Numerically accurate code synthesis for Gauss pivoting method to solve linear systems coming from mechanics

Mikaël Barboteu, Nacera Djehaf, Matthieu Martel

▶ **To cite this version:**

Mikaël Barboteu, Nacera Djehaf, Matthieu Martel. Numerically accurate code synthesis for Gauss pivoting method to solve linear systems coming from mechanics. Computers & Mathematics with Applications, 2018, 10.1016/j.camwa.2018.08.021 . hal-02008076

HAL Id: hal-02008076
https://hal.science/hal-02008076

Submitted on 22 Oct 2021

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

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial 4.0 International License

Version of Record: https://www.sciencedirect.com/science/article/pii/S0898122118304401
Manuscript_42a48a06d70ab05cd8c7fa49849537ab

Numerically Accurate Code Synthesis for Gauss Pivoting Method to Solve Linear Systems Coming from Mechanics

Mikaël Barboteu, Nacera Djehaf and Matthieu Martel

barboteu@univ-perp.fr, nacera.djehaf@univ-perp.fr, matthieu.martel@univ-perp.fr

Laboratoire de Mathématiques et Physique
Université de Perpignan Via Domitia
52 Avenue Paul Alduy, 66860 Perpignan, France

Abstract

In numerical analysis of mechanical problems, we usually have to solve huge linear systems, which may be non-symmetric or ill-conditioned. For these reasons, it is necessary to develop original and domain specific approaches to treat these family of systems. In this work, we introduce a new methodology to synthesize numerically accurate programs for the Gauss pivoting method. The synthesis is based on program transformation techniques and it is guided in its estimation of accuracy by interval arithmetic that computes the propagation of roundoff errors. We apply our code synthesis to the resolution of systems coming from finite element method arising from problems of Mechanics. We test our synthesizer on two problems concerning the flexion of a beam and the sliding contact of a viscoelastic body on a rigid foundation. Our experimental results show that the specialized synthesized code to solve the families of systems given in input is far more accurate and faster than the standard implementation of the gauss method.

AMS Subject Classification : 65-04, 65K15, 65Y04, 68-04, 68Q25, 68Q42, 68Q55, 74-04, 74M15, 90C05

Key words : Numerical accuracy, Computer arithmetic, Program transformation, Code synthesis, Finite element method, Linear systems, Gauss pivoting algorithm, Ill-conditioning, Mechanical problems.

1. Introduction

Problems of Mechanics are usually expressed in terms of partial differential equations (PDEs) and most of the times, these PDEs cannot be solved with analytical methods due to the presence of non trivial complex constitutive laws as viscoelasticity, hyperelasticity, contact, friction... To determinate the solution of these problems, numerical methods are needed. In our context, the numerical method consist in approximating the systems of PDEs and then in solving the resulting approximated systems. In the framework of Mechanics, the main discretization methods used in the literature are the finite element method, the finite difference method and the finite volume method, which are selected according to the kind of problems considered [8]. In all these classes of methods, the discretization of the problems leads to the resolution of a system of linear equations. In this work, we considered linear systems coming from the finite element method. Furthermore,

© 2018 published by Elsevier. This manuscript is made available under the CC BY NC user license
https://creativecommons.org/licenses/by-nc/4.0/

the linear systems which we are interested in solving are represented by huge ill-conditioned sparse matrices which are sensitive to roundoff errors. There exists many well-known algorithms to solve these systems, based on direct methods such as Gauss pivoting method or on iterative methods such as the conjugated gradient method [16, 3]. However these methods are sensitive to the roundoff errors introduced by the floating-point arithmetic [2, 7] used by computers and which may partly or totally falsen the results of the computation. Indeed, the arithmetic of floating-point numbers strongly differs from the arithmetic of real numbers. For example, the usual rules on elementary operations like associativity, distributivity, etc. do not hold any longer and the numerical accuracy of the computation depends on how formulas are written. For these reasons, it is necessary to develop original and domain specific approaches to treat these family of systems.

Recently, several tools such as Herbie [15] and Salsa [6] have been proposed to automatically rewrite the mathematical formulas occurring in programs into mathematically equivalent formulas which evaluate more accurately in the computer arithmetic (in the sense that we obtain a result closer to the mathematical result that we would obtain if the computer used the real arithmetic). This work is motivated by the fact that the floating-point arithmetic is particularly not intuitive and that it is hard for the programmer to determine by hand how formulas should be written. In this work, we go a step further by introducing a new tool to synthesize automatically algorithms specialized for a family of systems. More precisely, we generate numerically accurate and time efficient programs for the Gauss pivoting method, given a family of systems described by interval matrices (matrices whose elements are intervals). The synthesis generates the code and uses Salsa to rewrite the computations in function of the ranges of the variables given by the intervals. As a result, we obtain automatically specialized solving methods, optimized for a family of systems.

To demonstrate the efficiency of our code synthesizer, we use it to generate programs for the resolution of systems coming from finite element method arising in two problems of Mechanics. The first problem consists of an academic but relevant mechanical problem which concerns the flexion of a one dimensional elastic beam fixed on its extremities. For the second example, we consider a non-trivial problem which describes the sliding contact of a two dimensional viscoelastic body against a moving foundation. For both problems we show that the code synthesized by our tool is far more accurate and faster than a standard code for Gauss pivoting method. More generally, this show that code synthesis is a credible and promising approach to efficiently solve numerically difficult problems, in domains like Mechanics.

The rest of the paper is structured as follows. In Section 2 we give the state of the art of the program transformation techniques and code synthesis. In Section 3, we describe the numerically accurate code synthesis for the Gauss pivoting method. Next, in Section 4 we present several numerical simulations to highlight the performance and the efficiency of the synthesized code compared to the classical Gauss pivoting method. Finally, in Section 5, we conclude and discuss about the future work in the continuation of the present article.

2. Program Transformation and Code Synthesis

In this section, we introduce background material needed to understand the rest of this article. Section 2.1 introduces the IEEE754 Standard for floating-point arithmetic. Section 2.2 presents the arithmetic used to compute safe bounds on the roundoff errors and an overview of our program transformation techniques is given in Section 2.3.

2.1. IEEE Standard 754 for Floating-Point Arithmetic

We introduce here some elements of floating-point arithmetic [2, 7]. First of all, a *floating-point number* x in base β is defined by

$$x = s \cdot (d_0.d_1 \dots d_{p-1}) \cdot \beta^e = s \cdot m \cdot \beta^{e-p+1} \tag{2.1}$$

where $s \in \{-1, 1\}$ is the sign, $m = d_0 d_1 \dots d_{p-1}$ is the *significant*, $0 \leq d_i < \beta$, $0 \leq i \leq p-1$, p is the *precision* and e is the exponent, $e_{min} \leq e \leq e_{max}$.

A floating-point number x is *normalized* whenever $d_0 \neq 0$. Normalization avoids multiple representations of the same number. The IEEE754 Standard also defines denormalized numbers which are floating-point numbers with $d_0 = d_1 = \ldots = d_k = 0$, $k < p-1$ and $e = e_{min}$. Denormalized numbers make underflow gradual [7]. The IEEE754 Standard defines binary formats (with $\beta = 2$) and decimal formats (with $\beta = 10$). In this article, without loss of generality, we only consider normalized numbers and we always assume that $\beta = 2$ (which is the most common case in practice). The IEEE754 Standard also specifies a few values for p, e_{min} and e_{max} which are summarized in Figure 1. Finally, special values also are defined: `nan` (Not a Number) resulting from an invalid operation, $\pm\infty$ corresponding to overflows, and $+0$ and -0 (signed zeros).

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

Figure 1: Basic binary IEEE754 formats.

The IEEE754 Standard also defines five rounding modes for elementary operations between floating-point numbers. These modes are towards $-\infty$, towards $+\infty$, towards zero, to the nearest ties to even and to the nearest ties to away and we write them $\uparrow_{-\infty}$, $\uparrow_{+\infty}$, $\uparrow_0$, $\uparrow_{\sim_e}$ and $\uparrow_{\sim_a}$, respectively. The elementary operations $\diamond \in \{+,\ -,\ \times,\ \div\}$ are then defined by

$$f_1\ \diamond_{\uparrow_\circ}\ f_2\ = \uparrow_\circ (f_1\ \diamond\ f_2) \tag{2.2}$$

where $\circ \in \{-\infty, +\infty, 0, \sim_e, \sim_a\}$ denotes the rounding mode. Equation (2.2) states that the result of a floating-point operation $\diamond_\circ$ done with the rounding mode $\circ$ returns what we would obtain by performing the exact operation $\diamond$ and next rounding the result using $\circ$. The IEEE754 Standard also specifies how the square root function must be rounded in a similar way to Equation (2.2) but does not specify the roundoff of other functions like sin, log, etc.

Because of the roundoff errors, the results of the computations are not exact. For example, the value $v = 2.7182818\ldots$ can be computed using Bernoulli's formula:

$$v = \lim_{n \to +\infty} u_n \quad \text{with} \quad u_n = \left(1 + \frac{1}{n}\right)^n,\ n \geq 0.$$

In double precision, $u_8 = 2.718282$ but then the accuracy decreases as n grows: $u_{14} = 2.716110$, $u_{16} = 3.035035$ and $u_{17} = 1.0$. The transformation techniques detailed in Section 2.3 aim at generating an expression which is mathematically equal to the original one and which minimizes the roundoff error on the result, i.e. the distance $|r - \uparrow_\circ (r)|$ between the exact result r and the floating-point result $\uparrow_\circ (r)$. To deal with the errors introduced by the floating-point arithmetic, we introduce the function $\downarrow_\circ : \mathbb{R} \to \mathbb{R}$ which computes the exact error due to rounding operation.

$$\downarrow_\circ (x) = x - \uparrow_\circ (x) \tag{2.3}$$

2.2. Error Bound Computation

In order to compute the errors during the evaluation of arithmetic expressions, we compute with values which are pairs $(f, \varepsilon) \in \mathbb{F} \times \mathbb{R} = \mathbb{E}$ where f denotes the floating point number used by the machine and ε denotes the exact error $\downarrow_\circ (f)$ attached to f, i.e., the exact difference between the real and floating-point numbers as defined in Equation (2.3). For example, the real number $\frac{1}{3}$

is represented by the value $w = (\uparrow_\sim \left(\frac{1}{3}\right), \downarrow_\sim \left(\frac{1}{3}\right)) = (0.333333, (\frac{1}{3} - 0.333333))$. The elementary operations on $\mathbb{E}$ is defined in [14].

In practice, we use an ensemblist version of this arithmetic based on intervals. A so-called abstract value [5] is a pair of intervals such that the first interval corresponds to the range of the floating-point values of the program and the second interval corresponds to the range of the errors obtained by subtracting the floating-point values from the exact ones. In $([f], [\varepsilon]) \in \mathbb{E}^\sharp$, we have $[f]$ the interval for the range of the values and $[\varepsilon]$ the interval of errors on the values $[f]$. The pair $([f], [\varepsilon])$ abstracts the set of concrete values $\{(f, \varepsilon) \ : \ f \in [f], \text{and } \varepsilon \in [\varepsilon]\}$ by intervals in a component-wise way.

We now introduce the arithmetic expressions on $\mathbb{E}^\sharp$. We approximate an interval $[x]$ with real bounds by an interval based on floating-point bounds, denoted by $\uparrow_\circ^\sharp ([x])$.

$$\uparrow_\circ^\sharp ([\underline{x}, \overline{x}]) = [\uparrow_\circ (\underline{x}), \uparrow_\circ (\overline{x})] \ . \tag{2.4}$$

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

$$\downarrow_\circ^\sharp ([\underline{x}, \overline{x}]) = [-y, y] \quad \text{with} \quad y = \frac{1}{2}\text{ulp}\big(\max(|\underline{x}|, |\overline{x}|)\big) \ . \tag{2.5}$$

Formally, the *unit in the last place*, denoted by $ulp(x)$, is the weight of the least significant digit of the floating-point number x. Equations (2.6) to (2.7) give the semantics of the addition and multiplication among $\mathbb{E}^\sharp$, for other operations see [14]. If we sum two numbers, we must add errors on the operands to the error produced by the roundoff of the result. When multiplying two numbers, the semantics is given by the development of $([f]_1 + [\varepsilon]_1) \times ([f]_2 + [\varepsilon]_2)$.

$$([f]_1, [\varepsilon]_1) + ([f]_2, [\varepsilon]_2) = \big(\uparrow_\circ^\sharp ([f]_1 + [f]_2), [\varepsilon]_1 + [\varepsilon]_2 + \downarrow_\circ^\sharp ([f]_1 + [f]_2)\big) \ , \tag{2.6}$$

$$([f]_1, [\varepsilon]_1) \times ([f]_2, [\varepsilon]_2) = \big(\uparrow_\circ^\sharp ([f]_1 \times [f]_2), [f]_2 \times [\varepsilon]_1 + [f]_1 \times [\varepsilon]_2 + [\varepsilon]_1 \times [\varepsilon]_2 + \downarrow_\circ^\sharp ([f]_1 \times [f]_2)\big) \ . \tag{2.7}$$

2.3. Program Transformation for Numerical Accuracy

In this section, we describe intuitively how the floating-point computations occurring in programs may be transformed in order to improve their numerical accuracy. Basically, we use a data structure called APEG for Abstract Program Expression Graph [10]. An APEG copes with the combinatory problem by representing in polynomial size an exponential number of mathematically equivalent expressions. An APEG is made of abstraction boxes, representing, for a given operator and set of operands, any parsing of the expression up to associativity and commutativity and of equivalence classes which consist of offering a choice of alternative operators to build an expression. For instance, the APEG p of Figure 2 represents all the following expressions:

$$\mathcal{A}(p) = \left\{ \begin{array}{c} \big((a+a)+b\big) \times c, \ \big((a+b)+a\big) \times c, \ \big((b+a)+a\big) \times c, \\ \big((2 \times a)+b\big) \times c, \ c \times \big((a+a)+b\big), \ c \times \big((a+b)+a\big), \\ c \times \big((b+a)+a\big), \ c \times \big((2 \times a)+b\big), \ (a+a) \times c + b \times c, \\ (2 \times a) \times c + b \times c, \ b \times c + (a+a) \times c, \ b \times c + (2 \times a) \times c \end{array} \right\} \tag{2.8}$$

To improve an expression, we first build its APEG $\mathcal{A}(p)$ and then we search in $\mathcal{A}$ the most accurate expression following the error computation model of Section 2.2.

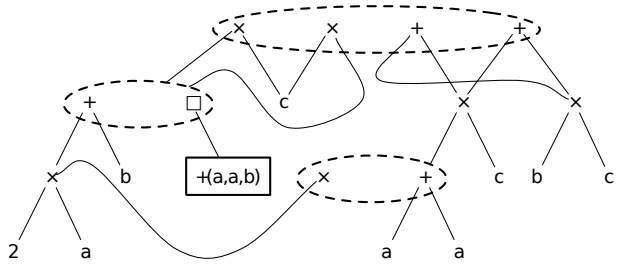

Figure 2: APEG for the expression $expr = \big((a + a) + b\big) \times c$.

For commands, i.e. assignments, conditionals, loops, functions, etc., we use a set of transformation rules allowing to mix the computations occurring in different instructions [6]. Basically, these rules build large expressions in order to offer more opportunities to rewrite them by associativity, commutativity, etc. For assignments, a first rule discards an assignment after saving it in the memory of the transformation tool and a second rule rewrites an assignment by inlining the memorized expressions in the current expression, in order to build a larger expression. When the obtained expressions become too large, we slice them at a defined level of the syntactic tree and we assign the sub-expressions to intermediary variables. For example, let us take the code of Figure 3 with three variables `x`, `y` and `z` and constants $a = 0.1$, $b = 0.01$, $c = 0.001$ and $d = 0.0001$. We aim at optimizing `z`.

Code	`x = a+c;`
	`y = b+d;`
	`z = x+y;`
		`y = b+d;`
		`z = x+y;`
			`z = x+y;`	`z =`
				`(a+c)+(b+d);`
					`z =`
					`a+(b+(c+d));`

Memory		x↦a+c	x↦a+c y↦b+d	x↦a+c y↦b+d	x↦a+c y↦b+d

Figure 3: Example of code transformation.

We remove the variable `x` and memorize it. So, the first assignment is discarded and memorized. We then repeat the same process for `y`. We may not remove `z` because it is the variable to be optimized. Then, we substitute `x` and `y` by their expressions and we transform the expression thanks to its APEG.

The second kind of rules deals with conditionals. If the condition is statically known, we execute the right branch, otherwise we rewrite both branches of the conditional. Other rules concerning the conditional consist of re-inserting variables that we have not to discard. For the while loop, one rule shows how to rewrite the body of the loop, and the other one is similar to the last one seen in conditionals. At last, we use some rules dealing with sequences of commands and functions.

2.4. Code synthesis

Code synthesis is the mechanized construction of a program. Synthesizing tools takes a specification of what the program should do, then it automatically generates an implementation that provably satisfies this specification. Obviously, the synthesized code has to be as efficient as possible. In our context efficiently means numerically accurate and fast.

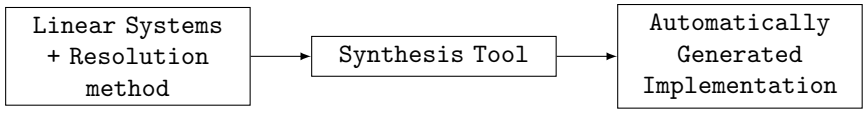

Figure 4: Code Synthesis Process.

In our case, as a specifications, we consider a particular family of linear systems coming from the finite element discretization of a Mechanical system of PDEs. The synthesizer tool should generate automatically a program for the numerical resolution method applied on the specified linear system. Figure 4 illustrates the process of program synthesis.

3. The `Rock-N-Roll` tool (`RNR`)

When we execute a numerical algorithm to solve a linear system of equations on a computer, each single operation introduces some roundoff errors, which are accumulated during the resolution process. Then, instead of the exact solution x of a linear system we get an approximate result. In order to solve this accuracy problem, we have developed a code synthesis tool that generates automatically a fast and accurate `C` program for a given family of linear systems. In this section, we present this tool, `Rock-N-Roll`. We detail its architecture, its inputs and outputs.

3.1. Architecture

In this section, we describe the main architecture of our tool as shown in Figure 5. `Rock-N-Roll` is written in `C` and made of several modules, described hereafter:

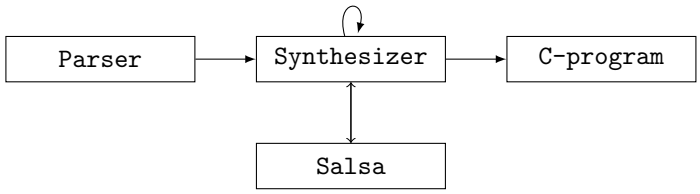

Figure 5: Software architecture of the `Rock-N-Roll` tool.

- `Parser:` Builds a family of systems [S] as follows: `RNR` takes as input a linear system `S:` **Ax=b** where $\mathbf{A} = (a_{ij})_{1 \leqslant i,j \leqslant n} \in \mathbb{R}$ and $\mathbf{b} = (b_i)_{1 \leqslant i \leqslant n} in \mathbb{R}$, then the interval matrix $[\mathbf{A}] = ({a_{ij}}^\sharp)_{1 \leqslant i,j \leqslant n}$ and the interval vector $[\mathbf{b}] = ({b_i}^\sharp)_{1 \leqslant i \leqslant n}$ are build for each i and j as follows:

$$\begin{cases} {a_{ij}}^\sharp = ([a_{ij} - a_{ij} \times k_1, a_{ij} + a_{ij} \times k_1], [-a_{ij} \times k_2, a_{ij} \times k_2]) \,, \\ {b_i}^\sharp = ([b_i - b_i \times k_1, b_i + b_i \times k_1], [-b_i \times k_2, b_i \times k_2]) \,. \end{cases}$$

  Where k_1 and k_2 are two real values given by the user (in our experiments Section 4, we gave $k_1 = 0.11$ and $k_2 = 0.00001$),

- `Salsa:` It is a tool that improves the numerical accuracy of programs by automatic transformation, it takes a program as input and returns more accurate one [6]. The optimization done by `Salsa` depends on the ranges (intervals) given as inputs for the variables of the code to be optimized,

- `Synthesizer:` This module implements the desired resolution method by generating the abstract syntax trees for the linear system resolution. Next, it generates specific code and gives it to the `Salsa` tool. When the `Salsa` transformation is achieved, the synthesizer replaces the previous code by the transformed one which is more accurate,

- `C-program: Rock-N-Roll` outputs a file containing a `C`-program corresponding to the efficient implementation of the resolution method specialized for a given family of linear systems with much better accuracy.

3.2. Inputs and Outputs of the Tool

In order to solve a linear system $S : \mathbf{Ax} = \mathbf{b}$ of size n using Gauss pivoting method, the tool takes as input two files that specify the ranges for both of $\mathbf{A} = (a_{ij})_{1 \leqslant i,j \leqslant n}$ and $\mathbf{b} = (b_i)_{1 \leqslant i \leqslant n}$, which are either introduced by the user or calculated by the `parser` module introduced in Section 3.1, using the error terms introduced by the user. Let $\mathbb{E}^\sharp$ the set introduced in 2.2. Instead of $S : \mathbf{Ax} = \mathbf{b}$, we use $[S] : [\mathbf{A}]\mathbf{x} = [\mathbf{b}]$ such that $\mathbf{A} \in [\mathbf{A}] = {a_{ij}}^\sharp)_{1 \leqslant i,j \leqslant n}$ and $\mathbf{b} \in [\mathbf{b}] = ({b_i}^\sharp)_{1 \leqslant i \leqslant n}$, with ${a_{ij}}^\sharp \in \mathbb{E}^\sharp$ and ${b_i}^\sharp \in \mathbb{E}^\sharp, 1 \leqslant i, j \leqslant n$. We have as input the following system:

$$[S] : \begin{pmatrix} {a_{11}}^\sharp & \cdots & {a_{1n}}^\sharp \\ \vdots & \ddots & \\ {a_{n1}}^\sharp & \cdots & {a_{nn}}^\sharp \end{pmatrix} x = \begin{pmatrix} {b_1}^\sharp \\ \vdots \\ {b_n}^\sharp \end{pmatrix}. \tag{3.1}$$

Since ${a_{ij}}^\sharp = ([f]_{ij}\ , [\varepsilon]_{ij})$ and ${b_i}^\sharp = ([f]_i\ , [\varepsilon]_i)$ for all $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant n$, we may rewrite Equation 3.1 as:

$$\begin{pmatrix} ([f]_{11}\ , [\varepsilon]_{11}) & \cdots & \cdots & ([f]_{1n}\ , [\varepsilon]_{1n}) \\ \vdots & \ddots & & \\ & & & \\ ([f]_{n1}\ , [\varepsilon]_{n1}) & \cdots & \cdots & ([f]_{nn}\ , [\varepsilon]_{nn}) \end{pmatrix} x = \begin{pmatrix} ([f]_1\ , [\varepsilon]_1) \\ \vdots \\ \vdots \\ ([f]_n\ , [\varepsilon]_n) \end{pmatrix} \tag{3.2}$$

Recall from Section 2.2 that in the pairs $([f], [\varepsilon])$, the first interval $[f]$ consists of the range of floating-point value and the second interval $[\varepsilon]$ consists of the error range associated to the floating-point interval $[f]$. Equation 3.2 describes a family of linear systems which, in our case correspond to a family of problems coming from a system of partial differential equations modeling a mechanical problem. The objective of the code synthesis is to produce automatically a program specialized in the resolution of this family of systems.

The output of our tool, `Rock-N-Roll`, is a `C`-program implementing Gauss pivoting method to compute an accurate solution x of $S \in [S]$. The next section is devoted to the presentation of the main features of the synthesized Gauss pivoting method.

3.3. Synthesized Gauss Pivoting Method

The resolution method synthesized by our tool `Rock-N-Roll` is Gauss pivoting method, which is one of the most widely used direct method to solve linear systems. In this section, we describe our algorithm for code synthesis. A first straightforward approach to write code for Gauss pivoting method consists in synthesizing a Gaussian elimination program for each element of $[S] : [\mathbf{A}]\mathbf{x} = [\mathbf{b}]$. Next, the synthesis of a back substitution computation program is considered to calculate the unknown x. Algorithm 1 below implements this approach.

Algorithm 1 Accurate Gauss pivoting algorithm

- **Input:**
  `[S]: [A]x=[b];`
- **Output:**
  `C`-program to compute the solution **x**;
- **Algorithm:**
  1. `GaussSynthesis([A],[b])`;
  2. `[S]`$\longleftarrow$`[S]': [A]'x=[b]'`;
  3. `BSubSynthesis([A]',[b]')`;
  4. C-program implementation;

The four steps of Algorithm 1 are detailed hereafter. We also illustrate on simple examples, the of each step of the algorithm.

- 1. `GaussSynthesis([A],[b])`: Produces a numerically optimized code for the Gaussian elimination rule for each abstract syntactic structure of ${a_{ij}}^\sharp$ and ${b_i}^\sharp$. In order to have more accurate results, the `GaussSynthesis` routine builds a specific code for the `Salsa` tool [6]. When the `Salsa` transformation is done, `Rock-N-Roll` replaces the old piece of code by the transformed one, which is more accurate.

Example: In this example, we give a piece of code for ${b_4}'$ computation, named $\mathtt{B}_4$ in our tool, which is the fourth component of the **b** of a linear 6-size system. The expression of b'_4 computation given by our synthesizer before `Salsa` is:

```
float B_4 =(((b4-(a4_1/a1_1)*b1)-((a4_2-(a4_1/a1_1)*a1_2)/(a2_2-(a2_1/a1_1)*a1_2))
*(b2-(a2_1/a1_1)*b1))-(((a4_3-(a4_1/a1_1)*a1_3)-((a4_2-(a4_1/a1_1)*a1_2)/(a2_2-(
a2_1/a1_1)*a1_2))*(a2_3-(a2_1/a1_1)*a1_3))/((a3_3-(a3_1/a1_1)*a1_3)-((a3_2-(a3_1
/a1_1)*a1_2)/(a2_2-(a2_1/a1_1)*a1_2))*(a2_3-(a2_1/a1_1)*a1_3)))*((b3-(a3_1/a1_1)
*b1)-((a3_2-(a3_1/a1_1)*a1_2)/(a2_2-(a2_1/a1_1)*a1_2))*(b2-(a2_1/a1_1)*b1))) ;
```

The expression of $\mathtt{B}_4$ computation after `Salsa` is given below. Note that this transformation depends on the given interval values of a_{ij} and b_i collected by the `parser` and that we would obtain a different expression for another family of systems.

```
float B_4 =((((b3-(b1*(a3_1/a1_1)))-((b2-(b1*(a2_1/a1_1)))*((a3_2-(a1_2*(a3_1/a1_1))
)/(a2_2-(a1_2*(a2_1/a1_1))))))*(((a4_3-(a1_3*(a4_1/a1_1)))-((a2_3-(a1_3*(a2_1/a1_1))
)*((a4_2-(a1_2*(a4_1/a1_1)))/(a2_2-(a1_2*(a2_1/a1_1))))))/-((a3_3-(a1_3*(a3_1/a1_1))
)-((a2_3-(a1_3*(a2_1/a1_1)))*((a3_2-(a1_2*(a3_1/a1_1)))/(a2_2-(a1_2*(a2_1/a1_1))))))
))+((b4-(b1*(a4_1/a1_1)))-((b2-(b1*(a2_1/a1_1)))*((a4_2-(a1_2*(a4_1/a1_1)))/(a2_2-(
a1_2*(a2_1/a1_1))))));
```

Note that `ai_j` and `bi` are the elements of our linear system.

- 2. At the end of the Gaussian elimination process, we obtain an equivalent upper triangular linear system named `[S]': [A]'x=[b]'`.

- 3. `BSubSynthesis([A]',[b]')`: Produces a numerically certified code for the back substitution resolution of the upper triangular system $[\mathbf{A}]'\mathrm{x}=[\mathbf{b}]'$. It construct the back substitution code on all the abstract syntactic structure of components of x and gives it to `Salsa`. At end of the `Salsa` transformation, the routine replaces the old piece of code by the accurate one given by `Salsa`.

Example: In this example, we give a piece of code for x_2 computation, which is the second component of the unknown x of a linear 6-size system. The expression of x_2 computation given by our synthesizer before `Salsa` is:

```
float x_2=(B2-((((A2_6*(B6/A6_6))+A2_5*(B5-(A5_6*(B6/A6_6)))/A5_5)+A2_4*(B4-((A4_6
*(B6/A6_6))+A4_5*(B5-(A5_6*(B6/A6_6)))/A5_5))/A4_4)+A2_3*(B3-(((A3_6*(B6/A6_6))+
A3_5*(B5-(A5_6*(B6/A6_6)))/A5_5)+A3_4*(B4-((A4_6*(B6/A6_6))+A4_5*(B5-(A5_6*(B6/
A6_6)))/A5_5))/A4_4))/A3_3))/A2_2;
```

After Salsa transformation, we have the expression below. Again this depends on the interval values of the variables.

```
float TMP_1 = A5_6;
float TMP_2 = (B6/A6_6);
float x_2=((B2-((((A2_6*(B6/A6_6))+(A2_5*((B5-(A5_6*(B6/A6_6)))/A5_5)))+(A2_4*((B4-
((A4_6*(B6/A6_6))+(A4_5*((B5-(A5_6*(B6/A6_6)))/A5_5))))/A4_4)))+(A2_3*((B3-(((A3_6
*(B6/A6_6))+(A3_5*((B5-(A5_6*(B6/A6_6)))/A5_5)))+(A3_4*((B4-((A4_6*(B6/A6_6))+(
A4_5*((B5-(TMP_1*TMP_2))/A5_5))))/A4_4))))/A3_3))))/A2_2);
```

Where `Ai_j` and `Bi` are the elements of the equivalent upper linear system obtained by the Gaussian elimination process.

- 4. `C-program` implementation: Finally, the tool extracts all the former information and builds a C-program whose execution will give a more accurate solution for any system belonging to the family of systems corresponding to the interval equation `[`**A**`]`**x**`=[`**b**`]`.

4. Numerical experimentations

The aim of this section is to present several numerical simulations which illustrate the performance and the efficiency of our tool, `Rock-N-Roll`, introduced in Section 3. Obviously, we aim at evaluating how much the numerical accuracy is improved but also the impact on the execution time. To do this, we have taken two examples based on two physical problems arising in Mechanics: The flexion of a beam fixed on its extremities and the compression of a viscoelastic body against a moving foundation. In both cases, the discretization is based on the finite element method (FEM) that was usually used to solve complicated problems in engineering, notably in elasticity and structural Mechanics modeling involving elliptic PDEs and complicated geometries. Note that the linear systems come from a Fortran computer code based on a MODULar Finite Element library (MODULEF)[1].

4.1. Flexion of a beam

The first example consists of an academic but relevant mechanical problem which concerns the flexion of an 1D elastic beam with Dirichlet boundary conditions on its extremities where the physical setting is depicted in Figure 6 (for homogeneous Dirichlet boundary conditions).

[1] `https://www.rocq.inria.fr/modulef/english.html`

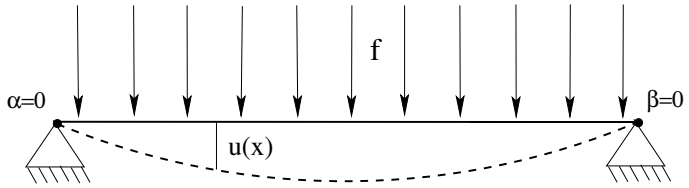

Figure 6: Physical setting of the flexion of a 1D beam.

To do that let us consider the following very simple 1D model problem which consist to find a displacement $u \in C^2([0,1],\mathbb{R})$ such that,

$$\begin{cases} -u''(x) = f \quad \forall x \in]0,1[\\ u(0) = \alpha \quad \text{and} \quad u(1) = \beta, \end{cases} \tag{4.1}$$

where f is a constant vertical force acting on the domain interval $\Omega = [0,1]$. In order to discretize the 1D elastic beam problem (4.1) and thus to obtain the related linear system, we use the finite element method. To do that, we have to introduce the mesh of the domain $\Omega = [0,1]$ by considering $N+1$ nodes $\{x_i,\ i = 1,..,N+1\}$ of the interval $[0,1]$ with $x_1 = 0$, $x_{N+1} = 1$ and $x_{i+1} = x_i + h_i$, for $i = 1,..,N$ where $h = max_{1\leq i\leq N}\{h_i\}$ is the mesh size. Therefore, the domain $[0,1]$ is discretized into N nonuniform intervals (x_i, x_{i+1}) that are the finite elements of size h_i. Then, we consider the simplest finite dimensional space that is to say the piecewise continuous linear function space defined over the mesh of the domain $\Omega = [0,1]$. Thus, after elementary calculus (see [11] and [12]) we finally obtain the following tridiagonal systems,

$$\begin{pmatrix} \frac{1}{h_1}+C & \frac{-1}{h_1} & & & \\ \frac{-1}{h_1} & \frac{1}{h_1}+\frac{1}{h_2} & \frac{-1}{h_2} & & \\ & \ddots & \ddots & \ddots & \\ & \frac{-1}{h_{N-1}} & \frac{1}{h_{N-1}}+\frac{1}{h_N} & \frac{-1}{h_N} & \\ & & \frac{1}{h_N} & \frac{-1}{h_N}+C \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \\ u_{N+1} \end{pmatrix} = \frac{f}{2} \begin{pmatrix} h_1 + C\alpha \\ h_1 + h_2 \\ \vdots \\ h_{N-1} + h_N \\ h_N + C\beta \end{pmatrix}$$

where C is a large penalization value in order to take into account the boundary conditions at $x = 0$ and $x = 1$.

In such type of problem, it is well known that the previous linear system is ill-conditioned and the condition number of the matrix is related to the $max_{1\leq i\leq N}\{\frac{1}{h_i}\}$. For this reason, it is an interesting example to test the Gauss Pivoting algorithm developed in Section 3. For our experiment, we considered that $f = -20N/m^2$, $max_{1\leq i\leq N}\{\frac{1}{h_i}\} = 10^6$, the penalization value $C = 10^6$ and that the beam is fixed on its extremities ($\alpha = \beta = 0$). First we created different linear systems of size $4 \leqslant N \leqslant 40$. Then, we calculated the solution of each system by the C-program given by our tool `Rock-N-Roll`: x_{RNR} and by a classical Gauss pivoting method program: x_{CG}. Finally, in order to highlight the differences between solutions, we computed and displayed in Figure 7 the relative error $RelErr(x) = \dfrac{\|\mathbf{A} * \mathbf{x} - \mathbf{b}\|_2}{\|\mathbf{b}\|_2}$ of each solution.

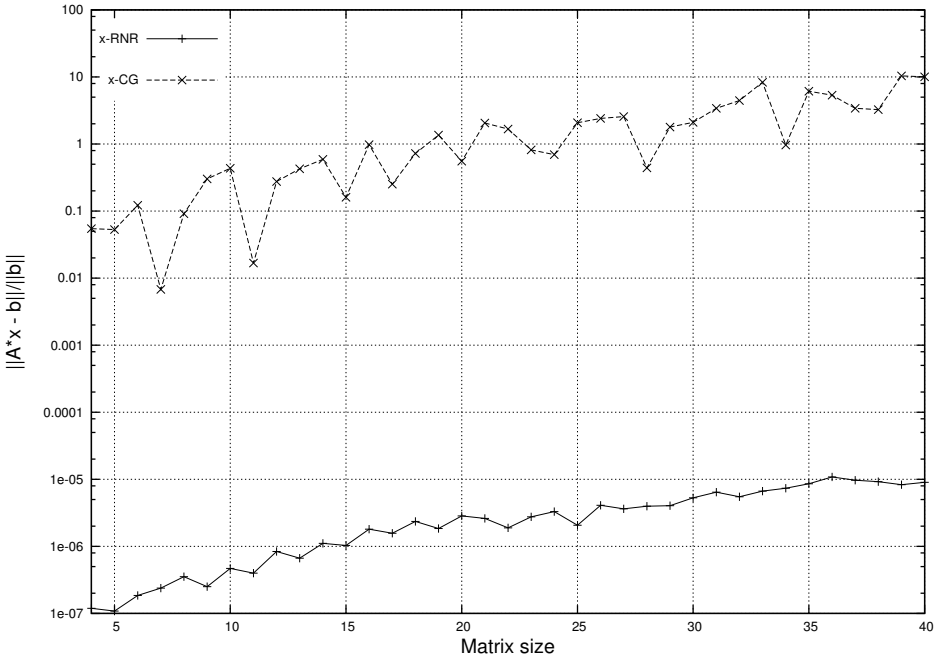

Figure 7: Behavior of the relative errors of the solutions

We can observe a significant difference between the curves corresponding to $RelErr(x_{RNR})$ and $RelErr(x_{CG})$, which are calculated with x_{RNR} and x_{CG} respectively. We see that the difference in accuracy between the results of the two methods is of the order of 5×10^{-6} on average. We can also see that the increase of the error is more regular and smoother with the solution calculated by the program generated by `Rock-N-Roll`.

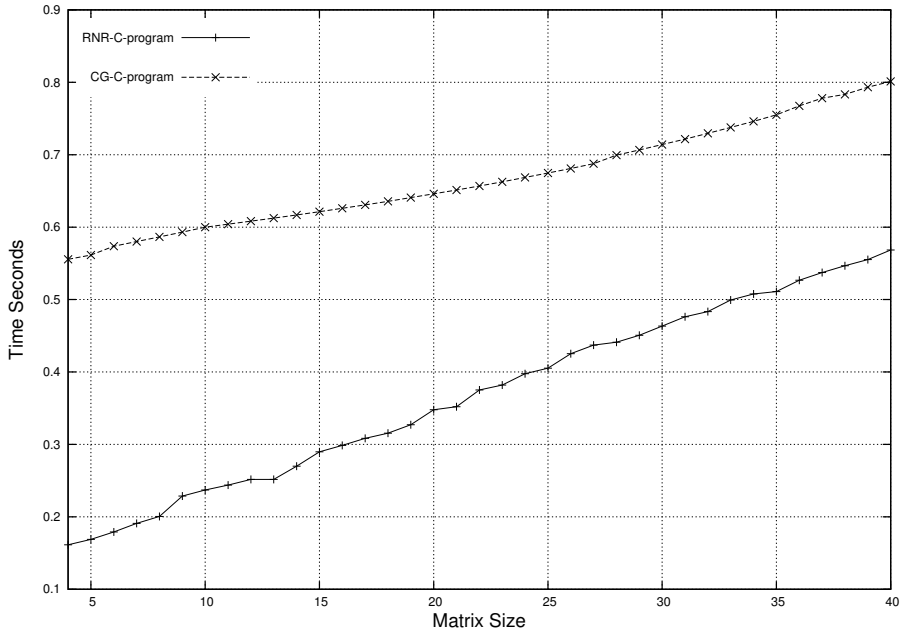

Figure 8: Execution time measurements of C-program generated by `RNR` and classical Gauss C-program

For execution time measurements, all the programs have been written in the C language and compiled with `GCC 4.9.2-03`, and executed on `Intel Core i7` in IEEE754 single precision in order to emphasize the effect of the finite precision. The results displayed in Figure 8 show that by synthesizing Gauss pivoting code, we improve not only its accuracy but we reduce its execution time too. This is mainly due to the fact that the systems from coming from our mechanical problem are sparse and that is the case, `Rock-N-Roll` is able to simplify the computation and to remove all the zero terms. For instance, in the examples of Section 3.3 some zero terms have been removed in the expression of x_2 and B_4 otherwise we would have a far more larger expression. The program synthesized by `Rock-N-Roll` contains far less computations then the original one. Remark that the computing time necessary for the obtention of the synthesizing Gauss pivoting code is not taken into account for the execution time measurements of the linear systems.

4.2. A frictional contact problem with a moving foundation

In this second example, we consider a non-trivial problem which describes the sliding contact of a 2D viscoelastic body against a moving foundation. Without going into details, we can say that the problem is discretized by combining the finite difference method and the finite element method for the time interval and the space domain, for more details about the discretization, we can refer to [13, 19]. Since frictional contact conditions are considered, the problem is non-linear and a Newton type method can be used to linearize it. Then, the resulting linearized problems are ill-conditioned and have to be solved by a robust and accurate numerical algorithm. For this reason, the linearized subproblems obtained at each Newton iterations are solved by the Gauss Pivoting algorithm developed in Section 3 and compared to the classical one. As for the first example, it is obvious that other methods of resolution (as preconditioned conjugated gradient for instance) can be used to solve such kind of systems. In this problem, the ill-conditioning comes from the frictional contact conditions that leads to large terms in the linearized systems related to the numerical treatment (augmented Lagrangian method and penalization method) of these non-smooth and non linear boundary conditions.

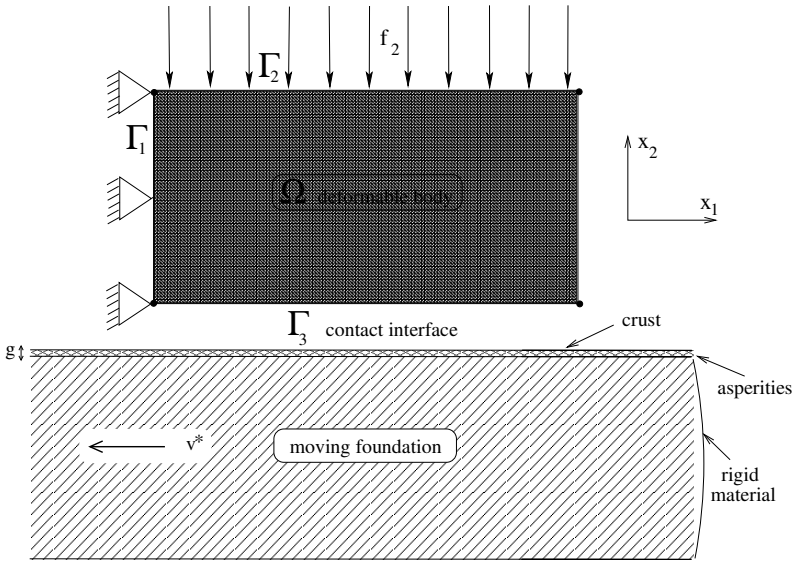

Figure 9: Physical setting of the sliding frictional contact problem.

The physical setting used for this problem is depicted in Figure 9. Here, we consider the frictional contact between a deformable body and a moving foundation. This specific foundation is composed by a rigid material covered by a thin crust and a deformable layer of asperities of depth g. Here

g represents the maximum value of the allowed penetration in the foundation. When this value of penetration is reached, the contact follows a unilateral condition without any additional penetration. Since the foundation is moving the friction condition is in a slip status within the Coulomb's form. The deformable body is a rectangle, $\Omega = (0,2) \times (0,1) \subset \mathbb{R}^2$, and its boundary Γ is split as follows: $\Gamma_1 = (\{0\} \times [0,1])$, $\Gamma_2 = ((0,2) \times \{1\}) \cup (\{2\} \times [0,1))$, $\Gamma_3 = (0,2] \times \{0\}$. The domain Ω represents the cross section of a three-dimensional linearly viscoelastic body subjected to the action of tractions in such a way that a plane stress hypothesis is assumed. On the part Γ_1 the body is clamped and, therefore, the displacement field vanishes there. Vertical compressions act on the part $(0,2) \times \{1\}$ of the boundary Γ_2 and the part $(\{2\} \times [0,1))$ is traction free. Constant vertical body forces are assumed to act on the viscoelastic body. The body is in frictional contact with an obstacle on the part Γ_3 of the boundary. For the numerical simulations, all the data concerning the problem can be found in [4].

In Figure 10, we present the two deformed configurations of the body with respect to two opposite velocities of the moving foundation.

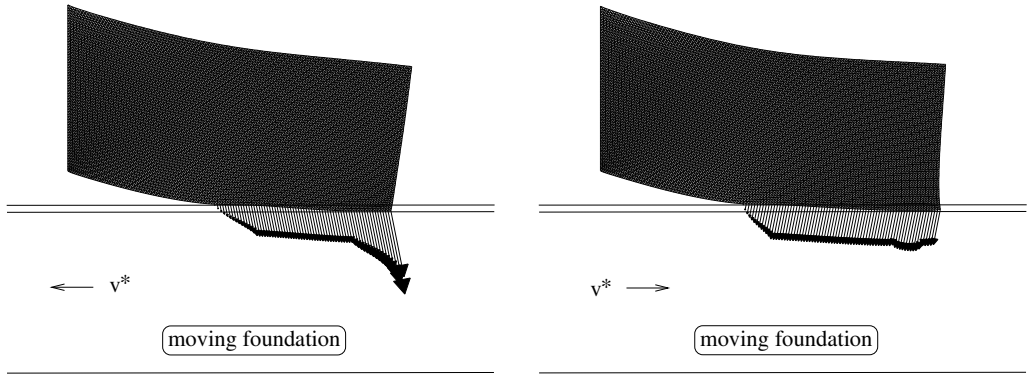

Figure 10: Deformed meshes with respect to two opposite velocities of the moving foundation.

For this second example, in order to illustrate the efficiency of the `Rock-N-Roll` tool we consider the first linearized system generated during the last iteration of the Newton solver. This linear system has the particularity to be non-symmetric due to the presence of friction terms, and ill-conditioned because of the augmented Lagrangian approach for the treatment of frictional contact conditions. (see [1, 4, 9, 13]).

matrix size	10	16	22	28
$\frac{\|\mathbf{A} * \mathbf{x_{RNR}} - \mathbf{b}\|_2}{\|\mathbf{b}\|_2}$	1.33974e-08	5.99737e-07	4.41512e-07	1.67206e-07
$\frac{\|\mathbf{A} * \mathbf{x_{CG}} - \mathbf{b}\|_2}{\|\mathbf{b}\|_2}$	8.02164e-06	9.12376e-05	5.62905e-05	3.55948e-05

Table 1: Behavior of the relative errors of the solutions.

matrix size	10	16	22	28
Synthesized Gauss pivoting (seconds)	0.252621	0.261653	0.266668	0.278529
Classical Gauss pivoting method (seconds)	0.328312	0.349035	0.363256	0.410162

Table 2: Execution time measurements of C-program generated by `Rock-N-Roll` and classical Gauss C-program.

In Table 1 and Table 2, the relative errors and execution times have been computed and displayed respectively, both for the classical Gauss pivoting method and for the C-program generated by our `Rock-N-Roll` tool with respect to 4 different sizes. As for the first example, a significant difference between the two methods is observed in favor of `Rock-N-Roll`. In Table 1, we see that the difference in accuracy is of order of 10^{-2}. The results displayed in Table 2 show that the C-program of Gauss pivoting method generated by `Rock-N-Roll` is faster. We can see a 30% increase (for the resolution time) for the classical Gaussian pivoting method whereas this increase is only 2% for the C-program of Gauss pivoting method implemented by our synthesizer `Rock-N-Roll`.

5. Conclusion

In this article, we have introduced a synthesized Gauss pivoting method implemented in `Rock-N-Roll`, an automatic synthesizer tool to improve the numerical accuracy of linear systems resolution, specifically systems coming from mechanical problems. We have detailed its architecture, and the different inputs and the outputs that it supports. We have tested `Rock-N-Roll` across experimental results obtained on two examples coming from two different mechanical problems with and without contact. The results obtained show the efficiency of our synthesized Gauss pivoting method which improves the numerical accuracy of computations compared to the classical Gauss pivoting method, so as the execution time. An interesting perspective consists of extending our work to synthesize Gauss pivoting method on partitioned matrices and also parallel. In this direction, we aim at solving the large matrices size problems. Furthermore, as prospect it would be interesting to add the conjugated gradient and the double conjugated gradient methods to our `Rock-N-Roll`. Taking into account non linear solvers as Newton type methods would be very challenging in the framework of numerical accuracy.

References

[1] P. Alart and A. Curnier, A mixed formulation for frictional contact problems prone to Newton like solution methods, *Comput. Meth. Appl. Mech., Engrg.* **92**, 353-375, 1991.

[2] ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*. ANSI/IEEE, std 754-2008 edition, 2008.

[3] Atkinson, Kendell A. An introduction to numerical analysis (2nd ed.). John Wiley and Sons. ISBN 0-471-50023-2, 1998.

[4] M. Barboteu & Y. Souleiman, Numerical Analysis of a Sliding frictional contact problem with Normal Compliance and Unilateral Contact, submitted to *Mathematical Methods in the Applied Sciences, Wiley.*

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[6] 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.

[7] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1), Mar, 1991.

[8] Grossmann, Christian, Roos, Hans-G., Stynes, Martin. Numerical Treatment of Partial Differential Equations. Springer. ISBN 978-3-540-71584-9, 2007.

[9] J. Haslinger and I. Hlavácek, *Numerical Methods for Unilateral Problems in Solid Mechanics*, in Handbook of Numerical Analysis, J.-L. Lions and P. Ciarlet, eds., Vol IV, North-Holland, Amsterdam, 313–485, 1996.

[10] 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.

[11] T. JR. Hughes, *The finite element method*, Prentice Hall, 1987.

[12] N. Kikuchi, *Finite element methods in Mechanics*, Cambridge, 1986.

[13] T. Laursen, *Computational Contact and Impact Mechanics*, Springer, Berlin, 2002.

[14] M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Higher-Order and Symbolic Computation*, 19(1):7–30, 2006.

[15] Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. In: PLDI. pp. 1–11. ACM, 2015.

[16] Saad, Yousef. Iterative methods for sparse linear systems (2nd ed.). Philadelphia, Pa.: Society for Industrial and Applied Mathematics. p. 195. ISBN 978-0-89871-534-7, 2003.

[17] M. Sofonea and A. Matei, *Mathematical Models in Contact Mechanics*, London Mathematical Society Lecture Note Series **398**, Cambridge University Press, Cambridge, 2012.

[18] M. Sofonea & Y. Souleiman, A Viscoelastic Sliding Contact Problem with Normal Compliance, Unilateral Constraint and Memory Term, *Mediterranean Journal of Mathematics*. **13**, 2863–2886, 2016.

[19] P. Wriggers, *Computational Contact Mechanics*, Wiley, Chichester, 2002.