2]

[1] University of Perpignan Via Domitia, LAMPS, France
[2]ENSTA ParisTech, Palaiseau, France
nasrine.damouche@univ-perp.fr matthieu.martel@univ-perp.fr
alexandre.chapoutot@ensta-paristech.fr

Abstract. Among other objectives, rewriting programs serves as a useful technique to improve numerical accuracy. However, this optimization is not intuitive and this is why we switch to automatic transformation techniques. We are interested in the optimization of numerical programs relying on the IEEE754 floating-point arithmetic. In this article, our main contribution is to study the impact of optimizing the numerical accuracy of programs on the time required by numerical iterative methods to converge. To emphasize the usefulness of our tool, we make it optimize several examples of numerical methods such as Jacobi's method, Newton-Raphson's method, etc. We show that significant speedups are obtained in terms of number of iterations, time and flops.

Keywords: Program Transformation, Floating-Point Numbers, IEEE754 Standard, Numerical Analysis, Convergence Acceleration.

1 Introduction

A few decades ago, program transformation techniques have been successfully applied to specialize programs by partial evaluation [16]. For example, the performances of Knuth-Morris-Pratt's algorithm were reached by specialized versions of the naive, quadratic, pattern matching algorithm [4]. Other killer applications of partial evaluation were ranging from ray-tracing [16] to communication protocol optimization [22]. In this context, partial evaluation was used to optimize the execution time of programs. Our current work seeks another grail, namely the optimization of the numerical accuracy of computations carried out in the IEEE754 floating-point arithmetic [2, 23]. As for partial evaluation, we perform source to source transformations guided by partial information on the data used at run-time [14]. In our case, we need ranges for the input variables of the programs, obtained by abstract interpretation of their codes [5]. In former articles, we have shown how our techniques make it possible to improve the accuracy of

[‡]This work was supported by the ANR Project ANR-12-INSE-0007 "CAFEIN".

various algorithms coming from control theory or from numerical analysis [8, 9, 15]. In this article, we study the effect of our transformation on the convergence of well-known iterative numerical methods such as Newton-Raphson's or Jacobi's method [17]. We show that more accurate implementations obtained by automatic program transformation converge much faster than the original ones. In other words, less iterations are needed to reach a result with a given accuracy. As a consequence, improving the accuracy significantly improves the performances of this class of programs, bringing us back to the concerns of partial evaluation.

In former work we have shown how to optimize automatically intra-procedural programs [8]. To optimize programs, we use static analysis by abstract interpretation [5, 11] to over-approximate the roundoff errors as well as a set of rewriting rules for the transformation itself, applied to programs that are written in SSA Form [7]. We have experimented our tool to improve the numerical accuracy of small control command programs (e.g. PID and lead-lag controllers) and numerical procedures (trapeze rule and Runge-Kutta methods [8]). We have also demonstrated the efficiency of our tool to optimize slightly larger codes like a rocket trajectory simulation code of about $\mathcal{O}(100)$ lines of code [9].

Our main contribution in this article is to show that our technique improves the execution time of programs by increasing their numerical accuracy. By optimizing programs 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, time and total number of floating-point operations (flops).

In Section 2, we discuss how we compute the error on the numerical accuracy as well as the basic techniques used to rewrite programs. In Section 3, we detail the programs that we want to optimize. We give the programs before and after optimization together with experimental results. We conclude in Section 4.

2 Program Transformation for Numerical Accuracy

In this section, we first introduce the method that we use to compute the errors on the numerical accuracy. In sections 2.2 and 2.3, we also recall the transformation techniques used to optimize the numerical accuracy of expressions and programs and which are detailed in [8, 15]. All the material introduced in this section is used in the tool that we use to optimize the programs of Section 3.

2.1 Floating-Point Arithmetic and Error Bound Computation

The floating-point arithmetic is defined by the IEEE754 Standard [2, 23]. Floating-point numbers are used to encode real numbers. However, because they are a finite representation of their mathematical cousins, roundoff errors arise during computations. A floating-point number x is defined by

$$x = s \cdot (d_0.d_1 \ldots d_{p-1}) \cdot \beta^e = s \cdot m \cdot \beta^{e-p+1} \qquad (1)$$

where $s \in \{-1, 1\}$ is the sign, $m = d_0 d_1 \dots d_{p-1}$ is the mantissa with digits $0 \leq d_i < \beta$, $0 \leq i \leq p-1$, p is the precision and e is the exponent, $e_{min} \leq e \leq e_{max}$. The IEEE754 Standard specifies several formats for the floating-point numbers by providing specific values for p, β, e_{min} and e_{max}. It also defines some rounding modes, towads $+\infty$, $-\infty$, 0 and to the nearest. Our transformation technique, introduced in sections 2.2 and 2.3, is independent of the selected rounding mode and, in this article, we assume that all the floating-point computations are done by using the rounding mode to the nearest. Let us write $\uparrow_{+\infty}$, $\uparrow_{-\infty}$, $\uparrow_0$ and $\uparrow_\sim$ the rounding functions, the IEEE754 Standard defines the semantics of the elementary operations by:

$$x \circledast_r y = \uparrow_r (x * y) \tag{2}$$

where $\circledast_r \in \{+, -, \times, \div\}$ is computed by using the rounding mode r and $*$ denotes an exact operation. Because of the roundoff errors, the results of the computations are not exact. For example, let us consider two functions f and g which are mathematically equivalent. We have $f(x) = x^2 - 2.0 \times x + 1.0$ and $g(x) = (x - 1.0) \times (x - 1.0)$. If we compute $f(0.999)$ we get $1.00000000002875566e^{-6}$ and if we compute g of the same value, we obtain $1.00000000000000186e^{-6}$. On small computations, we have obtained already different results.

We present now the computation of errors on the numerical accuracy of arithmetic expressions [19]. These errors are stored in an abstract value [5] using 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 $(x^\sharp, \mu^\sharp) \in E^\sharp$, 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 $x^\sharp$ abstracts a set of concrete values $\{(x, \mu) \ : \ x \in x^\sharp \text{ and } \mu \in \mu^\sharp\}$ by intervals in a component-wise way. We introduce now the semantics of arithmetic expressions on $E^\sharp$. We approximate an interval $x^\sharp$ with real bounds by an interval based on floating-point bounds, denoted by $\uparrow^\sharp_\sim (x^\sharp)$. Here bounds are rounded to the nearest (see Equation (3)).

$$\uparrow^\sharp_\sim [(\underline{x}, \overline{x})] = [\uparrow_\sim (\underline{x}), \uparrow_\sim (\overline{x})]. \tag{3}$$

In the other direction, we have the function $\downarrow^\sharp_\sim$ that abstracts the concrete function $\downarrow_\sim$. It computes the exact value of the error $\downarrow_\sim (x) = x - \uparrow_\sim (x)$. Every error associated to $x \in [\underline{x}, \overline{x}]$ is included in $\downarrow^\sharp_\sim [(\underline{x}, \overline{x})]$. We have

$$\downarrow^\sharp_\sim [(\underline{x}, \overline{x})] = [-y, y] \quad \text{with} \quad y = \frac{1}{2}\text{ulp}\big(\max(|\underline{x}|, |\overline{x}|)\big). \tag{4}$$

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 (5) and (6) give the semantics of the addition and multiplication over $E^\sharp$. If we sum two floating-point numbers, we may add the errors generated by the operands to the error produced by the roundoff of the result. When multiplying two floating-point numbers, the semantics is given by the development of $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$.

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = \big(\uparrow^\sharp_\sim (x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp_\sim (x_1^\sharp + x_2^\sharp) \big), \tag{5}$$

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

This semantics is used to select the most accurate expression, in the sense that it minimizes $\mu^\sharp$ during the transformation of expressions introduced in the next section. To analyze statically a full program, we use a standard abstract interpretation of commands [5, 19] with the abstract domain $E^\sharp$ of values.

2.2 Transformation of Expressions

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

$$\text{Expr} \ni e ::= id \mid cst \mid e + e \mid e - e \mid e \times e \mid e \div e. \tag{7}$$

Here, we briefly present former work [15, 20, 25] to semantically transform [6] 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 1. When an equivalence class (denoted by a dotted ellipse in Figure 1) 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 1, 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 $\{p\}$ of expressions contained inside an APEG p is defined inductively as follows: (1) If p is a value v or a variable x then $(p) = \{v\}$ or $(p) = \{x\}$, (2) If p is an expression $p_1 * p_2$ then $(p) = \bigcup_{e_1} \in (p_1),\ e_2 \in (p_2) e_1 * e_2$, (3) If p is a box $\boxed{*(p_1, \ldots, p_n)}$ then (p) contains all the parsings of $e_1 * \ldots * e_n$

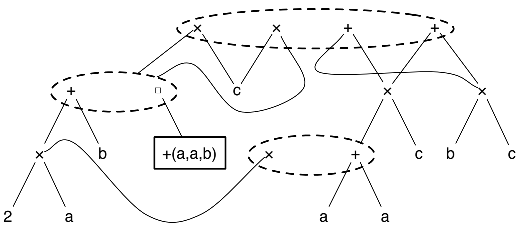


Fig. 1. APEG for the expression $e = \big((a + a) + c\big) \times c$.

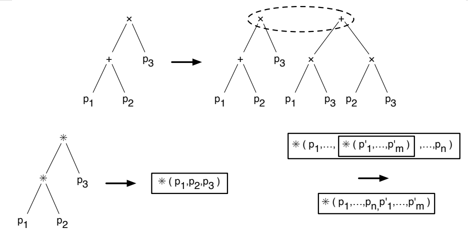


Fig. 2. Some rules for APEG construction by pattern matching.

for all $e_1 \in (p_1), \ldots, e_n \in (p_n)$ and (4) If p is an equivalence class $\{p_1, \ldots, p_n\}$ then $(p) = \bigcup_{1 \le i \le n} (p_i)$.
For instance, the APEG p of Figure 1 represents all the following expressions:

$$(p) = \left\{ \begin{array}{c} \big((a+a)+b\big) \times c,\ \ \big((a+b)+a\big) \times c,\ \ \big((b+a)+a\big) \times c, \\ \big((2 \times a)+b\big) \times c,\ \ c \times \big((a+a)+b\big),\ \ c \times \big((a+b)+a\big), \\ c \times \big((b+a)+a\big),\ \ c \times \big((2 \times a)+b\big),\ \ (a+a) \times c + b \times c, \\ (2 \times a) \times c + b \times c,\ \ b \times c + (a+a) \times c,\ \ b \times c + (2 \times a) \times c \end{array} \right\} \quad (8)$$

In their article on EPEGs, R. Tate *et al.* use rewritting rules to extend the structure up to saturation [25, 24]. 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 2. An alternative technique for APEG construction is to use dedicated algorithms. Such algorithms, working in polynomial time, have been proposed in [15].

2.3 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

$$\text{Com} \quad \ni \quad c ::= id = e \mid c_1 \ ;\ c_2 \mid \mathsf{if}_{\Phi}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}_{\Phi}\ e\ \mathsf{do}\ c \mid \mathsf{nop}. \quad (9)$$

The transformation relies on several hypotheses. First of all, programs are assumed to be in static single assignment form (SSA form) [7]. 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 undestand this intermediary representation, let us consider the example of Figure 3. 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. 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 (9),

- δ is an environment $\delta : \mathcal{V} \to \text{Expr}$ 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 [12]. It records the program englobing 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.

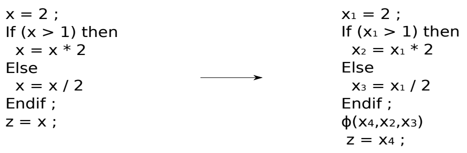
Let us consider first assignments. If (i) the variable v of some assignment $v = e$ does not exist in the domain of δ, if (ii) $v \notin \beta$ and if (iii) $v \neq \nu$ then we memorize e in δ and we remove the assignment from the program. Otherwise, if one of the conditions (i), (ii) or (iii) is not 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.

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.

$$\begin{array}{rl} & \langle \mathsf{x = a + b\ ;\ y = c + d\ ;\ z = x + y}\ ,\ \delta,\ [],\ \nu = z,\ [z]\rangle \\ \longrightarrow & \langle \mathsf{nop\ ;\ y = c + d\ ;\ z = x + y}, \delta' = \delta[\mathsf{x} \mapsto \mathsf{a + b}],\ [],\ \nu = z,\ [z]\rangle \\ \longrightarrow & \langle \mathsf{nop\ ;\ nop\ ;\ z = x + y}, \delta'' = \delta'[\mathsf{y} \mapsto \mathsf{c + d}],\ [],\ \nu = z,\ [z]\rangle \\ \longrightarrow & \langle \mathsf{nop\ ;\ nop\ ;\ z = ((d + c) + b) + a},\ \delta'',\ [],\ \nu = z,\ [z]\rangle \end{array} \tag{10}$$

In Equation (10), 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



```
x = 2 ;
If (x > 1) then
  x = x * 2
Else
  x = x / 2
Endif ;
z = x ;
```

```
x1 = 2 ;
If (x1 > 1) then
  x2 = x1 * 2
Else
  x3 = x1 / 2
Endif ;
ϕ(x4,x2,x3)
 z = x4 ;
```

Fig. 3. Original program (left) and its SSA Form (right).

of the conditional. When it is necessary, we re-inject variables that have been discarded from the main program. Let us take another example to explain how we transform conditionals.

$$x_1 = 0;\ \mathsf{if}_{\Phi(y_3,y_1,y_2)}\ x_1 > 1\ \mathsf{then}\ y_1 = x_1 + 2;\ \mathsf{else}\ y_2 = x_1 - 1; \nu = y_3. \quad (11)$$

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

$$\mathsf{if}_{\Phi(y_3,y_1,y_2)}\ x_1 > 1\ \mathsf{then}\ y_1 = x_1 + 2;\ \mathsf{else}\ y_2 = x_1 - 1; \nu = y_3. \quad (12)$$

This program is semantically incorrect since the test is undefined. So we re-inject the statement $x_1 = 0$ 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 optimized 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 $\beta \cup V$.

3 Case Studies

In this section, we consider four iterative programs performing numerical computations: Jacobi's Method, Newton-Raphson's Method, an Iterated Power Method used to compute the largest eigenvalue of a matrix and an iterative orthogonalization algorithm more stable than Gram-Schmidt Method. We demonstrate the efficiency of our techniques in accelerating the convergence of these algorithms by measuring the number of iterations before and after rewriting. We present speedups in terms of execution time and number of floating-point operations needed to achieve the computation.

We have implemented the original and optimized numerical iterative methods in the C programming language, compiled with GCC 4.2.1, and made them run on an Intel Core i5 with 4 Go memory in IEEE754 single precision in order to emphasize the effect of the finite precision. Programs are compiled with the default optimization level $-O2$. We have tried other levels of optimization without observing significant changes in our results.

3.1 Linear Systems of Equations

We start with a first case study concerning Jacobi's method [17] 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 $(\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(k)}, \mathbf{x}^{(k+1)}, \ldots)$ that converges towards the solution $\mathbf{x}^{(k)}$ of the system of linear equations.

To build the algorithm corresponding to this method, we decompose the initial matrix $\mathbf{A}$ into three matrices. The first one $\mathbf{D}$ contains the diagonal terms a_{ii} of the matrix. The second $\mathbf{U}$ contains the terms of the matrix which are above the main diagonal of $\mathbf{A}$ (a_{ij} with $j > i$) and the last one $\mathbf{L}$ contains the remaining terms of $\mathbf{A}$, i.e., the terms that are below the main diagonal (a_{ij} with $j < i$).

So, after transforming the matrix $\mathbf{A}$, we have the following equation to solve $\mathbf{Dx} = \mathbf{b} - (\mathbf{L} + \mathbf{U})\mathbf{x}$. To compute $\mathbf{x}^{(k+1)}$, we use:

$$x_i^{(k+1)} = \frac{b_i - \sum\limits_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)}}{a_{ii}} \qquad \text{where } \mathbf{x}^{(k)} \text{ is known.} \tag{13}$$

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

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}|. \tag{14}$$

Let us now examine how we can improve the convergence of Jacobi's method on the example given in Equation (15). This system is stable with respect to the sufficient condition of Equation (14) but it is close to be unstable in the sense that $\forall i, 1 \leq i \leq 4, |a_{ii}| \approx \sum\limits_{j=1, j\neq i}^{j=4} |a_{ij}|$.

$$\begin{pmatrix} 0.62 & 0.1 & 0.2 & -0.3 \\ 0.3 & 0.602 & -0.1 & 0.2 \\ 0.2 & -0.3 & 0.6006 & 0.1 \\ -0.1 & 0.2 & 0.3 & 0.601 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1.0/2.0 \\ 1.0/3.0 \\ 1.0/4.0 \\ 1.0/5.0 \end{pmatrix}. \tag{15}$$

We describe this system using the notations of Equation (13). To solve Equation (15) by Jacobi's method, we use the algorithm presented in Figure 4. This program is transformed with our tool by using the set of transformation rules described in Section 3. Note that, in the version of this program given to our

```
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;  }
```

Fig. 4. Listing of the initial 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 ;  }
```

Fig. 5. Listing of the optimized program of Jacobi's method.

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 longer 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 Figure 5. Note that this code is rather not intuitive and could very difficultly be written 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 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%.

In Figure 6, 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 optimized code. Obviously, the fact that the body of the loop is unfolded twice, in the optimized code is taken into account in the computation of the number of iterations needed to converge.

x_i	Initial Num of iteration	Iterations Num after optimization	Difference	Percentage
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

Fig. 6. Number of iterations of Jacobi's method before and after optimization to compute x_i, $1 \leq i \leq 4$.

3.2 Zero Finding

Newton-Raphson's Method [17] 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 (16). Note that the computation stops when $|u_{n-1} - u_n| < \epsilon$.

$$u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}. \tag{16}$$

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:

$$f(x) = x^5 - 10x^4 + 40x^3 - 80x^2 + 80x - 32, \tag{17}$$

$$f'(x) = 5x^4 - 40x^3 + 120x^2 - 160x + 80. \tag{18}$$

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 [18]. Consequently, this example presents some numerical difficulties for Newton-Raphson's method which converges slowly in this case.

The algorithm corresponding to Equation (16) is given in Figure 7. 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 Figure 8. The accuracy of the x_i's is improved up to 1.53% following the semantics of Section 2.

The results given in Figure 9 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 9. 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.

```
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) ;  } ;   }
```

Fig. 7. Listing of the initial 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) ;  } ; }
```

Fig. 8. Listing of the optimized Newton-Raphson's program.

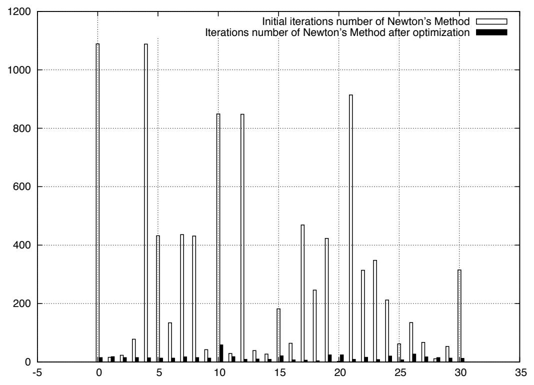


Fig. 9. Number of iterations of the Newton-Raphson's Method before and after optimization for initial values ranging from 0 to 3 (30 runs with a step of 0.1).

3.3 Eigenvalue Computation

The Iterated Power Method is a method used to compute the largest eigenvalue of a matrix and the related eigenvector [10]. We start by fixing 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 renormalizing $\mathbf{y}^{(1)}$ 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 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 dimention 4 with the eigenvector $(0.0\ \ 0.0\ \ 0.0\ \ 1.0)^T$ given on the Equation (15):

$$\mathbf{A} = \begin{pmatrix} \mathtt{d} & 0.01 & 0.01 & 0.01 \\ 0.01 & \mathtt{d} & 0.01 & 0.01 \\ 0.01 & 0.01 & \mathtt{d} & 0.01 \\ 0.01 & 0.01 & 0.01 & \mathtt{d} \end{pmatrix} \text{ with } \mathtt{d} \in [175.0, 200.0]. \tag{19}$$

```
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 ;}  }
```

Fig. 10. Listing of the Initial iterated power method.

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

$$\mathbf{A}\mathbf{x}^0 = \mathbf{y}^1, \quad \mathbf{A}^{y^1/y_4^1} = \mathbf{y}^2, \quad \mathbf{A}^{y^2/y_4^2} = \mathbf{y}^3, \quad \ldots \tag{20}$$

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

Our tool has improved the error bounds computed by the semantics of Section 2 of up to 25.76%. The optimized code is given in Figure 11.

```
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 ;}     }
```

Fig. 11. Listing of the optimized iterated power method.

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 12.

3.4 Iterative Gram-Schmidt Method

The Gram-Schmidt method is used to orthogonalize a set of non-zero vectors in a Euclidean or Hermitian space $\mathcal{R}^n$. 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 $1 \leq j \leq n$ [1,10,13]. The process followed by Gram-Schmidt method starts by defining the projection:

$$proj_{\mathbf{q'}}(\mathbf{q}) = \frac{\langle \mathbf{q}, \mathbf{q'} \rangle}{\langle \mathbf{q'}, \mathbf{q'} \rangle}\mathbf{q'}. \tag{21}$$

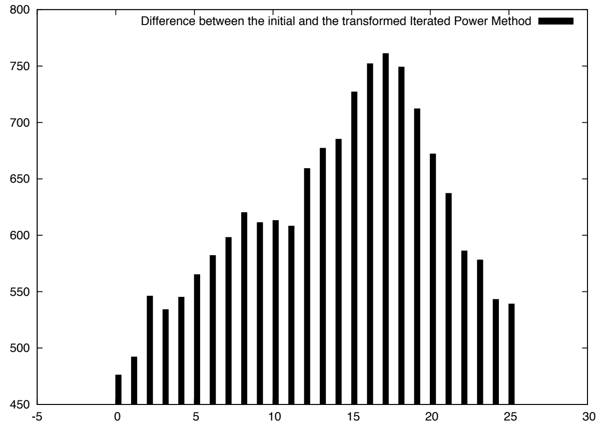


Fig. 12. Difference between numbers of iterations of initial and optimized Iterated Power Method (tests done for $\mathtt{d} \in [175, 200]$ with a step of 1).

In Equation (21), $\langle \mathbf{q}, \mathbf{q'} \rangle$ 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 results, as for example the algorithm of Figure 13 which repeats the orthogonalization step until the ratio $\frac{||\mathbf{q'}_j||_2}{||\mathbf{q}_j||_2}$ becomes large enough [13]. First, it starts by computing the orthogonal projection of $span(\{\mathbf{q_1}, \mathbf{q_2}, \mathbf{q_3}\})$. Then, it substracts this projection from the original vector and then normalizes the result to obtain $\mathbf{q}_3$, i.e., $span(\{\mathbf{q_1}, \mathbf{q_2}, \mathbf{q_3}\}) = span(\{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}\})$ 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 R^3 that we aim at orthogonalizing.

$$Q_n = \left\{ \mathbf{q}_1 = \begin{pmatrix} 1/7\mathtt{n} \\ 0 \\ 0 \end{pmatrix}, \mathbf{q}_2 = \begin{pmatrix} 0 \\ 1/25\mathtt{n} \\ 0 \end{pmatrix}, \mathbf{q}_3 = \begin{pmatrix} 1/2592 \\ 1/2601 \\ 1/2583 \end{pmatrix} \right\}. \quad (22)$$

```
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)  {
    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 ;   }
```

Fig. 13. 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 ; 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 - ((0.14285714285 *(((qj1 * qj3)) + (0.14285714285 * 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)) + (0.14285714285 * qj1))))) ;
  r1 = (r1 + ((TMP_6) + (0.14285714285 * 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 ;   }
```

Fig. 14. Listing of the optimized iterative Gram-Schmidt program.

For our experiments, we have chosen the values of `n` ranging from 1 to 10.

In Figure 14, 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 Figure 13, we show in Figure 15 that the transformed algorithm converges faster than the initial one by up to 14.5%.

3.5 Performance Analysis

We have shown in the former sections that we optimize 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 of a single iteration or by other side effects for example due to the compiler.

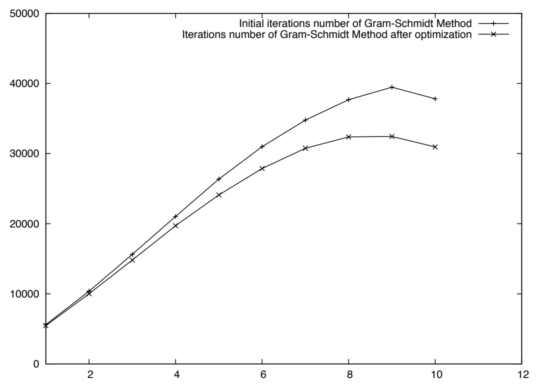


Fig. 15. Iterations number of initial and optimized iterative Gram-Schmidt Method for the family $(Q_n)_n$ of vectors, $1 \leq n \leq 10$.

We have chosen to observe the speedups of x_4 for Jacobi's method, and $x_0 = 3$ 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 Jacobi's method, we remark that we reduce its execution time by 74.5%. We give in Figure 16 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.

We have also counted the number of floating-point operations (flops) in the original and optimized codes. The numbers are given in Figure 17. 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. These results are coherent with the observed execution times.

	Original Code Execution Time in s	Optimized Code Execution Time in s	Percentage Improvement	Mean on n Runs
Jacobi	$1.49 \cdot 10^{-4}$	$0.38 \cdot 10^{-4}$	74.5%	10^4
Newton	$1.34 \cdot 10^{-3}$	$0.02 \cdot 10^{-3}$	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

Fig. 16. Execution time measurements of programs of Section 3.

Method	♯ of ± per it Original Code	♯ of ± per it Optimized Code	Total ♯ of ± Original Code	Total ♯ of ± opt Optimized Code	Percentage of Improvement
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	♯ of × per it Original Code	♯ of × per it Optimized Code	Total ♯ of × Original Code	Total ♯ of × opt Optimized Code	Percentage of Improvement
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

Fig. 17. Floating-point operations needed by programs of Section 3 to converge.

4 Conclusion

This article focuses on the impact of automatic transformation of programs in order to improve their numerical accuracy on the convergence time of numerical iterative algorithms. Our experiments show the usefulness of our approach on the time required by numerical iterative methods to converge. We have experimented several representative numerical iterative methods by giving them to our tool and we have shown that the transformed programs converge more quickly than the original ones without loss of accuracy. We have extended this study with complementary results concerning the execution time and the total number of floating-point operations.

What remains to be done is to have a more complete tool implementing other programming language patterns like functions and pointers. In addition, it would be interesting to extend the current work with a case study concerning a real size numerical application. The study described in [9] is a first step in this direction. Another future work would consist in studying 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 reproductability 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 reproductability.

References

1. N. Abdelmalek. Roundoff error analysis for gram-schmidt method and solution of linear least squares problem. *BIT*, 11:345–368, 1971.
2. ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*, std 754-2008 edition, 2008.
3. F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *PLDI '12, 2012*, pages 453–462. ACM, 2012.
4. C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.

5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points. In *POPL'77*, pages 238–252. ACM, 1977.
6. P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *POPL'02*, pages 178–190. ACM, 2002.
7. R. Cytron and R. Gershbein. Efficient accomodation of may-alias information in SSA form. In *PLDI'93*, pages 36–45. ACM, 1993.
8. N. Damouche, M. Martel, and A. Chapoutot. Intra-procedural optimization of the numerical accuracy of programs. In *FMICS'15*, volume 9128 of *LNCS*, pages 31–46. Springer, 2015.
9. N. Damouche, M. Martel, and A. Chapoutot. Optimizing the accuracy of a rocket trajectory simulation by program transformation. In *Computing Frontiers*, pages 40:1–40:2. ACM, 2015.
10. G. H. Golub and C. F. van Loan. *Matrix computations (3. ed.).* Johns Hopkins University Press, 1996.
11. E. Goubault and S. Putot. Static analysis of finite precision computations. In *VMCAI'11*, volume 6538 of *LNCS*. Springer, 2011.
12. E. Hankin. *Lambda Calculi A Guide For Computer Scientists.* Clarendon Press, Oxford, 1994.
13. V. Hernandez, J. E. Roman, A. Tomas, and V. Vidal. Orthogonalization routine in SLEPc Technical Report STR-1. In *Polytechnic University of Valencia.* STR1, 2007.
14. S. Hunt and D. Sands. Binding time analysis: A new perspective. In *PEPM'91*, pages 154–165, 1991.
15. A. Ioualalen and M. Martel. A new abstract domain for the representation of mathematically equivalent expressions. In *SAS'12*, volume 7460 of *LNCS*, pages 75–93. Springer, 2012.
16. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1993. ISBN 0-13-020249-5.
17. A. Kendall. *An Introduction to Numerical Analysis.* John Wiley & Sons, 1989.
18. Ph. Langlois and N. Louvet. How to ensure a faithful polynomial evaluation with the compensated horner algorithm. In *ARITH-18*, pages 141–149. IEEE Computer Society, 2007.
19. M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Higher-Order and Symbolic Computation*, 19(1):7–30, 2006.
20. M. Martel. Accurate evaluation of arithmetic expressions (invited talk). *Electr. Notes Theor. Comput. Sci.*, 287:3–16, 2012.
21. C. Mouilleron. *Efficient computation with structured matrices and arithmetic expressions.* PhD thesis, Université de Lyon-ENS de Lyon, November 2011.
22. G. Muller, E.-N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the sun commercial RPC protocol. In *PEPM'97*, pages 116–126. ACM, 1997.
23. J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic.* Birkhäuser Boston, 2010.
24. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *POPL'09*, pages 264–276. ACM, 2009.
25. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.