Efficient algorithms for verified scientific computing: Numerical linear algebra using interval arithmetic

Hong Diep Nguyen

▶ **To cite this version:**

Hong Diep Nguyen. Efficient algorithms for verified scientific computing: Numerical linear algebra using interval arithmetic. Other [cs.OH]. Ecole normale supérieure de lyon - ENS LYON, 2011. English. NNT: 2011ENSL0617. tel-00680352

HAL Id: tel-00680352

https://theses.hal.science/tel-00680352

Submitted on 19 Mar 2012

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.

THÈSE

en vue de l'obtention du grade de

Docteur de l'École Normale Supérieure de Lyon - Université de Lyon

Spécialité : Informatique

Laboratoire de l'Informatique du Parallélisme

École Doctorale de Mathématiques et Informatique Fondamentale

présentée et soutenue publiquement le 18 janvier 2011

par M. *Hong Diep NGUYEN*

Efficient algorithms for verified scientific computing: numerical linear algebra using interval arithmetic

---

Directeur de thèse : M. Gilles VILLARD
Co-directrice de thèse : Mme Nathalie REVOL

Après l'avis de : M. Philippe LANGLOIS
M. Jean-Pierre MERLET

Devant la commission d'examen formée de :
M. Luc JAULIN, Président
M. Philippe LANGLOIS, Rapporteur
M. Jean-Pierre MERLET, Rapporteur
Mme Nathalie REVOL, Co-directrice
M. Siegfried RUMP, Membre
M. Gilles VILLARD, Directeur

Efficient algorithms for verified scientific computing: numerical linear algebra using interval arithmetic

Nguyễn Hồng Diệp

18 January 2011

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Description of the document . . . 3
- 1.3 Interval arithmetic . . . 4
  - 1.3.1 Operations . . . 4
  - 1.3.2 Algebraic properties . . . 7
  - 1.3.3 Absolute value properties . . . 8
  - 1.3.4 Midpoint and radius properties . . . 8
  - 1.3.5 Variable dependency . . . 11
  - 1.3.6 Wrapping effect . . . 11
  - 1.3.7 Implementation . . . 12
  - 1.3.8 Implementation environment . . . 15
- 1.4 Conclusion . . . 15

2 Interval matrix multiplication **17**
- 2.1 Introduction . . . 17
- 2.2 State of the art . . . 17
- 2.3 Algorithm 1 using endpoints representation . . . 21
  - 2.3.1 Special cases . . . 21
  - 2.3.2 General case . . . 23
  - 2.3.3 Implementation . . . 29
- 2.4 Algorithm 2 using midpoint-radius representation . . . 34
  - 2.4.1 Scalar interval product . . . 34
  - 2.4.2 Matrix and vector operations . . . 38
  - 2.4.3 Implementation . . . 39
- 2.5 Conclusion . . . 42

3 Verification of the solution of a linear system **45**
- 3.1 Introduction . . . 45
- 3.2 State of the art, `verifylss` function . . . 46
- 3.3 Floating-point iterative refinement . . . 48
- 3.4 Interval iterative refinement . . . 49
  - 3.4.1 Numerical behavior . . . 51
  - 3.4.2 Updating the computed solution . . . 52
  - 3.4.3 Initial solution . . . 52
  - 3.4.4 Interval iterative improvement . . . 54
  - 3.4.5 Detailed algorithm . . . 55

3.5 Numerical experiments . . . 57
3.5.1 Implementation issues . . . 57
3.5.2 Experimental results . . . 59
3.6 Conclusion . . . 61

4 Relaxation method **63**
4.1 Introduction . . . 63
4.2 Interval improvement: Jacobi and Gauss-Seidel iterations . . . 64
4.2.1 Convergence rate . . . 66
4.2.2 Accuracy . . . 67
4.2.3 Initial solution of the relaxed system . . . 70
4.3 Implementation and numerical experiments . . . 71
4.3.1 Implementation . . . 71
4.3.2 Numerical results . . . 71
4.4 Conclusion . . . 73

5 Effects of the computing precision **75**
5.1 Introduction . . . 75
5.2 Implementation using variable computing precisions . . . 75
5.2.1 Residual computation . . . 76
5.2.2 Floating-point working precision . . . 76
5.2.3 Needed working precision . . . 77
5.3 Error analysis . . . 78
5.3.1 Normwise relative error . . . 82
5.3.2 Componentwise relative error . . . 83
5.4 Doubling the working precision for approximate solution . . . 84
5.4.1 Implementation issues . . . 84
5.4.2 Numerical results . . . 85
5.4.3 Other extended precisions . . . 85
5.5 Conclusion . . . 87

Conclusions and Perspectives **89**

List of Figures

1.1 Wrapping effect. 12

2.1 Performance of `igemm`. 32
2.2 Performance of igemm3. 42

3.1 Solution set of the interval linear system (3.3). 51
3.2 MatLab implementation results for 1000×1000 matrices. 60

4.1 Solution set of the relaxed system. 65
4.2 Relaxation technique: MatLab implementation results for 1000×1000 matrices. 73

5.1 Effects of the residual precision. 77
5.2 Effects of the working precision. 78
5.3 Minimal working precision needed. 78
5.4 Effect of doubling the working precision for the approximate solution. 87
5.5 Effect of other extended precisions. 88

List of Algorithms

2.1 Decomposition algorithm for endpoint intervals 29
3.1 Classical iterative refinement 49
3.2 Interval iterative refinement 50
3.3 Testing the H-matrix property 54
3.4 Solving and verifying a linear system 56
4.1 Interval improvement function `refine` using relaxed Jacobi iterations . . . 72
5.1 Solving and verifying a linear system 76
5.2 Doubling the precision for the approximate solution 86

Chapter 1

Introduction

This chapter presents the general context and the motivation of my Ph.D. preparation. It gives a brief introduction to interval arithmetic together with its advantages as well as a discussion on the main obstacles to its application. Interval arithmetic always provides reliable results but suffers from low performance with respect to floating-point arithmetic. Moreover, a naive application of interval arithmetic does not provide a sharp enclosure of the exact result, or can even fail to provide guaranteed results. In this chapter, we present some properties of interval arithmetic, which will serve as the basis to derive efficient implementations of interval algorithms in the following chapters.

1.1 Motivation

Floating-point arithmetic is used for scientific computing purpose and it allows to obtain accurate results at a very fast speed. In many cases, floating-point arithmetic provides results of a quality which is satisfactory for user demand. However, with its well-known effect of rounding errors, computed results are rarely exact. Each number being encoded by a finite number of digits, the set of floating-point numbers is indeed discrete and thus cannot represent all real values. Meanwhile the result of a mathematical operation between real numbers or floating-point numbers is a real value and in many cases is not representable by another floating-point number. Thus, to be able to be encoded by another floating-point number, the computed result takes as value a floating-point number which is close to the exact result. More precisely, the computed result is rounded to a nearby floating-point value. The difference between the floating-point value and the exact value is called the rounding error.

These computed results are indeed approximate results. To reduce the effect of rounding errors, one can increase the computing precision. The higher the computing precision is, the smaller the rounding errors are. However, increasing the computing precision means increasing the storage space as well as increasing significantly the computing time. Therefore, fixed finite precision is chosen to gain performance. The most common formats are IEEE single and double precisions. Using the IEEE double precision, which is encoded by 64 bits with 1 sign bit, 11 exponent bits, and 52 mantissa bits, the effect of rounding errors in many applications is negligible. Nevertheless, in some cases, it is needed to assess how accurate the computed results are.

There exists a number of techniques for assessing the effect of rounding errors. Firstly,

an a priori error bound upon the computed result can be obtained by performing numerical analysis. This error bound is based on the theory of perturbations [Hig02], and is usually estimated using the condition number which measures the stability of results with respect to small changes in inputs. Nevertheless, when the problem is ill-conditioned, these condition-based error bounds are usually pessimistic and can give only an overview of how accurate the computed results could be expected, regardless of which computation model is employed. In some cases, a rigorous assessment of result accuracy is needed.

Another option is to use running error analysis, which yields an a posteriori error bound simultaneously with the computed approximate result [Hig02, Wil60, Lan04]. Running error computation is based on the rounding error model: $|x \circ y - \mathtt{fl}(x \circ y)| \leq \epsilon |\mathtt{fl}(x \circ y)|$, where ϵ is the machine relative error. Running error computation uses the actual computed intermediate quantities, therefore a running error bound is usually smaller than an a priori error bound. However, running error computation depends on every specific problem. Each individual problem needs to have its own model of computing running errors. This is disadvantageous when the problem in question is of high complexity.

Interval arithmetic is designed to automate this process of computing reliable results, as stated in the dissertation of R. E. Moore entitled "*Interval Arithmetic and Automatic Error Analysis in Digital Computing*" [Moo63] which is the most popular among the pioneers to introduce interval arithmetic in the early 60s. Interval arithmetic does not aim directly at improving the accuracy of computations, but at providing a certificate of accuracy. The fundamental principle of interval arithmetic is to enclose all quantities which are not known exactly during computations. Using intervals to represent data, and interval arithmetic to perform computations, one obtains not an approximation but a range of values in which lies the exact result. With this enclosure, one gets a certificate of accuracy, i.e. a bound on the distance between the exact result and the computed result. This computed result is said to be verified in the sense that a value which is not included in the computed range cannot be the exact result.

Interval arithmetic always provides verified results. Nevertheless, if the computed enclosure is too large, then the provided certificate is of no use. Therefore, accuracy is of great concern when developing numerical methods that yield reliable results using interval arithmetic.

Interval arithmetic allows to obtain certification for computed results. Meanwhile, arbitrary precision allows to increase the accuracy of computations by reducing the impact of rounding errors. The combination of these two components provides accurate results accompanied by a sharp error bound.

On the one hand, having to compute an enclosure of all possible values, interval arithmetic suffers from low performance. The ratio of execution time of interval arithmetic with respect to floating-point arithmetic is 4 in theory, as this can be seen on the formula for interval multiplication. In practice, this factor is much higher if care is not taken when implementing it. Moreover, using arbitrary precision also introduces an important overhead in execution time. Although interval arithmetic is able to obtain reliable results endowed with a certificate of accuracy, if the computation time is too long then it is not of much use.

On the other hand, for many problems which have been solved efficiently in floating-point arithmetic, when porting to interval arithmetic, simply replacing floating-point operations by their interval counterparts does not lead to an efficient algorithm and mostly does not provide results of high precision, or it can even fail to provide guaranteed results.

Therefore, I proceeded the Ph.D. preparation with the aim of obtaining an efficient implementation of interval arithmetic for some numerical problems. "Efficient" means computing enclosures of small relative width at a cost of reasonable execution time, which is of a constant modest factor with respect to computing approximate results using floating-point arithmetic. Additionally, increasing the accuracy is likely to increase the computation time. Hence, in some cases, a trade-off between accuracy and speed is mandatory.

As a first step toward providing a general methodology to obtain an efficient implementation of interval arithmetic, the 3 years of my PhD preparation succeeded to provide some techniques to improve the efficiency of interval arithmetic for several problems in numerical linear algebra, as well as some new tradeoffs between accuracy and speed.

1.2 Description of the document

In the context of this Ph.D., we propose several accurate algorithms and efficient implementations in verified linear algebra using interval arithmetic. Two fundamental problems are addressed, namely the multiplication of interval matrices and the verification of floating-point solutions of linear systems.

For the multiplication of interval matrices, we propose two algorithms which are based respectively on endpoints and midpoint-radius representations of intervals [NR10, Ngu]. These algorithms offer new trade-offs between speed and accuracy. Indeed, they are less accurate but much faster than the natural algorithm, the speedup is of order 10 for matrices of dimensions 100×100 and might attain order 100 for matrices of dimensions 1000×1000. On the other hand, they are slower than the fastest algorithm (proposed by S. M. Rump), and the factor of execution time is around 2, but they provide sharper bounds of the exact results with an overestimation factor less than 1.18, compared to 1.5 for the fastest algorithm.

To verify the floating-point solution of a linear system, we adapt floating-point iterative refinements to interval arithmetic [NR]. By using alternately floating-point and interval refinements, our method provides an accurate solution together with a tight error bound upon it. To reduce the slowdown due to the intensive use of interval operations, we introduce a relaxation technique, which helps to reduce drastically the algorithm's execution time. Additionally, the use of extended precision for the approximate solution yields solutions which are guaranteed to almost the last bit at a cost of an acceptable overhead of execution time.

This document is organized as follows: the rest of this first chapter briefly introduces interval arithmetic together with some of its properties. A more thorough introduction to interval arithmetic and its applications can be found in [Moo63, MB79, MKC09, JKDW01, Neu90, HKKK08]. The second chapter presents algorithms for implementing the multiplication of interval matrices. The third chapter is devoted to the underlying approach to verify a linear system of equations. Chapter 4 presents the relaxation technique to improve the performance of the algorithm presented in chapter 3. Chapter 5 studies the effect of the computing precisions on the accuracy of computed results.

1.3 Interval arithmetic

Denote by $\mathbb{R}$ the set of real numbers. A real interval is a closed connected subset of $\mathbb{R}$, which is of the form

$$\mathbf{a} = [\underline{a}, \overline{a}] \quad \underline{a}, \overline{a} \in \mathbb{R} \cup \{-\infty, +\infty\}$$

where $\underline{a} \leq \overline{a}$. In particular, if $\underline{a} = \overline{a}$ then $\mathbf{a}$ is said to be a thin interval, or a degenerate interval. If $\underline{a} = -\overline{a}$ then $\mathbf{a}$ is centered in zero.

The set of real intervals is denoted by $\mathbb{IR}$. Interval arithmetic defines operations over $\mathbb{IR}$.

Notation. *In this document, boldface letters denote intervals, lower case letters denote scalars and vectors, upper case letters denote matrices.*

1.3.1 Operations

An interval algebraic operation is defined by taking all the possible results of the corresponding operation, applied to all the possible pairs of real values taken from the input intervals. Interval operations must always obey the fundamental property of *isotonicity*, regardless of representation or implementation:

$$\forall a \in \mathbf{a} \quad \forall b \in \mathbf{b} \ldots \qquad \mathtt{f}(a, b, \ldots) \in \mathtt{f}(\mathbf{a}, \mathbf{b}, \ldots).$$

Convex hull

The convex hull of a set of real values is the smallest interval which includes all these real values. Let S be a set of real values then the convex hull of S, denoted by $\square$, is defined by

$$\square S = [\mathtt{min}(S), \mathtt{max}(S)] .$$

Additionally, let $\mathbf{a} = [\underline{a}, \overline{a}] \in \mathbb{IR}$, $\mathbf{b} = \left[\underline{b}, \overline{b}\right] \in \mathbb{IR}$ be two intervals, then the hull of these two intervals is defined as

$$\square(\mathbf{a}, \mathbf{b}) = \left[\mathtt{min}(\underline{a}, \underline{b}), \mathtt{max}(\overline{a}, \overline{b})\right] .$$

Unary operations

For $\mathbf{a} = [\underline{a}, \overline{a}] \in \mathbb{IR}$, a unary operation $\mathtt{f}$ on $\mathbf{a}$ is defined as

$$\mathtt{f}(\mathbf{a}) = \square\{\mathtt{f}(a) \quad a \in \mathbf{a}\}.$$

Table 1.1 lists some unary interval operations: the formula are established using the monotonicity of the operations when possible.

It is easy to prove the isotonicity of these operations. Let us take for example the case of the square operation. For all $a \in \mathbf{a}$, then $\underline{a} \leq a \leq \overline{a}$:

- if $\underline{a} \geq 0$ then $0 \leq \underline{a} \leq a \leq \overline{a} \Rightarrow 0 \leq \underline{a}^2 \leq a^2 \leq \overline{a}^2$, hence $a^2 \in [\underline{a}^2, \overline{a}^2]$;
- if $\overline{a} \leq 0$ then $\underline{a} \leq a \leq \overline{a} \leq 0 \Rightarrow 0 \leq \overline{a}^2 \leq a^2 \leq \underline{a}^2$, hence $a^2 \in [\overline{a}^2, \underline{a}^2]$;
- if $\underline{a} < 0 < \overline{a}$. Because $\underline{a} \leq a \leq \overline{a}$ then $|a| \leq \mathtt{max}(|\underline{a}|, |\overline{a}|) \Rightarrow a^2 \leq \mathtt{max}(|\underline{a}^2|, |\overline{a}^2|)$. It is always true that $a^2 \geq 0$. Hence $a^2 \in [0, \mathtt{max}(\overline{a}^2, \underline{a}^2)]$. Furthermore, as 0, $\underline{a}$ and $\overline{a}$ belong to $\mathbf{a}$, 0, $\underline{a}^2$ and $\overline{a}^2$ belong to $\mathbf{a}^2$, hence $[0, \mathtt{max}(\overline{a}^2, \underline{a}^2)]$ is the tightest interval.

Negation	$-\mathbf{a} = [-\overline{a}, -\underline{a}]$	
Square	$\mathbf{a}^2 = [\underline{a}^2, \overline{a}^2]$	if $\underline{a} \geq 0$
	$\mathbf{a}^2 = [\overline{a}^2, \underline{a}^2]$	if $\overline{a} \leq 0$
	$\mathbf{a}^2 = [0, \mathtt{max}(\overline{a}^2, \underline{a}^2)]$	if $\underline{a} < 0 < \overline{a}$
Inverse	$1/\mathbf{a} = [1/\overline{a}, 1/\underline{a}]$	if $\underline{a} > 0$ or $\overline{a} < 0$
	$1/\mathbf{a} = [-\infty, \infty]$	if $\underline{a} < 0 < \overline{a}$
	$1/\mathbf{a} = [1/\overline{a}, \infty]$	if $\underline{a} = 0$
	$1/\mathbf{a} = [-\infty, 1/\underline{a}]$	if $\overline{a} = 0$

Table 1.1: Unary interval operations.

Binary operations

Let $\mathbf{a} = [\underline{a}, \overline{a}] \in \mathbb{IR}, \mathbf{b} = [\underline{b}, \overline{b}] \in \mathbb{IR}$ be two intervals. A binary operation $\circ \in \{+ - * /\}$ between $\mathbf{a}$ and $\mathbf{b}$ is defined as

$$\mathbf{a} \circ \mathbf{b} = \square\{a \circ b \quad a \in \mathbf{a}, b \in \mathbf{b}\}.$$

Addition

Due to its monotonicity, the addition of two intervals can be computed by adding their corresponding endpoints:

$$\mathbf{a} + \mathbf{b} = \left[\underline{a} + \underline{b}, \overline{a} + \overline{b}\right].$$

The isotonicity of the addition is directly deduced from its monotonicity.

Subtraction

Similarly to the addition, the subtraction between two intervals is computed by

$$\mathbf{a} - \mathbf{b} = \left[\underline{a} - \overline{b}, \overline{a} - \underline{b}\right].$$

Multiplication

The multiplication of two intervals is formed by taking the minimum and maximum values of the four products between their two pairs of endpoints

$$\mathbf{a} * \mathbf{b} = \left[\mathtt{min}(\underline{a} * \underline{b}, \underline{a} * \overline{b}, \overline{a} * \underline{b}, \overline{a} * \overline{b}), \mathtt{max}(\underline{a} * \underline{b}, \underline{a} * \overline{b}, \overline{a} * \underline{b}, \overline{a} * \overline{b})\right].$$

Therefore, an interval multiplication requires four punctual products and six comparisons. Otherwise, one can reduce the number of punctual products and comparisons by inspecting the sign of input intervals. For example, if $\mathbf{a} \geq 0$ then

$$\mathbf{a} * \mathbf{b} = \left[\mathtt{min}(\underline{a} * \underline{b}, \overline{a} * \underline{b}), \mathtt{max}(\underline{a} * \overline{b}, \overline{a} * \overline{b})\right].$$

We can then inspect the sign of endpoints of $\mathbf{b}$:

- if $\mathbf{b} \geq 0$ then $\mathbf{a} * \mathbf{b} = \left[\underline{a} * \underline{b}, \overline{a} * \overline{b}\right]$.
- if $\mathbf{b} \leq 0$ then $\mathbf{a} * \mathbf{b} = \left[\overline{a} * \underline{b}, \underline{a} * \overline{b}\right]$.
- if $\underline{b} < 0 < \overline{b}$ then $\mathbf{a} * \mathbf{b} = \left[\overline{a} * \underline{b}, \overline{a} * \overline{b}\right]$.

The worst case is when $\underline{a} < 0 < \overline{a}$ and $\underline{b} < 0 < \overline{b}$, in this case we say that $\mathbf{a}$ and $\mathbf{b}$ contain 0 in their interior. Then

$$\mathbf{a} * \mathbf{b} = \left[\mathtt{min}(\underline{a} * \overline{b}, \overline{a} * \underline{b}), \mathtt{max}(\underline{a} * \underline{b}, \overline{a} * \overline{b})\right],$$

we still have to compute all the four punctual products, except that only two comparisons are needed instead of six. This explains that the cost of interval arithmetic is 4 times the cost of real arithmetic.

Note that, different from real number operations, an interval square is not equivalent to a multiplication of an interval by itself. For $\mathbf{a} \in \mathbb{IR}$, it is always true that $\mathbf{a}^2 \subseteq \mathbf{a} * \mathbf{a}$, and equality holds when $\mathbf{a}$ does not contain 0 in its interior. For example, if $\mathbf{a} = [1, 2]$ then $\mathbf{a}^2 = [1, 4]$, and $\mathbf{a} * \mathbf{a} = [1, 4]$. Hence, $\mathbf{a}^2 = \mathbf{a} * \mathbf{a}$. Meanwhile, if $\mathbf{a} = [-1, 2]$ then by definition $\mathbf{a}^2 = [0, 4]$ whereas $\mathbf{a} * \mathbf{a} = [-2, 4]$.

Division

Interval division is defined via multiplication and inverse, see Table 1.1. Let $\mathbf{a} \in \mathbb{IR}, \mathbf{b} \in \mathbb{IR}$ be two intervals then

$$\mathbf{a}/\mathbf{b} = \mathbf{a} * (1/\mathbf{b}).$$

Set operations

Intervals are connected closed set of real values, hence some set operations on intervals apply to intervals.

Infimum/supremum

The infimum and supremum of an interval are respectively its inferior and superior endpoints. For $\mathbf{a} = [\underline{a}, \overline{a}] \in \mathbb{IR}$, then

$$\begin{aligned} \mathtt{inf}(\mathbf{a}) &= \underline{a}, \\ \mathtt{sup}(\mathbf{a}) &= \overline{a}. \end{aligned}$$

Maximal magnitude value, denoted by $\mathtt{mag}(.)$ or $|.|$, is the maximal absolute value of all real numbers taken from the interval. For $\mathbf{a} = [\underline{a}, \overline{a}] \in \mathbb{IR}$, then

$$\begin{aligned} \mathtt{mag}(\mathbf{a}) &= \mathtt{max}\{|a| : a \in \mathbf{a}\} \\ &= \mathtt{max}(|\underline{a}|, |\overline{a}|). \end{aligned}$$

Otherwise, the maximal magnitude value can be computed by the following proposition.

Proposition 1.3.1. *For $\mathbf{a} = [\underline{a}, \overline{a}] \in \mathbb{IR}$, it holds that*

$$|\mathbf{a}| = \mathtt{max}(-\underline{a}, \overline{a}).$$

Proof. We prove that $\mathtt{max}(-\underline{a}, \overline{a}) = \mathtt{max}(|\underline{a}|, |\overline{a}|)$. There are three possible cases:

- If $\underline{a} \geq 0$ then $\overline{a} \geq \underline{a} \geq 0 \geq -\underline{a}$. Hence $\mathtt{max}(|\underline{a}|, |\overline{a}|) = \overline{a}$, and $\mathtt{max}(-\underline{a}, \overline{a}) = \overline{a}$. It implies that $\mathtt{max}(-\underline{a}, \overline{a}) = \mathtt{max}(|\underline{a}|, |\overline{a}|)$.
- If $\overline{a} \leq 0$ then $\underline{a} \leq \overline{a} \leq 0 \leq -\underline{a}$. Hence $\mathtt{max}(|\underline{a}|, |\overline{a}|) = -\underline{a}$, and $\mathtt{max}(-\underline{a}, \overline{a}) = -\underline{a} = \mathtt{max}(|\underline{a}|, |\overline{a}|)$.
- If $\underline{a} < 0 < \overline{a}$ then $-\underline{a} = |\underline{a}|, \overline{a} = |\overline{a}|$. Hence $\mathtt{max}(-\underline{a}, \overline{a}) = \mathtt{max}(|\underline{a}|, |\overline{a}|)$.

□

Minimal magnitude value, denoted by mig(.) or $\langle . \rangle$, is the minimal absolute value of all real numbers taken from the interval. For $\mathbf{a} = [\underline{a}, \overline{a}] \in \mathbb{IR}$, then

$$\begin{aligned} \mathtt{mig}(\mathbf{a}) &= \mathtt{min}\{|a| : a \in \mathbf{a}\} \\ &= \begin{cases} 0 & \textit{if } \underline{a} < 0 < \overline{a}, \\ \mathtt{min}(|\underline{a}|, |\overline{a}|) & \textit{otherwise.} \end{cases} \end{aligned}$$

We can also compute the minimal magnitude value by the following proposition.

Proposition 1.3.2. *For* $\mathbf{a} = [\underline{a}, \overline{a}] \in \mathbb{IR}$*, it holds that*

$$\langle \mathbf{a} \rangle = \mathtt{max}(0, \mathtt{max}(\underline{a}, -\overline{a})).$$

The proof is by inspecting the same three cases as in the proof of Proposition 1.3.1.

Midpoint

$$\mathtt{mid}(\mathbf{a}) = \frac{1}{2}(\underline{a} + \overline{a}).$$

Radius

$$\mathtt{rad}(\mathbf{a}) = \frac{1}{2}(\overline{a} - \underline{a}).$$

1.3.2 Algebraic properties

Let $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{IR}$ be three intervals. The following laws are true:

1. neutral and unit elements: $\mathbf{a} + 0 = \mathbf{a}$ and $\mathbf{a} * 1 = \mathbf{a}$,
2. commutativity: $\mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a}$ and $\mathbf{a} * \mathbf{b} = \mathbf{b} * \mathbf{a}$,
3. associativity: $(\mathbf{a} + \mathbf{b}) + \mathbf{c} = \mathbf{a} + (\mathbf{b} + \mathbf{c})$ and $(\mathbf{a} * \mathbf{b}) * \mathbf{c} = \mathbf{a} * (\mathbf{b} * \mathbf{c})$,
4. sub-distributivity: $\mathbf{a} * (\mathbf{b} + \mathbf{c}) \subseteq \mathbf{a} * \mathbf{b} + \mathbf{a} * \mathbf{c}$.

The sub-distributivity property is one of the important sources of overestimation in interval arithmetic.

Example 1. Let $\mathbf{a} = [1, 2]$, $\mathbf{b} = [-1, 2]$, $\mathbf{c} = [2, 3]$.

$$\begin{aligned} \mathbf{a} * \mathbf{b} + \mathbf{a} * \mathbf{c} &= [1, 2] * [-1, 2] + [1, 2] * [2, 3] = [-2, 4] + [2, 6] = [0, 10] \\ \mathbf{a} * (\mathbf{b} + \mathbf{c}) &= [1, 2] * ([-1, 2] + [2, 3]) = [1, 2] * [1, 5] = [1, 10]\,. \end{aligned}$$

One can see that $\mathbf{a} * (\mathbf{b} + \mathbf{c}) \subset \mathbf{a} * \mathbf{b} + \mathbf{a} * \mathbf{c}$ and the inclusion is strict.

1.3.3 Absolute value properties

Some properties of the absolute value for real numbers can be extended to intervals.

Proposition 1.3.3. *For* $\mathbf{a}, \mathbf{b} \in \mathbb{IR}$*, the following properties hold:*

(1) $|-\mathbf{a}| = |\mathbf{a}|$ *and* $\langle -\mathbf{a} \rangle = \langle \mathbf{a} \rangle$*,*

(2) triangular inequality: $|\mathbf{a}| - |\mathbf{b}| \leq |\mathbf{a} \pm \mathbf{b}| \leq |\mathbf{a}| + |\mathbf{b}|$*,*

(3) $|\mathbf{a} * \mathbf{b}| = |\mathbf{a}| * |\mathbf{b}|$*,*

(4) $\langle \mathbf{a} \pm \mathbf{b} \rangle \leq \langle \mathbf{a} \rangle + \langle \mathbf{b} \rangle$*,*

(5) $\langle \mathbf{a} * \mathbf{b} \rangle = \langle \mathbf{a} \rangle * \langle \mathbf{b} \rangle$*.*

Proof. (1),(2),(3),(5) can be deduced directly from the properties of the real absolute value. Let us prove (4). For all $a \in \mathbf{a}, b \in \mathbf{b}$, we have

$$|a| - |b| \leq |a \pm b| \leq |a| + |b| \qquad |a * b| = |a| * |b|.$$

Let $\hat{a} \in \mathbf{a}, \hat{b} \in \mathbf{b}$ such that $|\hat{a}| = \langle \mathbf{a} \rangle$ and $|\hat{b}| = \langle \mathbf{b} \rangle$. It is obvious that $\hat{a} \pm \hat{b} \in \mathbf{a} \pm \mathbf{b}$. Therefore $\langle \mathbf{a} \pm \mathbf{b} \rangle \leq |\hat{a} \pm \hat{b}| \leq |\hat{a}| + |\hat{b}| \leq \langle \mathbf{a} \rangle + \langle \mathbf{b} \rangle$. It implies (4). □

Note that the following inequality is not always true: $\langle \mathbf{a} \rangle - \langle \mathbf{b} \rangle \leq \langle \mathbf{a} \pm \mathbf{b} \rangle$. For example, if $\mathbf{a} = [1, 2]$ and $\mathbf{b} = [-1, 2]$ then $\langle \mathbf{a} \rangle - \langle \mathbf{b} \rangle = 1 - 0 = 1$. Meanwhile, $\langle \mathbf{a} + \mathbf{b} \rangle = \langle [0, 4] \rangle = 0$. Hence, $\langle \mathbf{a} \rangle - \langle \mathbf{b} \rangle > \langle \mathbf{a} + \mathbf{b} \rangle$. On the other hand, if $\mathbf{a} \geq 0$ and $\mathbf{b} \geq 0$ then $\langle \mathbf{a} + \mathbf{b} \rangle = \underline{a} + \underline{b} = \langle \mathbf{a} \rangle + \langle \mathbf{b} \rangle \geq \langle \mathbf{a} \rangle - \langle \mathbf{b} \rangle$.

1.3.4 Midpoint and radius properties

Both addition and subtraction increase the width of intervals

$$\mathtt{rad}(\mathbf{a} \pm \mathbf{b}) = \mathtt{rad}(\mathbf{a}) + \mathtt{rad}(\mathbf{b}).$$

The following proposition assesses the width of interval multiplication.

Proposition 1.3.4. *[MKC09] For* $\mathbf{a}, \mathbf{b} \in \mathbb{IR}$*, the following properties hold:*

(1) $|\mathbf{a}| = |\mathtt{mid}(\mathbf{a})| + \mathtt{rad}(\mathbf{a})$*,*

(2) $\mathtt{rad}(\mathbf{a} * \mathbf{b}) = |\mathbf{a}| * \mathtt{rad}(\mathbf{b})$ *if* $\mathbf{a}$ *is thin or* $\mathbf{b}$ *is centered in zero,*

(3) $\mathtt{rad}(\mathbf{a} * \mathbf{b}) \leq |\mathtt{mid}(\mathbf{a})| * \mathtt{rad}(\mathbf{b}) + \mathtt{rad}(\mathbf{a}) * |\mathtt{mid}(\mathbf{b})| + \mathtt{rad}(\mathbf{a}) * \mathtt{rad}(\mathbf{b})$*,*

(4) $\mathtt{rad}(\mathbf{a} * \mathbf{b}) \leq |\mathbf{a}| * \mathtt{rad}(\mathbf{b}) + \mathtt{rad}(\mathbf{a}) * |\mathbf{b}|$*,*

(5) $\mathtt{rad}(\mathbf{a} * \mathbf{b}) \geq \mathtt{rad}(\mathbf{a}) * \mathtt{rad}(\mathbf{b}) + \langle \mathbf{a} \rangle * \mathtt{rad}(\mathbf{b}) + \mathtt{rad}(\mathbf{a}) * \langle \mathbf{b} \rangle$*.*

Proof.

(1) By definition, $|\mathbf{a}| = \mathtt{max}(|\underline{a}|, |\overline{a}|) = \mathtt{max}(|\mathtt{mid}(\mathbf{a}) - \mathtt{rad}(\mathbf{a})|, |\mathtt{mid}(\mathbf{a}) + \mathtt{rad}(\mathbf{a})|) = |\mathtt{mid}(\mathbf{a})| + \mathtt{rad}(\mathbf{a})$.

(2) If $\mathbf{a}$ is thin, i.e $\underline{a} = \overline{a}$ then $\mathbf{a} * \mathbf{b}$ is equal to either $\left[\overline{a} * \underline{b}, \overline{a} * \overline{b}\right]$ or $\left[\overline{a} * \overline{b}, \overline{a} * \underline{b}\right]$, which implies that $\mathtt{rad}(\mathbf{a}*\mathbf{b}) = 1/2|\overline{a}*\overline{b}-\overline{a}*\underline{b}| = 1/2|\overline{a}(\overline{b}-\underline{b})| = |\overline{a}|*1/2|\overline{b}-\underline{b}| = |\mathbf{a}|*\mathtt{rad}(\mathbf{b})$. If $\mathbf{b}$ is centered in zero, i.e $-\underline{b} = \overline{b} = \mathtt{rad}(\mathbf{b}) = |\mathbf{b}|$, then for all $a \in \mathbf{a}, b \in \mathbf{b}$ we have $a * \underline{b} = -a * \overline{b}, |b| \leq |\mathbf{b}|$, and $-|a| * |\mathbf{b}| \leq |a * b| \leq |a| * |\mathbf{b}|$, which implies that $a*\mathbf{b} = [-|a| * |\mathbf{b}|, |a| * |\mathbf{b}|]$. Hence $\mathbf{a}*\mathbf{b} = \Box\{a*\mathbf{b}, a \in \mathbf{a}\} = \Box\{[-|a| * |\mathbf{b}|, |a| * |\mathbf{b}|], a \in \mathbf{a}\} = [-|\mathbf{a}| * |\mathbf{b}|, |\mathbf{a}| * |\mathbf{b}|] \Rightarrow \mathtt{rad}(\mathbf{a} * \mathbf{b}) = |\mathbf{a}| * |\mathbf{b}| = |\mathbf{a}| * \mathtt{rad}(\mathbf{b})$ and $\mathtt{mid}(\mathbf{a} * \mathbf{b}) = 0$.

(3) This proof is established by differentiating the cases where $\mathbf{a} \geq 0$, $\mathbf{a} \leq 0$ and $\mathbf{a} \ni 0$ and similarly for $\mathbf{b}$. When an interval contains 0, one must furthermore distinguish according to the sign of the midpoint of the interval. The cases where the two intervals do not contain 0 and where one interval contains 0 are detailed below, the other cases are similar.

When $\mathbf{a} \leq 0$ and $\mathbf{b} \geq 0$, then the formula which applies for the product $\mathbf{a} * \mathbf{b}$ is $\mathbf{a} * \mathbf{b} = [\underline{a} * \overline{b}, \overline{a} * \underline{b}]$ and thus $\mathtt{rad}(\mathbf{a} * \mathbf{b}) = \frac{1}{2}(\overline{a} * \underline{b} - \underline{a} * \overline{b})$. The right-hand side of the inequality can be calculated and simplified into $\frac{1}{4}(\underline{a}*\underline{b} - 3\underline{a}*\overline{b} + \overline{a}*\underline{b} + \overline{a}*\overline{b})$. Checking whether $\mathtt{rad}(\mathbf{a} * \mathbf{b}) \leq \frac{1}{4}(\underline{a} * \underline{b} - 3\underline{a} * \overline{b} + \overline{a} * \underline{b} + \overline{a} * \overline{b})$ is equivalent to checking whether $(\overline{a} - \underline{a}) * (\overline{b} - \underline{b}) \geq 0$, which holds.

When $\mathbf{a} \geq 0$ and $\mathbf{b} \ni 0$, then the formula which applies for the product $\mathbf{a} * \mathbf{b}$ is $\mathbf{a}*\mathbf{b} = [\overline{a}*\underline{b}, \overline{a}*\overline{b}]$ and thus $\mathtt{rad}(\mathbf{a}*\mathbf{b}) = \frac{1}{2}\overline{a}*(\overline{b}-\underline{b})$. To be able to calculate and simplify the right-hand side of the inequality, one must distinguish further according to the sign of $\mathtt{mid}(\mathbf{b})$. If $\mathtt{mid}(\mathbf{b}) \geq 0$ then this right-hand side is equal to $\frac{1}{4}(-\underline{a}*\underline{b}-\underline{a}*\overline{b}-\overline{a}*\underline{b}+3\overline{a}*\overline{b})$. Checking whether $\mathtt{rad}(\mathbf{a}*\mathbf{b}) \leq \frac{1}{4}(-\underline{a}*\underline{b}-\underline{a}*\overline{b}-\overline{a}*\underline{b}+3\overline{a}*\overline{b})$ is equivalent to checking whether $(\overline{a}-\underline{a})*(\underline{b}+\overline{b}) \geq 0$, which holds since $\mathtt{mid}(\mathbf{b}) = \frac{1}{2}(\underline{b}+\overline{b})$ is assumed to be non-negative. If $\mathtt{mid}(\mathbf{b}) \leq 0$ then this right-hand side is equal to $\frac{1}{4}(\underline{a}*\underline{b}+\underline{a}*\overline{b}-3\overline{a}*\underline{b}+\overline{a}*\overline{b})$. Checking whether $\mathtt{rad}(\mathbf{a} * \mathbf{b}) \leq \frac{1}{4}(\underline{a} * \underline{b} + \underline{a} * \overline{b} - 3\overline{a} * \underline{b} + \overline{a} * \overline{b})$ is equivalent to checking whether $-(\overline{a} - \underline{a}) * (\underline{b} + \overline{b}) \geq 0$, which holds since $\mathtt{mid}(\mathbf{b}) = \frac{1}{2}(\underline{b} + \overline{b})$ is assumed to be non-positive.

(4) Inequality (3) can also be written as $\mathtt{rad}(\mathbf{a} * \mathbf{b}) \leq (|\mathtt{mid}(\mathbf{a})| + \mathtt{rad}(\mathbf{a})) * |\mathtt{mid}(\mathbf{b})| + \mathtt{rad}(\mathbf{a}) * \mathtt{rad}(\mathbf{b})$, and using (1) one gets

$$\begin{aligned}\mathtt{rad}(\mathbf{a} * \mathbf{b}) &\leq |\mathbf{a}| * |\mathtt{mid}(\mathbf{b})| + \mathtt{rad}(\mathbf{a}) * \mathtt{rad}(\mathbf{b}) \\ &\leq |\mathbf{a}| * |\mathtt{mid}(\mathbf{b})| + \mathtt{rad}(\mathbf{a}) * |\mathtt{mid}(\mathbf{b})| + \mathtt{rad}(\mathbf{a}) * \mathtt{rad}(\mathbf{b}).\end{aligned}$$

Using (1) again, one gets a formula with a right-hand side symmetric in $\mathbf{a}$ and $\mathbf{b}$: $\mathtt{rad}(\mathbf{a} * \mathbf{b}) \leq |\mathbf{a}| * \mathtt{rad}(\mathbf{b}) + \mathtt{rad}(\mathbf{a}) * |\mathbf{b}|$.

(5) Let $\alpha = \mathtt{sign}(\mathtt{mid}(\mathbf{a})) * \mathtt{rad}(\mathbf{a})$, $\beta = \mathtt{sign}(\mathtt{mid}(\mathbf{b}))\mathtt{rad}(\mathbf{b})$. Then all of $\mathtt{mid}(\mathbf{a}) \pm \alpha \in \mathbf{a}$, $\mathtt{mid}(\mathbf{b}) \pm \beta \in \mathbf{b}$, and $\mathtt{mid}(\mathbf{a}) * \mathtt{mid}(\mathbf{b})$, $\mathtt{mid}(\mathbf{a}) * \beta$, $\alpha * \mathtt{mid}(\mathbf{b})$, $\alpha * \beta$ have the same sign.

- If $0 \in \mathbf{a}$ and $0 \in \mathbf{b}$ then $\langle \mathbf{a} \rangle = 0$ and $\langle \mathbf{b} \rangle = 0$. Both $(\mathtt{mid}(\mathbf{a}) + \alpha) * (\mathtt{mid}(\mathbf{b}) + \beta)$ and $(\mathtt{mid}(\mathbf{a}) + \alpha) * (\mathtt{mid}(\mathbf{b}) - \beta)$ are included in $\mathbf{a} * \mathbf{b}$, hence

$$\begin{aligned}\mathtt{rad}(\mathbf{a} * \mathbf{b}) &\geq 1/2|(\mathtt{mid}(\mathbf{a}) + \alpha) * (\mathtt{mid}(\mathbf{b}) + \beta) - (\mathtt{mid}(\mathbf{a}) + \alpha) * (\mathtt{mid}(\mathbf{b}) - \beta)| \\ &\geq |\mathtt{mid}(\mathbf{a}) * \beta + \alpha * \beta| \\ &\geq |\mathtt{mid}(\mathbf{a})| * |\beta| + |\alpha| * |\beta| \geq |\alpha| * |\beta| \\ &\geq \mathtt{rad}(\mathbf{a}) * \mathtt{rad}(\mathbf{b}) \\ &\geq \mathtt{rad}(\mathbf{a}) * \mathtt{rad}(\mathbf{b}) + \langle \mathbf{a} \rangle * \mathtt{rad}(\mathbf{b}) + \mathtt{rad}(\mathbf{a}) * \langle \mathbf{b} \rangle.\end{aligned}$$

- If $0 \in \mathbf{a}$ and $0 \notin \mathbf{b}$ then $\langle\mathbf{a}\rangle = 0$. Moreover, $(\mathtt{mid}(\mathbf{a}) + \mathtt{rad}(\mathbf{a})) * \mathtt{mid}(\mathbf{b}) \in \mathbf{a} * \mathbf{b}$, and $(\mathtt{mid}(\mathbf{a}) - \mathtt{rad}(\mathbf{a})) * \mathtt{mid}(\mathbf{b}) \in \mathbf{a} * \mathbf{b}$, hence,

$$\begin{aligned}\mathtt{rad}(\mathbf{a} * \mathbf{b}) &\geq 1/2|(\mathtt{mid}(\mathbf{a}) + \mathtt{rad}(\mathbf{a})) * \mathtt{mid}(\mathbf{b}) - (\mathtt{mid}(\mathbf{a}) - \mathtt{rad}(\mathbf{a})) * \mathtt{mid}(\mathbf{b})| \\ &\geq |\mathtt{rad}(\mathbf{a}) * \mathtt{mid}(\mathbf{b})| = \mathtt{rad}(\mathbf{a}) * |\mathtt{mid}(\mathbf{b})|.\end{aligned}$$

As $|\mathtt{mid}(\mathbf{b})| = \langle\mathbf{b}\rangle + \mathtt{rad}(\mathbf{b})$, then

$$\begin{aligned}\mathtt{rad}(\mathbf{a} * \mathbf{b}) &\geq \mathtt{rad}(\mathbf{a}) * (\langle\mathbf{b}\rangle + \mathtt{rad}(\mathbf{b})) \\ &\geq \mathtt{rad}(\mathbf{a}) * \langle\mathbf{b}\rangle + \mathtt{rad}(\mathbf{a}) * \mathtt{rad}(\mathbf{b}) \\ &\geq \mathtt{rad}(\mathbf{a}) * \mathtt{rad}(\mathbf{b}) + \mathtt{rad}(\mathbf{a}) * \langle\mathbf{b}\rangle + \langle\mathbf{a}\rangle * \mathtt{rad}(\mathbf{b}).\end{aligned}$$

- The case $0 \notin \mathbf{a}$ and $0 \in \mathbf{b}$ is similar, as $\mathbf{a}$ and $\mathbf{b}$ play symmetric roles.
- If $0 \notin \mathbf{a}$ and $0 \notin \mathbf{b}$. Again, $|\mathtt{mid}(\mathbf{a})| = \langle\mathbf{a}\rangle + \mathtt{rad}(\mathbf{a})$, $|\mathtt{mid}(\mathbf{b})| = \langle\mathbf{b}\rangle + \mathtt{rad}(\mathbf{b})$. Both $(\mathtt{mid}(\mathbf{a}) + \alpha) * (\mathtt{mid}(\mathbf{b}) + \beta)$ and $(\mathtt{mid}(\mathbf{a}) - \alpha) * (\mathtt{mid}(\mathbf{b}) - \beta)$ are included in $\mathbf{a} * \mathbf{b}$, hence, thanks to the sign conditions imposed on α and β,

$$\begin{aligned}\mathtt{rad}(\mathbf{a} * \mathbf{b}) &\geq 1/2|(\mathtt{mid}(\mathbf{a}) + \alpha) * (\mathtt{mid}(\mathbf{b}) + \beta) - (\mathtt{mid}(\mathbf{a}) - \alpha) * (\mathtt{mid}(\mathbf{b}) - \beta)| \\ &\geq |\mathtt{mid}(\mathbf{a}) * \beta + \alpha * \mathtt{mid}(\mathbf{b})| \\ &\geq |\mathtt{mid}(\mathbf{a})| * |\beta| + |\alpha| * |\mathtt{mid}(\mathbf{b})| \\ &\geq (\langle\mathbf{a}\rangle + \mathtt{rad}(\mathbf{a})) * \mathtt{rad}(\mathbf{b}) + \mathtt{rad}(\mathbf{a}) * (\langle\mathbf{b}\rangle + \mathtt{rad}(\mathbf{b})) \\ &\geq \mathtt{rad}(\mathbf{a}) * \mathtt{rad}(\mathbf{b}) + \langle\mathbf{a}\rangle * \mathtt{rad}(\mathbf{b}) + \mathtt{rad}(\mathbf{a}) * \langle\mathbf{b}\rangle.\end{aligned}$$

□

There is no cancellation in interval arithmetic. For every non-degenerate interval $\mathbf{a} \in \mathbb{IR}$, then $\mathbf{a} - \mathbf{a}$ strictly contains 0. This phenomenon is called sub-cancellation. By definition, $\mathbf{a} - \mathbf{a} = [\underline{a} - \overline{a}, \overline{a} - \underline{a}]$, hence the result is in fact an interval which is centered in zero. For example, if $\mathbf{a} = [1, 2]$, then $\mathbf{a} - \mathbf{a} = [-1, 1]$. Another way of expressing the sub-cancellation property is: $(\mathbf{a} - \mathbf{b}) + \mathbf{b} \supseteq \mathbf{a}$.

Similarly, division is not the inverse of the multiplication: for every non-degenerate interval $\mathbf{a} \in \mathbb{IR}$, then $\mathbf{a}/\mathbf{a}$ strictly contains 1. For example, if $\mathbf{a} = [1, 2]$, then $\mathbf{a}/\mathbf{a} = [1/2, 2] \ni 1$.

Proposition 1.3.5. *For $\mathbf{a} \in \mathbb{IR}, \mathbf{b} \in \mathbb{IR}$, such that $\mathbf{b}$ does not contain zero, $\gamma \in \mathbb{R}$ and $\gamma \neq 0$, the following holds*

(1) $|\mathbf{a}/\mathbf{b}| = |\mathbf{a}|/\langle\mathbf{b}\rangle$,

(2) $\mathtt{rad}(\mathbf{a}/\gamma) = \mathtt{rad}(\mathbf{a})/|\gamma|$,

(3) $\mathtt{rad}(\mathbf{a}/\mathbf{b}) \geq \mathtt{rad}(\mathbf{a})/\langle\mathbf{b}\rangle$,

(4) $|\mathtt{mid}(\mathbf{a}/\mathbf{b})| \leq |\mathtt{mid}(\mathbf{a})|/\langle\mathbf{b}\rangle$.

Proof. (1) is trivial. If $\gamma > 0$ then $\mathbf{a}/\gamma = [\underline{a}/\gamma, \overline{a}/\gamma]$, hence $\mathtt{rad}(\mathbf{a}/\gamma) = 1/2(\overline{a}/\gamma - \underline{a}/\gamma) = 1/2(\overline{a} - \underline{a})/\gamma = \mathtt{rad}(\mathbf{a})/|\gamma|$. On the other hand, if $\gamma < 0$ then $\mathbf{a}/\gamma = [\overline{a}/\gamma, \underline{a}/\gamma]$, hence $\mathtt{rad}(\mathbf{a}/\gamma) = 1/2(\underline{a}/\gamma - \overline{a}/\gamma) = 1/2(\overline{a} - \underline{a})/(-\gamma) = \mathtt{rad}(\mathbf{a})/|\gamma|$. Therefore we have (2). Let $b \in \mathbf{b}$ such that $|b| = \langle\mathbf{b}\rangle$, then $\mathbf{a}/b \subseteq \mathbf{a}/\mathbf{b}$. Hence $\mathtt{rad}(\mathbf{a}/\mathbf{b}) \geq \mathtt{rad}(\mathbf{a}/b) = \mathtt{rad}(\mathbf{a})/|b|$, which implies (3). Following Proposition 1.3.4, $|\mathtt{mid}(\mathbf{a}/\mathbf{b})| = |\mathbf{a}/\mathbf{b}| - |\mathtt{rad}(\mathbf{a}/\mathbf{b})|$. Therefore, following (1) and (3), $|\mathtt{mid}(\mathbf{a}/\mathbf{b})| = |\mathbf{a}|/\langle\mathbf{b}\rangle - |\mathtt{rad}(\mathbf{a}/\mathbf{b})| \leq |\mathbf{a}|/\langle\mathbf{b}\rangle - \mathtt{rad}(\mathbf{a})/\langle\mathbf{b}\rangle \leq (|\mathbf{a}| - \mathtt{rad}(\mathbf{a}))/\langle\mathbf{b}\rangle$. Moreover, $|\mathbf{a}| - \mathtt{rad}(\mathbf{a}) = |\mathtt{mid}(\mathbf{a})|$, thus we have (4). □

1.3.5 Variable dependency

Sub-cancellation and sub-distributivity properties are in fact the simplest cases of the dependency problem in interval arithmetic. When an expression is evaluated in floating-point arithmetic, each occurrence of the same variable takes exactly the same value. Differently, when the expression is evaluated using interval arithmetic, each occurrence of a variable represents a range of values. The computation assumes implicitly that the quantity in each occurrence of a variable varies independently from the quantities in other occurrences of the same variable. It means that we no longer have the dependency between occurrences of a single variable when evaluating an interval expression whereas, in fact, they are correlated. That leads to an unwanted expansion of the computed result even if each interval operation is evaluated exactly. The difference between the interval arithmetic evaluation and the actual range of an expression is called overestimation.

Consider the following example of interval expression evaluation:

$$\mathbf{x}^2 - 2\mathbf{x} \quad \text{where} \quad \mathbf{x} = [1,2]\,.$$

Performing interval operations, we have $\mathbf{x}^2 = [1,4]$, $2\mathbf{x} = [2,4]$, and finally $\mathbf{x}^2 - 2\mathbf{x} = [-3,2]$.

Meanwhile, if we rewrite the above expression as follows

$$(\mathbf{x} - 1)^2 - 1$$

then the computed result is: $[0,1]^2 - 1 = [-1,0]$.

As this can be seen in the above example, mathematically equivalent forms give different results when they are evaluated in interval arithmetic. That is because by rewriting the expression, the dependency between variables is expressed differently. One rule to obtain an optimal result is to rewrite the expression in such a way that each variable appears only once in the expression. In the above example, the variable $\mathbf{x}$ appears only once in the second expression, thus the result computed by the second expression is sharper than that computed by the first expression, and it is even exactly the range of this expression.

Nevertheless, not all expressions can be rewritten that way. Therefore the dependency problem is a major source of overestimation. Another source of overestimation comes from the so-called wrapping effect when computing in multiple dimensions.

1.3.6 Wrapping effect

The wrapping effect is the overestimation due to the enclosure of multidimensional sets by the Cartesian product of intervals, i.e. geometrically, by boxes with sides parallel to the axes.

Example 2. Let $\mathbf{x} = \{[4,6], [-1,1]\} \in \mathbb{IR}^2$ be a box whose coordinates are the two intervals $\mathbf{x}_1 = [4,6]$, and $\mathbf{x}_2 = [-1,1]$. This Cartesian product of intervals $\mathbf{x}$ is depicted by the black box in Figure 1.1.

Let us now rotate $\mathbf{x}$ by a rotation matrix $A = \begin{pmatrix} 4/5 & -3/5 \\ 3/5 & 4/5 \end{pmatrix}$ and then rotate it back by the inverse matrix $A^{-1} = \begin{pmatrix} 4/5 & 3/5 \\ -3/5 & 4/5 \end{pmatrix}$.

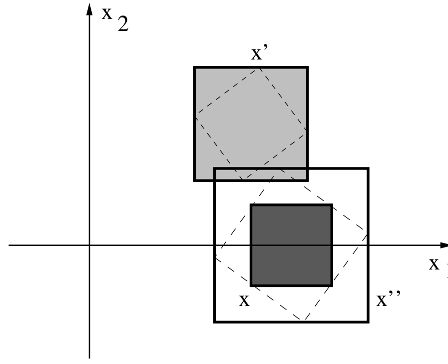


Figure 1.1: Wrapping effect.

In theory, $\mathbf{x}$ should not change after these two rotations. However, this computation performed using interval arithmetic gives a result which is a larger interval vector $\mathbf{x}''$ including $\mathbf{x}$ inside its interior. After the first rotation, we get an interval vector $\mathbf{x}'$, represented by the gray box in Figure 1.1.

$$\mathbf{x}' = \begin{pmatrix} 4/5 & -3/5 \\ 3/5 & 4/5 \end{pmatrix} \begin{pmatrix} [4,6] \\ [-1,1] \end{pmatrix} = \begin{pmatrix} [13/5, 27/5] \\ [8/5, 22/5] \end{pmatrix}.$$

This box $\mathbf{x}'$ wraps the exact image of $\mathbf{x}$ (dotted lines) in a box whose sides are parallel to the axes. After the second rotation, we get the final result, represented by the white box

$$\mathbf{x}'' = \begin{pmatrix} 4/5 & 3/5 \\ -3/5 & 4/5 \end{pmatrix} \begin{pmatrix} [13/5, 27/5] \\ [8/5, 22/5] \end{pmatrix} = \begin{pmatrix} [76/25, 174/25] \\ [-49/25, 49/25] \end{pmatrix}.$$

One can see on Figure 1.1 that $\mathbf{x}''$ is much larger than $\mathbf{x}$. The factor of overestimation of $\mathbf{x}''$ with respect to $\mathbf{x}$ is $f = \{49/25, 49/25\}$, which means that each side of the original box has been expanded by a factor of nearly 2.

1.3.7 Implementation

Interval arithmetic is implemented on computers using floating-point arithmetic. The most common representation of intervals uses their endpoints. Each endpoint of an interval is encoded by one floating-point number. Therefore, we have a new set of intervals whose endpoints are discrete

$$\mathbb{IF} = \{[\underline{a}, \overline{a}] \in \mathbb{IR}, \quad \underline{a}, \overline{a} \in \mathbb{F}\}.$$

Using floating-point arithmetic, we have to handle the problem of rounding errors. In the presence of rounding errors, the exactness of endpoint computations is no longer guaranteed. Let us consider, for example, the addition of two intervals. The result's upper bound is, by definition, the sum of the upper bounds of the two summands, which is a real number and in most cases not representable by a floating-point number. Hence, to guarantee the isotonicity property, we adapt the definition to operations on $\mathbb{IF}$.

Definition 1.3.6. *For $\mathbf{x} \in \mathbb{IF}, \mathbf{y} \in \mathbb{IF}$, the result of an operation $\mathbf{x} \circ \mathbf{y}$, $\circ \in \{+ - * /\}$ is the smallest interval $\mathbf{z} \in \mathbb{IF}$ containing*

$$\{z = x \circ y \quad x \in \mathbf{x}, y \in \mathbf{y}\}.$$

Directed rounding modes

To satisfy the principle of inclusion, the upper bound of the computed result must be greater than or equal to the upper bound of the exact interval result. Similarly, the lower bound of the computed result must be smaller than or equal to the lower bound of the exact interval result. Therefore, one straightforward and effective way to handle rounding errors when implementing interval arithmetic is to used directed rounding modes, namely the upward rounding mode for the upper bound and the downward rounding mode for the lower bound.

In upward rounding mode, the result of an arithmetic operation is the smallest floating-point number which is greater than or equal to the exact result. In downward rounding mode, the result of an arithmetic operation is the greatest floating-point number which is smaller than or equal to the exact result. In this document, the following rounding modes are used:

$$\begin{array}{ll} \triangle & \text{upward rounding mode,} \\ \triangledown & \text{downward rounding mode.} \end{array}$$

For ranges of arithmetic operations, directed rounding modes allow to obtain a rigorous and tight enclosure of the exact result interval.

Let $\mathbf{a} = [\underline{a}, \overline{a}] \in \mathbb{IF}, \mathbf{b} = [\underline{b}, \overline{b}] \in \mathbb{IF}$. Using directed rounding modes, basic interval arithmetic operations can easily be implemented.

Addition

$$\mathbf{a} \oplus \mathbf{b} = \left[\triangledown\left(\underline{a} + \underline{b}\right), \triangle\left(\overline{a} + \overline{b}\right)\right].$$

The isotonicity of this operation is simple. For all $a \in \mathbf{a}, b \in \mathbf{b}$, it is obvious that

$$\triangledown\left(\underline{a} + \underline{b}\right) \leq \underline{a} + \underline{b} \leq a + b \leq \overline{a} + \overline{b} \leq \triangle\left(\overline{a} + \overline{b}\right).$$

Hence, $a + b \in \mathbf{a} + \mathbf{b} \subset \mathbf{a} \oplus \mathbf{b}$. In much the same way, we can prove the isotonicity of the following operations.

Subtraction

$$\mathbf{a} \ominus \mathbf{b} = \left[\triangledown\left(\underline{a} - \overline{b}\right), \triangle\left(\overline{a} - \underline{b}\right)\right].$$

Multiplication

$$\begin{aligned} \mathbf{a} \otimes \mathbf{b} = \big[&\mathtt{min}(\triangledown\left(\underline{a} * \underline{b}\right), \triangledown\left(\underline{a} * \overline{b}\right), \triangledown\left(\overline{a} * \underline{b}\right), \triangledown\left(\overline{a} * \overline{b}\right)), \\ &\mathtt{max}(\triangle\left(\underline{a} * \underline{b}\right), \triangle\left(\underline{a} * \overline{b}\right), \triangle\left(\overline{a} * \underline{b}\right), \triangle\left(\overline{a} * \overline{b}\right))\big]. \end{aligned}$$

In exact arithmetic, an interval multiplication requires four punctual products. However, when floating-point arithmetic is used to implement interval arithmetic, each sub-product must be computed twice, once in downward rounding mode and once in upward rounding mode. Hence, in total, an interval multiplication requires eight floating-point products. As with real interval arithmetic, one can inspect the sign of input intervals to reduce the number of sub-products. This solution suffers from low performance due to branching and testing the sign of floating-point numbers.

For the rest of the document, we will refer to this implementation as directed interval arithmetic.

Successor and predecessor

Directed rounding modes might not be supported on some architectures. For example, on the Synergistic Processor Elements of the IBM Cell Broadband Engine Architecture [KDH+05], only the rounding mode toward zero, also called truncation rounding mode, is supported for IEEE single precision. In addition, some programming languages, Java for example, do not support functions for changing rounding modes. In these cases, interval arithmetic can be implemented using predecessor and successor functions.

Predecessor and successor functions consist in computing respectively the previous and the next floating-point number with respect to a given floating-point number. Hence, they are used to simulate downward and upward rounding modes. Work on predecessor and successor functions, or interval arithmetic in rounding to nearest can be found in [RZBM09]. One advantage of this approach over directed rounding mode interval arithmetic is that it does not require any rounding mode changes. Hence, it triggers no context switching during computation and can be better pipelined. Nevertheless, this approach suffers from a significant overhead of execution time introduced by predecessor and successor functions.

In this document, we assume that directed rounding mode interval arithmetic is used. We will show that the problem of changing rounding mode can be overcome for operations on matrices and vectors.

Implementation issues

In IEEE 754-1985 processors, i.e. on most processors including IA-32 [Int07], changing the rounding mode of floating-point operations is done by changing the value of some global register. This requires flushing the floating-point pipeline: all the floating-point operations initiated beforehand must be terminated before performing the new ones with the new rounding mode. Therefore changing rounding mode frequently as required to implement interval arithmetic will lead to a dramatic slowdown in performance: in practice, the computation is slowed down compared to its floating-point counterpart, by a factor between 20 and 50 [Rum99a].

This problem can be eradicated in more recent instruction sets which allow to change rounding-mode per-instruction without penalty [MBdD+10]. Unfortunately, this feature can be accessed only from low-level programming language by manipulating directly the instruction word, not from current high level languages which are designed to be compliant with the IEEE 754-1985 [IEE85] standard. The new IEEE 754-2008 [IEE08] standard for floating-point arithmetic corrects this issue. However, it may take some time before it is fully supported.

Otherwise, the inclusion property of interval arithmetic usually requires obtaining the maximal and minimal values from intermediate results, e.g. interval multiplication or square. These maximum and minimum functions require themselves at least a comparison followed by a selection of the right value. They are usually coded by a test and branch or on some particular processors, such as the CELL processor, by comparison and bit selection. These `max` and `min` functions introduce a significant overhead of execution time.

With the obstacles mentioned above, a lot of work is needed in order to obtain an efficient interval arithmetic implementation, which is well pipelined and parallelized. It may even require redesigning and reimplementing everything from scratch. That is not desirable, as there exists a number of efficient implementations for the floating-point versions of the corresponding algorithms which have been well optimized and parallelized. A more

realistic solution is to exploit these existing optimized implementations to obtain an efficient implementation for interval arithmetic. This idea is the foundation of my Ph.D., targeting at an efficient implementation for verified linear algebra using interval arithmetic.

1.3.8 Implementation environment

During my Ph.D. preparation, I used the IntLab library, which is a library in MatLab developed by professor S.M. Rump [Rum99b] for interval arithmetic, to implement our algorithms. The main reason is that the IntLab library is based on BLAS and uses directed rounding modes to implement interval arithmetic. The BLAS (Basic Linear Algebra Subprograms) [LHKK79] are routines that provide standard building blocks for performing basic vector and matrix operations. To my knowledge, IntLab is the most efficient implementation of interval arithmetic for linear algebra. Its extensive use of the BLAS routines is the main key to achieve high speed for interval computations. Another main advantage of using the BLAS routines is that a code using the BLAS routines can run fast on a variety of computers. Since the BLAS routines are widely available, it is supported by a vast majority of processors. Manufacturers usually provide very fast, well optimized implementations of the BLAS routines for their specific computers.

Another reason to choose IntLab as the implementation environment is its ease of use. Since IntLab is a library in MatLab, which is an interactive programming environment, algorithms can be implemented in MatLab in such a way that is close to pseudo-code. Especially matrix operations are expressed in a very simple and straightforward way.

Nevertheless, the algorithms presented in this document are not restricted to be implemented using the IntLab library. The implementations presented in this document, which use the IntLab library, are only reference implementations to demonstrate the correctness as well as the performance of these algorithms. Otherwise, the presented algorithms can be implemented using any optimized BLAS library, in any programming language which supports functions for changing rounding modes.

1.4 Conclusion

Interval arithmetic paves the way for a new powerful tool to obtain verified results in scientific computing. Instead of giving an approximation, algorithms using interval arithmetic provide an enclosure of the results. It means that the exact results are guaranteed to be included in the computed results. On the contrary, a value not included in the computed results can never be the exact result. Therefore using interval arithmetic provides some level of certification. The accuracy of computed results can also be easily checked by assessing the width of computed intervals.

Nevertheless, there are obstacles that prevent interval arithmetic from being broadly used. Firstly, interval computations usually provide results of large width. Some important sources of overestimation can be named here: variable dependency and wrapping effects. A naive application of interval arithmetic to existing algorithms for floating-point arithmetic does not, in general case, provide sharp enclosures of the results. The width of intervals explodes very fast if care is not taken when implementing an algorithm in interval arithmetic.

Secondly, to ensure the isotonicity property of interval computations, redundant operations must be performed to obtain the convex hull of the exact results. Therefore the factor of interval arithmetic with respect to floating-point arithmetic in terms of execution time in theory is 4. It is even much worse in implementation when rounding errors and parallelism are taken into account.

Although some general techniques are available, such as making use of original punctual data or performing as many as possible real operations without having to change rounding mode, the way to obtain an efficient algorithm depends on every specific problem. Moreover, for many problems there is no perfect solution to both these two obstacles, i.e. there is no algorithm which provides very sharp enclosures and at the same time runs very fast. Therefore a tradeoff between accuracy and performance must usually be made.

In the context of this thesis, we deal with two fundamental problems of linear algebra, that are the multiplication of interval matrices (Chapter 2) and the verification of floating-point solutions of linear systems of equations (Chapter 3, 4 and 5). Nonetheless, techniques introduced in this document are applicable to other problems as well since they are based on interval properties such as symmetry and sub-distributivity.

Chapter 2

Interval matrix multiplication

This chapter presents two new algorithms for the multiplication of interval matrices, which are based on endpoints and midpoint-radius representations of interval respectively. These algorithms are based on a decomposition of intervals into two parts, one which is centered in zero, and the other which captures the largest part not containing zero in its interior. They are implemented using 9 and 7 floating-point matrix products respectively. They are a bit slower than the most efficient algorithm, to my knowledge, proposed by S. M. Rump, which costs only 4 floating-point matrix products. Nonetheless, the overestimation factor of the results computed by the two new algorithms is always bounded by 1.18, compared to 1.5 for Rump's algorithm. Both algorithms are implemented using IntLab, which is a BLAS-based library for interval arithmetic, to demonstrate their performance.

2.1 Introduction

In this chapter we study the case of matrix multiplication which involves $\mathcal{O}(n^3)$ operations on $\mathcal{O}(n^2)$ data. Interval matrix product implemented by the natural algorithm provides good results in terms of accuracy but suffers from low performance in terms of execution time. An idea is to exploit existing libraries which are well optimized for floating-point matrix operations such as BLAS, ATLAS, etc. Rump [Rum99a] proposed an efficient algorithm to calculate the interval matrix multiplication with a cost of 4 floating-point matrix products. The factor of overestimation in the worst case of this algorithm is 1.5. In this chapter, we propose a new algorithm which offers a trade-off between performance and accuracy: this algorithm costs 7 floating-point matrix products with a factor of overestimation in the worst case of 1.18.

2.2 State of the art

Let $\mathbf{A}, \mathbf{B} \in \mathbb{IR}^{n \times n}$. The product between $\mathbf{A}$ and $\mathbf{B}$ is also an interval matrix whose elements are defined as

$$\mathbf{C}_{i,j} = \sum_{k=1}^{n} \mathbf{A}_{i,k} * \mathbf{B}_{k,j}. \tag{2.1}$$

Using floating-point arithmetic to implement interval arithmetic, we have to deal with rounding errors. Let $\mathbf{A}, \mathbf{B} \in \mathbb{IF}^{n\times n}$, the product between $\mathbf{A}$ and $\mathbf{B}$ cannot be represented exactly by an interval matrix which is in $\mathbb{IF}^{n\times n}$. In fact, to compute the product between $\mathbf{A}$ and $\mathbf{B}$ we compute an interval matrix $\mathbf{C} \in \mathbb{IF}^{n\times n}$ which includes the exact product

$$\sum_{k=1}^{n} \mathbf{A}_{i,k} * \mathbf{B}_{k,j} \subseteq \mathbf{C}_{i,j}.$$

It should be noted here that a Strassen-like algorithm [vzGG03, chap.12], [Hig02, chap.23], [Str69, Hig90] in order to obtain a fast implementation of matrix multiplication, which costs $\mathcal{O}(n^\omega)$ floating-point operations with $\omega < 3$, cannot be applied to interval arithmetic [CK04], as such algorithms are based on identities such as $(a+b)*x+b*(-x+y) = a*x+b*y$ where $a, b, x, y \in \mathbb{R}$, which do not hold in interval arithmetic. For $\mathbf{a}, \mathbf{b}, \mathbf{x}, \mathbf{y} \in \mathbb{IR}$, we only have that $(\mathbf{a}+\mathbf{b}) * \mathbf{x} + \mathbf{b} * (-\mathbf{x}+\mathbf{y}) \supseteq \mathbf{a} * \mathbf{x} + \mathbf{b} * \mathbf{y}$, since the two expressions are mathematically equivalent meanwhile in the second expression each variable appears only once hence the second expression provides the optimal result of the expression, which implies the inclusion. For example, if $\mathbf{a} = [0,1], \mathbf{b} = [-1,1], \mathbf{x} = [0,1], \mathbf{y} = [-1,1]$ then, $\mathbf{a}*\mathbf{x}+\mathbf{b}*\mathbf{y} = [-1,2]$, meanwhile $(\mathbf{a}+\mathbf{b})*\mathbf{x}+\mathbf{b}*(-\mathbf{x}+\mathbf{y}) = [-3,4]$ is much larger than $\mathbf{a}*\mathbf{x}+\mathbf{b}*\mathbf{y}$. Therefore a recursive application of Strassen-like formula leads to a very fast increase of the width of result components.

Using natural algorithm (2.1), the computation of each result element requires n interval products and n interval additions. Moreover, given that interval arithmetic is implemented on computer using floating-point arithmetic and directed rounding mode as explained in Section 1.3.7, each interval addition requires two floating-point additions, meanwhile each interval product requires eight floating-point products. Hence, in total formula (2.1) requires $8n^3$ floating-point products and $2n^3$ floating-point additions.

The problem is that, using directed interval arithmetic for each interval operation, we have to switch between upward and downward rounding modes to compute the upper and lower bounds respectively. Moreover, the interval product uses `max` and `min` functions which are costly. Otherwise, we can test the sign of input intervals to reduce the number of floating-point products. But testing the sign is costly also. These problems make the natural algorithm difficult to be pipelined and to be parallelized. For instance, Rump's approach is faster than the BIAS approach [Knu06] which is based on the natural algorithm by a factor 20 to 100 [Rum99a].

To overcome this problem, Rump proposes in [Rum99a] an algorithm to compute efficiently an interval matrix product. His idea is to (i) perform as many floating-point operations as possible without having to change rounding mode, and to (ii) sacrifice a bit the result accuracy in order to improve significantly the performance.

In reality, the fastest algorithm to compute the multiplication of interval matrices is provided by Ogita et al [OO05]. This algorithm is also based on Rump's algorithm and relaxes the radius computation to gain speed: it is 2 times faster in comparison with Rump's algorithm. Nevertheless, the accuracy of the results computed by Ogita's algorithm is pessimistic with an overestimation factor up to 5. Therefore we consider here only Rump's algorithm. Note that the matrices in question here are non-degenerate. Algorithms to enclose a product of point matrices can be found in [Rum99a, OOO10].

Rump's algorithm is based on midpoint-radius representation of intervals: each interval is represented by its midpoint and its radius. Let $\mathbf{a} = \langle \hat{a}, \alpha \rangle$ and $\mathbf{b} = \left\langle \hat{b}, \beta \right\rangle$ be two intervals

represented in midpoint-radius format, with $\hat{a}, \hat{b}$ be the midpoints of $\mathbf{a}, \mathbf{b}$ respectively, and α, β be the radii of $\mathbf{a}, \mathbf{b}$ respectively. The basic operations are defined as follows:

$$\begin{cases} \mathbf{a} \oplus \mathbf{b} = \left\langle \hat{a} + \hat{b}, \alpha + \beta \right\rangle, \\ \mathbf{a} \odot \mathbf{b} = \left\langle \hat{a} * \hat{b}, |\hat{a}| * \beta + |\hat{b}| * \alpha + \alpha * \beta \right\rangle. \end{cases} \tag{2.2}$$

The isotonicity of (2.2) is clear. For all $a \in \mathbf{a}$ and $b \in \mathbf{b}$, we have:

$$|a - \hat{a}| \leq \alpha \quad \texttt{and} \quad |b - \hat{b}| \leq \beta.$$

Hence

$$\begin{array}{rll} & a * b - \hat{a} * \hat{b} & = \hat{a} * (b - \hat{b}) + (a - \hat{a}) * \hat{b} + (a - \hat{a}) * (b - \hat{b}) \\ \Rightarrow & |a * b - \hat{a} * \hat{b}| & \leq |\hat{a} * (b - \hat{b})| + |(a - \hat{a}) * \hat{b}| + |(a - \hat{a}) * (b - \hat{b})| \\ & & \leq |\hat{a}| * \beta + |\hat{b}| * \alpha + \alpha * \beta \\ \Rightarrow & a * b & \in \left\langle \hat{a} * \hat{b}, |\hat{a}| * \beta + |\hat{b}| * \alpha + \alpha * \beta \right\rangle \\ & & \in \mathbf{a} \odot \mathbf{b}. \end{array}$$

Regarding the exactness of operations, it can easily be verified that $\mathbf{a} \oplus \mathbf{b} = \mathbf{a} + \mathbf{b}$. For the case of multiplication, except for the case where either one of the two multipliers is punctual or is centered in zero, (2.2) always provides overestimated results. The overestimation factor of this formula is provided in Proposition 2.2.2 [Rum99a]:

Definition 2.2.1. *A real interval $\mathbf{a} = \langle a, \alpha \rangle$ not containing 0 is said to be of relative precision $e, 0 \leq e \in \mathbb{R}$, if*

$$\alpha \leq e * |a|.$$

A real interval containing 0 is said to be of relative precision 1.

Proposition 2.2.2. *Let $\mathbf{a} = \langle \hat{a}, \alpha \rangle \in \mathbb{IR}, \mathbf{b} = \left\langle \hat{b}, \beta \right\rangle \in \mathbb{IR}$, and denote the overestimation of $\mathbf{a} \odot \mathbf{b}$ as defined by (2.2) with respect to $\mathbf{a} * \mathbf{b}$ by*

$$\rho = \frac{\texttt{rad}(\mathbf{a} \odot \mathbf{b})}{\texttt{rad}(\mathbf{a} * \mathbf{b})},$$

where 0/0 is interpreted as 1. Then the following holds.

1. *It always holds that $\rho \leq 1.5$.*

2. *If one of the intervals $\mathbf{a}$ and $\mathbf{b}$ is of relative precision e, then*

$$\rho \leq 1 + \frac{e}{1 + e}.$$

3. *If interval $\mathbf{a}$ is of relative precision e, and $\mathbf{b}$ is of relative precision f, then*

$$\rho \leq 1 + \frac{ef}{1 + e}.$$

All estimations are sharp.

Matrix operations can be defined in the same way as scalar operations. Suppose that each interval matrix is represented by two punctual matrices, one representing the midpoints and the other representing the radius. Let $\mathbf{A} = \langle A, \Lambda \rangle \in \mathbb{IR}^{n \times n}$, $\mathbf{B} = \langle B, \Sigma \rangle \in \mathbb{IR}^{n \times n}$

$$\begin{cases} \mathbf{A} \oplus \mathbf{B} = \langle A + B, \Lambda + \Sigma \rangle, \\ \mathbf{A} \odot \mathbf{B} = \langle A * B, |A| * \Sigma + |B| * \Lambda + \Lambda * \Sigma \rangle. \end{cases} \tag{2.3}$$

The main advantage of Rump's algorithm is that using formula (2.3), an interval matrix product can be computed by performing merely point matrix products. It means that the interval matrix product as defined by formula (2.3) can be implemented using existing optimized libraries for floating-point matrix operations, and can benefit from pipelined or parallelized implementations.

Nonetheless, care must be taken when implementing (2.3) in finite precision. Indeed, in the presence of rounding errors, one cannot compute exactly the product of two floating-point matrices. Therefore, firstly, we have to enclose the midpoint of the result by computing it following (2.3) twice, in downward and upward rounding modes. Secondly, to guarantee the property of isotonicity, we must compute an upper bound of the radius. According to (2.3), the expression to compute the radius involves only products of non-negative matrices. Thus an upper bound of the radius can be obtained by simply evaluating in upward rounding mode. Additionally, (2.3) can be rewritten as $\mathbf{A} \odot \mathbf{B} = \langle A * B, (|A| + \Lambda) * \Sigma + |B| * \Lambda \rangle$. Therefore, in total, the implementation of Rump's algorithm in finite precision to compute an interval matrix product requires only 4 floating-point matrix products together with a few rounding mode changes.

In regard to accuracy, the addition $\mathbf{A} \oplus \mathbf{B}$ is always exactly equal to $\mathbf{A} + \mathbf{B}$ in infinite precision. For the case of multiplication:

$$\begin{aligned}
\mathbf{A} * \mathbf{B} &= \left\{ \sum_{k=1}^{n} \mathbf{A}_{i,k} * \mathbf{B}_{k,j}, i = 1:n, j = 1:n \right\} \\
&\subseteq \left\{ \sum_{k=1}^{n} \mathbf{A}_{i,k} \odot \mathbf{B}_{k,j}, i = 1:n, j = 1:n \right\} \\
&\subseteq \left\{ \sum_{k=1}^{n} \langle A_{i,k} * B_{k,j}, |A_{i,k}| * \Sigma_{k,j} + \Lambda_{i,k} * |B_{k,j}| + \Lambda_{i,k} * \Sigma_{k,j} \rangle, i = 1:n, j = 1:n \right\} \\
&\subseteq \left\{ \left\langle \sum_{k=1}^{n} A_{i,k} * B_{k,j}, \sum_{k=1}^{n} |A_{i,k}| * \Sigma_{k,j} + \Lambda_{i,k} * |B_{k,j}| + \Lambda_{i,k} * \Sigma_{k,j} \right\rangle \right\} \\
&\subseteq \langle A * B, |A| * \Sigma + |B| * \Lambda + \Lambda * \Sigma \rangle \\
&\subseteq \mathbf{A} \odot \mathbf{B}.
\end{aligned}$$

The overestimation factor of the interval matrix product defined by (2.3) is also given in [Rum99a]:

Proposition 2.2.3. *For $\mathbf{A}$ an interval matrix with all components of relative precision e, and for $\mathbf{B}$ an interval matrix with all components of relative precision f, the overestimation of each component of the result of multiplication is bounded by*

$$1 + \frac{ef}{e+f}.$$

In any case, the overestimation is uniformly bounded by 1.5.

The overestimation factor of 1.5 can be achieved, as in the following example.

Example 3. Let's compute the following interval matrix product by both the natural algorithm and Rump's algorithm:

$$
\begin{aligned}
\begin{pmatrix} [0,4] & [0,2] \\ [0,2] & [0,4] \end{pmatrix} * \begin{pmatrix} [0,2] & [0,2] \\ [0,2] & [0,2] \end{pmatrix} &= \begin{pmatrix} [0,12] & [0,12] \\ [0,12] & [0,12] \end{pmatrix}. \\
\begin{pmatrix} [0,4] & [0,2] \\ [0,2] & [0,4] \end{pmatrix} \odot \begin{pmatrix} [0,2] & [0,2] \\ [0,2] & [0,2] \end{pmatrix} &= \begin{pmatrix} \langle 2,2 \rangle & \langle 1,1 \rangle \\ \langle 1,1 \rangle & \langle 2,2 \rangle \end{pmatrix} \odot \begin{pmatrix} \langle 1,1 \rangle & \langle 1,1 \rangle \\ \langle 1,1 \rangle & \langle 1,1 \rangle \end{pmatrix} \\
&= \begin{pmatrix} \langle 3,9 \rangle & \langle 3,9 \rangle \\ \langle 3,9 \rangle & \langle 3,9 \rangle \end{pmatrix} \\
&= \begin{pmatrix} [-6,12] & [-6,12] \\ [-6,12] & [-6,12] \end{pmatrix}.
\end{aligned}
$$

The overestimation factor of each component is $\rho = 18/12 = 1.5$.

In practice, the factor between results computed by (2.3) and those computed by (2.1) is interestingly much smaller than this bound, and in many cases close to 1. In some cases when the majority of input elements are small, i.e their radius is small with respect to their midpoint, the matrix product computed by (2.3) is even sharper than that computed by (2.1).

On the other hand, if all components of $\mathbf{A}$ and $\mathbf{B}$ are of large relative precision $e \approx 1$, then the overestimation of each component of the result computed by (2.3) is close to 1.5. This might not be acceptable in some applications, for example in global optimization. In the following sections we will introduce new algorithms to reduce this overestimation factor while not deteriorating significantly the performance.

2.3 Algorithm 1 using endpoints representation

In this section, we assume that each interval matrix is represented by two real matrices, one representing the lower bound and the other representing the upper bound. First, we study some special cases.

2.3.1 Special cases

If some information about the input matrices is known in advance, then we can efficiently compute the multiplication using the natural algorithm (2.1) without any overestimation.

Centered-in-zero multiplier

Let $\mathbf{a} = [\underline{a}, \overline{a}] \in \mathbb{IR}$ be a centered-in-zero interval, meaning that $\underline{a} = -\overline{a}$ and $\overline{a} > 0$. Let $\mathbf{b} = [\underline{b}, \overline{b}]$ be an interval, then the product between $\mathbf{a}$ and $\mathbf{b}$ can be computed by

$$
\begin{aligned}
\mathbf{a} * \mathbf{b} &= \left[\underline{a} * \mathtt{max}(|\underline{b}|, |\overline{b}|), \overline{a} * \mathtt{max}(|\underline{b}|, |\overline{b}|)\right] \\
&= \left[-\overline{a} * \mathtt{mag}(\mathbf{b}), \overline{a} * \mathtt{mag}(\mathbf{b})\right].
\end{aligned}
$$

Note that this formula is still valid in the presence of rounding errors, with $\overline{a} * \texttt{mag}(\mathbf{b})$ being computed in upward rounding mode. Indeed, suppose that $c = \triangle(\overline{a} * \texttt{mag}(\mathbf{b}))$ then $c \geq \overline{a} * \texttt{mag}(\mathbf{b})$, which implies $-c \leq -\overline{a} * \texttt{mag}(\mathbf{b}) \leq \overline{a} * \texttt{mag}(\mathbf{b}) \leq c$. Hence $\mathbf{a} * \mathbf{b} \subseteq [-c, c]$.

Due to the symmetry, the interval multiplication can be computed by one punctual multiplication for the upper bound, followed by one negation to compute the lower bound. This also holds for a matrix product.

Let $\mathbf{A} = \left[\underline{A}, \overline{A}\right] \in \mathbb{IR}^{n \times n}$ be a centered-in-zero interval matrix, i.e. $\underline{A} = -\overline{A}$ and $\overline{A} > 0$. Let $\mathbf{B} = \left[\underline{B}, \overline{B}\right]$ be an interval matrix, then the product between $\mathbf{A}$ and $\mathbf{B}$ can be computed by

$$\mathbf{A} * \mathbf{B} = \left[-\overline{A} * \texttt{mag}(\mathbf{B}), \overline{A} * \texttt{mag}(\mathbf{B})\right]. \tag{2.4}$$

The implementation in floating-point arithmetic simply sets the rounding mode to upward before computing $\overline{A} * \texttt{mag}(\mathbf{B})$.

The lower bound is opposite to the upper bound, hence the product $\mathbf{A} * \mathbf{B}$ requires only one punctual matrix product.

Non-negative multiplier

Let $\mathbf{a} = [\underline{a}, \overline{a}] \in \mathbb{IR}$ be an interval where $\overline{a} \geq \underline{a} \geq 0$. Let $\mathbf{b} = \left[\underline{b}, \overline{b}\right] \in \mathbb{IR}$ be an interval, then

$$\begin{array}{rclrcl} \texttt{max}(\overline{a} * \underline{b}, \overline{a} * \overline{b}) & = & \overline{a} * \overline{b}, & \texttt{max}(\underline{a} * \underline{b}, \underline{a} * \overline{b}) & = & \underline{a} * \overline{b}, \\ \texttt{min}(\overline{a} * \underline{b}, \overline{a} * \overline{b}) & = & \overline{a} * \underline{b}, & \texttt{min}(\underline{a} * \underline{b}, \underline{a} * \overline{b}) & = & \underline{a} * \underline{b}. \end{array}$$

Hence

$$\begin{array}{rcl} \texttt{sup}(\mathbf{a} * \mathbf{b}) & = & \texttt{max}(\overline{a} * \overline{b}, \underline{a} * \overline{b}), \\ \texttt{inf}(\mathbf{a} * \mathbf{b}) & = & \texttt{min}(\overline{a} * \underline{b}, \underline{a} * \underline{b}). \end{array}$$

Denote by $x^+ = \begin{cases} x & \text{if} \quad x \geq 0 \\ 0 & \text{if} \quad x < 0 \end{cases}$ and $x^- = \begin{cases} 0 & \text{if} \quad x \geq 0 \\ x & \text{if} \quad x < 0 \end{cases}$.

It is easy to verify that

$$\begin{array}{rcl} \texttt{max}(\overline{a} * \overline{b}, \underline{a} * \overline{b}) & = & \overline{a} * \overline{b}^+ + \underline{a} * \overline{b}^- \\ \texttt{min}(\overline{a} * \underline{b}, \underline{a} * \underline{b}) & = & \overline{a} * \underline{b}^- + \underline{a} * \underline{b}^+ \\ \Rightarrow \mathbf{a} * \mathbf{b} & = & \left[\overline{a} * \underline{b}^- + \underline{a} * \underline{b}^+, \overline{a} * \overline{b}^+ + \underline{a} * \overline{b}^-\right]. \end{array}$$

By induction on the dimension of the matrices, we can generalize this formula to matrix multiplication.

Proposition 2.3.1. *For* $\mathbf{A} = \left[\underline{A}, \overline{A}\right] \in \mathbb{IR}^{n \times n}$*, with* $\mathbf{A}_{i,j} \geq 0$ *for all* $1 \leq i \leq n, 1 \leq j \leq n$*, and* $\mathbf{B} = \left[\underline{B}, \overline{B}\right] \in \mathbb{IR}^{n \times n}$*, the multiplication is computed by*

$$\mathbf{A} * \mathbf{B} = \left[\overline{A} * \underline{B}^- + \underline{A} * \underline{B}^+, \overline{A} * \overline{B}^+ + \underline{A} * \overline{B}^-\right]. \tag{2.5}$$

Hence, in this case, an interval matrix product requires four punctual matrix products.

Similarly, when $\mathbf{A}$ is non-positive, the multiplication is also computed by four punctual matrix products:

$$\mathbf{A} * \mathbf{B} = \left[\overline{A} * \overline{B}^- + \underline{A} * \overline{B}^+, \overline{A} * \underline{B}^+ + \underline{A} * \underline{B}^-\right]. \tag{2.6}$$

Not containing zero multiplier

More generally, if one multiplier does not contain zero then we can combine the two previous subcases.

For $\mathbf{a} \in \mathbb{IR}$, denote by

$$\begin{aligned} \mathbf{a}^+ &= \{a \in \mathbf{a} \mid a \geq 0\}, \\ \mathbf{a}^- &= \{a \in \mathbf{a} \mid a \leq 0\} \end{aligned}$$

where the empty interval is replaced by 0.

Proposition 2.3.2. *If* $\mathbf{a}$ *does not contain zero then for any* $\mathbf{b} \in \mathbb{IR}$

$$\mathbf{a} * \mathbf{b} = \mathbf{a}^+ * \mathbf{b} + \mathbf{a}^- * \mathbf{b}.$$

Proof. $\mathbf{a}$ does not contain zero, hence

- if $\mathbf{a} \geq 0$ then $\mathbf{a}^+ = \mathbf{a}$ and $\mathbf{a}^- = 0$, which gives $\mathbf{a}^+ * \mathbf{b} + \mathbf{a}^- * \mathbf{b} = \mathbf{a} * \mathbf{b} + 0 * \mathbf{b} = \mathbf{a} * \mathbf{b}$;
- if $\mathbf{a} \leq 0$ then $\mathbf{a}^+ = 0$ and $\mathbf{a}^- = \mathbf{a}$, which gives $\mathbf{a}^+ * \mathbf{b} + \mathbf{a}^- * \mathbf{b} = 0 * \mathbf{b} + \mathbf{a} * \mathbf{b} = \mathbf{a} * \mathbf{b}$.

□

Proposition 2.3.2 means that the computation of the interval multiplication in this case requires eight punctual multiplications.

We can generalize Proposition 2.3.2 to matrix multiplication by induction. One matrix does not contain zero if each element of this matrix does not contain zero.

Proposition 2.3.3. *For* $\mathbf{A} \in \mathbb{IR}^{n \times n}$ *not containing zero, and* $\mathbf{B} \in \mathbb{IR}^{n \times n}$*, the multiplication* $\mathbf{A} * \mathbf{B}$ *can be computed by eight punctual matrix multiplications:*

$$\begin{aligned} \mathbf{A} * \mathbf{B} =& \mathbf{A}^+ * \mathbf{B} + \mathbf{A}^- * \mathbf{B} \\ =& \left[\overline{\mathbf{A}^+} * \underline{B}^- + \underline{\mathbf{A}^+} * \underline{B}^+, \overline{\mathbf{A}^+} * \overline{B}^+ + \underline{\mathbf{A}^+} * \overline{B}^-\right] \\ &+ \left[\overline{\mathbf{A}^-} * \overline{B}^- + \underline{\mathbf{A}^-} * \overline{B}^+, \overline{\mathbf{A}^-} * \underline{B}^+ + \underline{\mathbf{A}^-} * \underline{B}^-\right] \\ =& \left[\overline{\mathbf{A}^+} * \underline{B}^- + \underline{\mathbf{A}^+} * \underline{B}^+ + \overline{\mathbf{A}^-} * \overline{B}^- + \underline{\mathbf{A}^-} * \overline{B}^+,\right. \\ & \left.\overline{\mathbf{A}^+} * \overline{B}^+ + \underline{\mathbf{A}^+} * \overline{B}^- + \overline{\mathbf{A}^-} * \underline{B}^+ + \underline{\mathbf{A}^-} * \underline{B}^-\right]. \end{aligned}$$

Note that all the formulae for these special cases provide results without overestimation.

2.3.2 General case

When no information about the input matrices is known in advance, then to compute exactly the matrix multiplication, it is mandatory to use the natural algorithm (2.1), which is not efficient. Rump's algorithm can be seen as decomposing input matrices into two parts, one punctual part and one centered-in-zero part, then computing sub-products which can be computed exactly and efficiently, and finally accumulating all the sub-products. Even though each sub-product can be computed exactly, the final computed result is an enclosure of the exact result because of the sub-distributivity property of interval arithmetic.

Let $\mathbf{A} = \langle A, \Lambda \rangle \in \mathbb{IR}^{n\times n}$ and $\mathbf{B} = \langle B, \Sigma \rangle \in \mathbb{IR}^{n\times n}$, we have that

$$\begin{aligned}
\mathbf{A} &= A + \langle 0, \Lambda \rangle \\
\mathbf{B} &= B + \langle 0, \Sigma \rangle \\
\Rightarrow \mathbf{A} * \mathbf{B} &= (A + \langle 0, \Lambda \rangle) * (B + \langle 0, \Sigma \rangle) \\
&\subseteq A * B + A * \langle 0, \Sigma \rangle + \langle 0, \Lambda \rangle * B + \langle 0, \Lambda \rangle * \langle 0, \Sigma \rangle \\
&\subseteq \langle A * B, 0 \rangle + \langle 0, |A| * \Sigma \rangle + \langle 0, \Lambda * |B| \rangle + \langle 0, \Lambda * \Sigma \rangle \\
&\subseteq \langle A * B, |A| * \Sigma + \Lambda * |B| + \Lambda * \Sigma \rangle .
\end{aligned}$$

With this decomposition, the overestimation in the worst case is 1.5. Our idea is to find another decomposition such that:

- sub-products can be efficiently computed, and
- the overestimation is small.

As we can see in Section 2.3.1, an interval product can be efficiently computed when one of the two multipliers is centered in zero or does not contain zero. This leads to our following proposition of decomposition.

Proposition 2.3.4. *Let* $\mathbf{A}$ *be an interval matrix. If* $\mathbf{A}$ *is decomposed into two interval matrices* $\mathbf{A}^0$ *and* $\mathbf{A}^*$ *which satisfy:*

$$\begin{cases}
\text{if } 0 \le \underline{A}_{i,j} \le \overline{A}_{i,j} & \text{then } \mathbf{A}^0_{i,j} = 0, & \mathbf{A}^*_{i,j} = \mathbf{A}_{i,j} \\
\text{if } \underline{A}_{i,j} \le \overline{A}_{i,j} \le 0 & \text{then } \mathbf{A}^0_{i,j} = 0, & \mathbf{A}^*_{i,j} = \mathbf{A}_{i,j} \\
\text{if } \underline{A}_{i,j} < 0 < |\underline{A}_{i,j}| \le \overline{A}_{i,j} & \text{then } \mathbf{A}^0_{i,j} = \left[\underline{A}_{i,j}, -\underline{A}_{i,j}\right], & \mathbf{A}^*_{i,j} = \left[0, \underline{A}_{i,j} + \overline{A}_{i,j}\right] \\
\text{if } \underline{A}_{i,j} < 0 < \overline{A}_{i,j} < |\underline{A}_{i,j}| & \text{then } \mathbf{A}^0_{i,j} = \left[-\overline{A}_{i,j}, \overline{A}_{i,j}\right], & \mathbf{A}^*_{i,j} = \left[\underline{A}_{i,j} + \overline{A}_{i,j}, 0\right]
\end{cases} \tag{2.7}$$

then

(1) $\mathbf{A}^0$ *is centered in zero,*

(2) $\mathbf{A}^*$ *does not contain zero,*

(3) $\mathbf{A}^0 + \mathbf{A}^* = \mathbf{A}$*, and*

(4) $\mathtt{mag}(\mathbf{A}^0) + \mathtt{mag}(\mathbf{A}^*) = \mathtt{mag}(\mathbf{A})$.

Proof. Easily deduced from the formulae of decomposition. □

Proposition 2.3.5. *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *two interval matrices and* $(\mathbf{A}^0, \mathbf{A}^*)$ *a decomposition of* $\mathbf{A}$ *by Proposition 2.3.4. Let* $\mathbf{C}$ *the interval matrix being computed by*

$$\mathbf{C} = \mathbf{A}^0 * \mathbf{B} + \mathbf{A}^* * \mathbf{B}, \tag{2.8}$$

then $\mathbf{A} * \mathbf{B}$ *is contained in* $\mathbf{C}$*. Denote* $\mathbf{A} \circledast \mathbf{B} = \mathbf{C}$.

Proof. Following Proposition 2.3.4: $\mathbf{A} = \mathbf{A}^0 + \mathbf{A}^*$. Interval multiplication is sub-distributive, hence

$$\mathbf{A} * \mathbf{B} = (\mathbf{A}^0 + \mathbf{A}^*) * \mathbf{B} \subseteq \mathbf{A}^0 * \mathbf{B} + \mathbf{A}^* * \mathbf{B} = \mathbf{C}.$$

□

Moreover, $\mathbf{A}^0$ is centered in zero and $\mathbf{A}^*$ does not contain zero, according to Section 2.3.1, $\mathbf{A}^0 * \mathbf{B}$ and $\mathbf{A}^* * \mathbf{B}$ can be computed exactly using respectively one and eight punctual matrix products. Hence, it requires in total nine punctual matrix products to compute an enclosure of $\mathbf{A} * \mathbf{B}$ by Proposition 2.3.5. Nevertheless, because of the sub-distributivity, the result computed by this proposition is always an overestimation of the exact result.

The following theorem provides the overestimation factor of this algorithm in case of scalar inputs.

Theorem 2.3.6. *For $\mathbf{a} \in \mathbb{IR}, \mathbf{b} \in \mathbb{IR}$, with $\mathbf{a}$ being decomposed, following Proposition 2.3.4, into two parts: $\mathbf{a} = \mathbf{a}^0 + \mathbf{a}^*$, the overestimation of $\mathbf{a} \circledast \mathbf{b} = \mathbf{a}^0 * \mathbf{b} + \mathbf{a}^* * \mathbf{b}$ with respect to $\mathbf{a} * \mathbf{b}$ is defined as*

$$\rho^1 = \frac{\mathtt{diam}(\mathbf{a} \circledast \mathbf{b})}{\mathtt{diam}(\mathbf{a} * \mathbf{b})}.$$

The following holds:

(1) ρ^1 is always bounded by $4 - 2\sqrt{2}$;

(2) if either $\mathbf{a}$ or $\mathbf{b}$ is centered in zero then $\rho^1 = 1$;

(3) if either $\mathbf{a}$ or $\mathbf{b}$ does not contain zero then $\rho^1 = 1$.

Proof. We will first prove the property (2).

If a is centered in zero then $\mathbf{a}^* = 0 \rightarrow \mathbf{a} * \mathbf{b} = \mathbf{a}^0 * \mathbf{b}$. Thus the result is exact, i.e. $\rho = 1$.

If b is centered in zero then

$$\begin{aligned} \mathbf{a} * \mathbf{b} &= \left[-\mathtt{mag}\left(\mathbf{a}\right) * \overline{b}, \mathtt{mag}\left(\mathbf{a}\right) * \overline{b}\right] \\ \mathbf{a}^0 * \mathbf{b} + \mathbf{a}^* * \mathbf{b} &= \left[-\mathtt{mag}(\mathbf{a}^0) * \overline{b}, \mathtt{mag}(\mathbf{a}^0) * \overline{b}\right] + \left[-\mathtt{mag}(\mathbf{a}^*) * \overline{b}, \mathtt{mag}(\mathbf{a}^*) * \overline{b}\right] \\ &= \left[-(\mathtt{mag}(\mathbf{a}^0) + \mathtt{mag}(\mathbf{a}^*)) * \overline{b}, (\mathtt{mag}(\mathbf{a}^0) + \mathtt{mag}(\mathbf{a}^*)) * \overline{b}\right]. \end{aligned}$$

Moreover, following Proposition 2.3.4

$$\mathtt{mag}(\mathbf{a}^0) + \mathtt{mag}(\mathbf{a}^*) = \mathtt{mag}\left(\mathbf{a}\right),$$

which implies that $\mathbf{a} \circledast \mathbf{b} = \mathbf{a} * \mathbf{b}$, or $\rho^1 = 1$. Hence, property (2) is true.

Properties (1) and (3) can be proven by inspecting subcases related to the values of $\mathbf{a}$ and $\mathbf{b}$. There are two cases for the value of $\mathbf{a}$, which are "$\mathbf{a}$ does not contain zero" and "$\mathbf{a}$ contains zero".

If a does not contain zero then $\mathbf{a}^0 = 0 \Rightarrow \mathbf{a} * \mathbf{b} = \mathbf{a}^* * \mathbf{b} \Rightarrow \rho^1 = 1$.

If a contains zero. It means that $\underline{a} < 0 < \overline{a}$. The case $\overline{a} < |\underline{a}|$ is similar to the case $\overline{a} > |\underline{a}|$, hence without loss of generality suppose that $\overline{a} > |\underline{a}|$. Following Proposition 2.3.4, $\mathbf{a}^0 = [\underline{a}, -\underline{a}]$ and $\mathbf{a}^* = [0, \overline{a} + \underline{a}]$. Hence, following Proposition 2.3.5

$$\begin{aligned} \mathbf{a} \circledast \mathbf{b} &= \mathbf{a}^0 * \mathbf{b} + \mathbf{a}^* * \mathbf{b} \\ &= [\underline{a}, -\underline{a}] * \mathbf{b} + [0, \overline{a} + \underline{a}] * \mathbf{b} \\ &= [\underline{a} * \mathtt{mag}\left(\mathbf{b}\right), -\underline{a} * \mathtt{mag}\left(\mathbf{b}\right)] + [0, \overline{a} + \underline{a}] * \mathbf{b}. \end{aligned}$$

There are four subcases for the value of $\mathbf{b}$, which are $\mathbf{b} \geq 0$, $\mathbf{b} \leq 0$, $\overline{b} > |\underline{b}| > 0 > \underline{b}$, and $|\underline{b}| > \overline{b} > 0 > \underline{b}$.

1. Case $\mathbf{b} \geq 0$

$$\begin{aligned} \mathtt{mag}(\mathbf{b}) &= \overline{b} \\ \Rightarrow \mathbf{a} \circledast \mathbf{b} &= [\underline{a} * \mathtt{mag}(\mathbf{b}), -\underline{a} * \mathtt{mag}(\mathbf{b})] + [0, \overline{a} + \underline{a}] * \mathbf{b} \\ &= \left[\underline{a} * \overline{b}, -\underline{a} * \overline{b}\right] + \left[0, (\overline{a} + \underline{a}) * \overline{b}\right] \\ &= \left[\underline{a} * \overline{b}, \overline{a} * \overline{b}\right]. \end{aligned}$$

Meanwhile, $\mathbf{b} \geq 0 \to \mathbf{a} * \mathbf{b} = \left[\underline{a} * \overline{b}, \overline{a} * \overline{b}\right]$. Hence $\mathbf{a} \circledast \mathbf{b} = \mathbf{a} * \mathbf{b}$. It means that there is no overestimation in this case, i.e $\rho^1 = 1$.

2. Case $\mathbf{b} \leq 0$: this case is similar to the case $\mathbf{b} \geq 0$. It means that if $\mathbf{a}$ contains zero but $\mathbf{b}$ does not contain zero then $\rho^1 = 1$. Hence property (3) is true.

3. Case $\overline{b} > 0 > \underline{b}$ and $\overline{b} \geq |\underline{b}| \to \mathtt{mag}(\mathbf{b}) = \overline{b}$ and $(\overline{a} + \underline{a}) * \underline{b} < 0 < (\overline{a} + \underline{a}) * \overline{b}$. Hence

$$\begin{aligned} \mathbf{a} \circledast \mathbf{b} &= [\underline{a} * \mathtt{mag}(\mathbf{b}), -\underline{a} * \mathtt{mag}(\mathbf{b})] + [0, \overline{a} + \underline{a}] * \mathbf{b} \\ &= \left[\underline{a} * \overline{b}, -\underline{a} * \overline{b}\right] + \left[(\overline{a} + \underline{a}) * \underline{b}, (\overline{a} + \underline{a}) * \overline{b}\right] \\ &= \left[(\overline{a} + \underline{a}) * \underline{b} + \underline{a} * \overline{b}, \overline{a} * \overline{b}\right] \\ \Rightarrow \mathtt{diam}(\mathbf{a} \circledast \mathbf{b}) &= \overline{a} * \overline{b} - ((\overline{a} + \underline{a}) * \underline{b} + \underline{a} * \overline{b}) \\ &= \overline{a} * \overline{b} * (1 - \underline{a}/\overline{a} - \underline{b}/\overline{b} - \underline{a}/\overline{a} * \underline{b}/\overline{b}). \end{aligned}$$

As $\overline{a} > 0 > \underline{a} > -\overline{a}$ and $\overline{b} > 0 > \underline{b} > -\overline{b}$

$$\begin{aligned} &\overline{a} * \underline{b} < 0 \\ &\underline{a} * \overline{b} < 0 < \underline{a} * \underline{b} < \overline{a} * \overline{b} \\ \Rightarrow \mathbf{a} * \mathbf{b} &= \left[\mathtt{min}(\overline{a} * \underline{b}, \underline{a} * \overline{b}), \overline{a} * \overline{b}\right] \\ \Rightarrow \mathtt{diam}(\mathbf{a} * \mathbf{b}) &= \overline{a} * \overline{b} - \mathtt{min}(\overline{a} * \underline{b}, \underline{a} * \overline{b}) \\ &= \overline{a} * \overline{b} * (1 - \mathtt{min}(\underline{a}/\overline{a}, \underline{b}/\overline{b})). \end{aligned}$$

Denote by $M = \mathtt{max}(|\underline{a}|/\overline{a}, |\underline{b}|/\overline{b})$ and $m = \mathtt{min}(|\underline{a}|/\overline{a}, |\underline{b}|/\overline{b})$ then

$$\begin{cases} 0 < m \leq M \leq 1, & \\ \mathtt{min}(\underline{a}/\overline{a}, \underline{b}/\overline{b}) & = -M, \\ \underline{a}/\overline{a} + \underline{b}/\overline{b} & = -m - M, \\ \underline{a}/\overline{a} * \underline{b}/\overline{b} & = m * M. \end{cases} \tag{2.9}$$

The factor of overestimation is computed by:

$$\begin{aligned} \rho^1 &= \frac{\mathtt{diam}(\mathbf{a} \circledast \mathbf{b})}{\mathtt{diam}(\mathbf{a} * \mathbf{b})} \\ \rho^1 &= \frac{1 - \underline{a}/\overline{a} - \underline{b}/\overline{b} - \underline{a}/\overline{a} * \underline{b}/\overline{b}}{1 - \mathtt{min}(\underline{a}/\overline{a}, \underline{b}/\overline{b})} \\ \rho^1 &= \frac{1 + M + m - Mm}{1 + M} \\ \rho^1 &= 1 + \frac{m(1 - M)}{1 + M} \\ \rho^1 &\leq 1 + \frac{M(1 - M)}{1 + M} \equiv f(M). \end{aligned}$$

By studying the variations of the function $f(M)$, with M varying between 0 and 1, we obtain that $f(M) \leq 4 - 2\sqrt{2}$. Equality holds when $M = \sqrt{2} - 1$.

Hence, $\rho^1 \leq 4 - 2\sqrt{2}$. Equality holds when $\underline{a}/\overline{a} = \underline{b}/\overline{b} = 1 - \sqrt{2}$.

4. Case $\overline{b} > 0 > \underline{b}$ and $\overline{b} < |\underline{b}|$: this case is similar to the case $\overline{b} > 0 > \underline{b}$ and $\overline{b} \geq |\underline{b}|$. By denoting by $M = \mathtt{max}(|\underline{a}|/\overline{a}, \overline{b}/|\underline{b}|)$ and $m = \mathtt{min}(|\underline{a}|/\overline{a}, \overline{b}/|\underline{b}|)$, the factor of overestimation in this case is

$$\rho^1 = 1 + \frac{m(1-M)}{1+M},$$

which is also bounded by $\rho^1 \leq 4 - 2\sqrt{2}$. Equality holds when $\underline{a}/\overline{a} = \overline{b}/\underline{b} = 1 - \sqrt{2}$.

□

We can also generalize this theorem to the case of matrix multiplication.

Theorem 2.3.7. *For $\mathbf{A} \in \mathbb{IR}^{n\times n}, \mathbf{B} \in \mathbb{IR}^{n\times n}$, the overestimation factor of each component of the multiplication defined by Proposition 2.3.5 is bounded by*

$$\rho \leq 4 - 2\sqrt{2}.$$

Moreover, if all the components of either $\mathbf{A}$ or $\mathbf{B}$ do not contain zero then all the computed components are exacts.

Proof. By induction from Theorem 2.3.6 and from the fact that

$$\mathtt{diam}(\mathbf{a} + \mathbf{b}) = \mathtt{diam}(\mathbf{a}) + \mathtt{diam}(\mathbf{b}).$$

□

Hence, it is clear that the overestimation in the worst case of our algorithm, which is $4 - 2\sqrt{2} \approx 1.18$, is smaller than the overestimation in the worst case of Rump's algorithm, which is 1.5. For each individual pair of inputs, the following proposition gives a rough comparison between the two algorithms.

Proposition 2.3.8. *For $\mathbf{a} = [\underline{a}, \overline{a}] = \langle a, \alpha\rangle \in \mathbb{IR}$, $\mathbf{b} = \left[\underline{b}, \overline{b}\right] = \langle b, \beta\rangle \in \mathbb{IR}$*

$$\mathbf{a} \circledast \mathbf{b} \subseteq \mathbf{a} \odot \mathbf{b}.$$

Equality holds when either one of the two inputs is punctual or centered in zero.

Proof. We distinguish between two cases for the values of $\mathbf{a}$ and $\mathbf{b}$:

1. Either $\mathbf{a}$ or $\mathbf{b}$ does not contain zero.

   In this case, following Theorem 2.3.6, the overestimation of $\mathbf{a} \circledast \mathbf{b}$ is $\rho^1 = 1$, or in other words there is no overestimation. Hence, it is obvious that

$$\mathbf{a} \circledast \mathbf{b} \subseteq \mathbf{a} \odot \mathbf{b}.$$

2. Both $\mathbf{a}$ and $\mathbf{b}$ contain zero. There are four possible sub-cases where:

(a) $\overline{a} \geq |\underline{a}| > 0 > \underline{a}$ and $\overline{b} \geq |\underline{b}| > 0 > \underline{b}$,

(b) $\overline{a} \geq |\underline{a}| > 0 > \underline{a}$ and $\underline{b} < 0 < \overline{b} \leq |\underline{b}|$,

(c) $\underline{a} < 0 < \overline{a} \leq |\underline{a}|$ and $\overline{b} \geq |\underline{b}| > 0 > \underline{b}$,

(d) $\underline{a} < 0 < \overline{a} \leq |\underline{a}|$ and $\underline{b} < 0 < \overline{b} \leq |\underline{b}|$.

We will consider here only the first sub-case. The proof of the other 3 sub-cases is similar.

$[\underline{a}, \overline{a}]$ and $\langle a, \alpha \rangle$ represent the same interval, i.e.

$$\left\{ \begin{array}{lcl} \underline{a} & = & a - \alpha \\ \overline{a} & = & a + \alpha \end{array} \right. \quad \text{and} \quad \left\{ \begin{array}{lcl} a & = & 1/2 * (\underline{a} + \overline{a}) \\ \alpha & = & 1/2 * (\overline{a} - \underline{a}). \end{array} \right.$$

Similarly we have

$$\left\{ \begin{array}{lcl} \underline{b} & = & b - \beta \\ \overline{b} & = & b + \beta \end{array} \right. \quad \text{and} \quad \left\{ \begin{array}{lcl} b & = & 1/2 * (\underline{b} + \overline{b}) \\ \beta & = & 1/2 * (\overline{b} - \underline{b}). \end{array} \right.$$

We have $\overline{a} \geq |\underline{a}| > 0 > \underline{a}, \overline{b} \geq |\underline{b}| > 0 > \underline{b}$ hence $\alpha > a > 0$ and $\beta > b > 0$.

$\mathbf{a} \odot \mathbf{b}$ as defined by (2.2) is

$$\begin{array}{rcl} \texttt{mid}(\mathbf{a} \odot \mathbf{b}) & = & a * b, \\ \texttt{rad}(\mathbf{a} \odot \mathbf{b}) & = & |a| * \beta + \alpha * |b| + \alpha * \beta \\ & = & a * \beta + \alpha * b + \alpha * \beta. \end{array}$$

As mentioned in the proof of Theorem 2.3.6, when $\overline{a} \geq |\underline{a}| > 0 > \underline{a}$ and $\overline{b} \geq |\underline{b}| > 0 > \underline{b}$, $\mathbf{a} \circledast \mathbf{b}$ is

$$\begin{array}{rcl} \mathbf{a} \circledast \mathbf{b} & = & \left[(\overline{a} + \underline{a}) * \underline{b} + \underline{a} * \overline{b}, \overline{a} * \overline{b}\right] \\ & = & [2 * a * (b - \beta) + (a - \alpha) * (b + \beta), (a + \alpha) * (b + \beta)] \\ & = & [3 * a * b - a * \beta - \alpha * \beta - \alpha * \beta, a * b + a * \beta + \alpha * b + \alpha * \beta] \\ & = & [2 * a * b + \texttt{mid}(\mathbf{a} \odot \mathbf{b}) - \texttt{rad}(\mathbf{a} \odot \mathbf{b}), \texttt{mid}(\mathbf{a} \odot \mathbf{b}) + \texttt{rad}(\mathbf{a} \odot \mathbf{b})] \\ & \subseteq & [\texttt{mid}(\mathbf{a} \odot \mathbf{b}) - \texttt{rad}(\mathbf{a} \odot \mathbf{b}), \texttt{mid}(\mathbf{a} \odot \mathbf{b}) + \texttt{rad}(\mathbf{a} \odot \mathbf{b})] \quad \text{as} \quad a * b > 0 \\ & \subseteq & \mathbf{a} \odot \mathbf{b}. \end{array}$$

□

This proposition can be generalized to the multiplication of interval matrices by induction on row and column indices and by the fact that if $\mathbf{a}' \subseteq \mathbf{a}$ and $\mathbf{b}' \subseteq \mathbf{b}$ then $\mathbf{a}' + \mathbf{b}' \subseteq \mathbf{a} + \mathbf{b}$. Therefore Proposition 2.3.5 always provides sharper results than those computed by Rump's proposition.

2.3.3 Implementation

We will now detail the implementation of the formula given in Proposition 2.3.5. Using floating-point arithmetic to implement interval arithmetic, we have to deal with rounding errors.

For the matrix decomposition as defined by Proposition 2.3.4, the only source of rounding error is the addition between the two bounds, as the rest can be expressed in terms of `max` and `min` functions which are free of rounding errors. Hereafter, MatLab notations are used to describe algorithms: = means assignment. Operations take place between scalars, vectors or matrices as long as their dimensions agree. Indeed, to increase the parallelism of algorithm, instead of using loops, we will express algorithms solely in terms of matrix operations, using MatLab-like syntax.

Let $\mathbf{A} = \left[\underline{A}, \overline{A}\right] \in \mathbb{IF}^{n\times n}$ be an interval matrix, the following algorithm decomposes $\mathbf{A}$ into two parts as defined by Proposition 2.3.4. All operations are componentwise.

Algorithm 2.1 Decomposition algorithm for endpoint intervals

$$\begin{aligned} A0 &= \mathtt{max}(\mathtt{min}(-\underline{A}, \overline{A}), 0) \\ A1 &= \triangledown(\underline{A} + A0) \\ A2 &= \triangle(\overline{A} - A0) \end{aligned}$$

The validity of this algorithm is given by the following proposition.

Proposition 2.3.9. *Let $\mathbf{A} = \left[\underline{A}, \overline{A}\right] \in \mathbb{IF}^{n\times n}$ be an interval matrix. For $A0, A1, A2 \in \mathbb{F}^{n\times n}$ being computed by Algorithm 2.1, it holds that*

1. $\mathbf{A}^0 = [-A0, A0]$,
2. $\mathbf{A}^* \subseteq [A1, A2]$.

Proof. We will inspect several subcases for the component $\mathbf{A}_{i,j}, 1 \leq i, j \leq n$ of $\mathbf{A}$:

- **If** $0 \leq \underline{A}_{i,j} \leq \overline{A}_{i,j}$, then $\mathbf{A}^0_{i,j} = 0$ and $\mathbf{A}^*_{i,j} = \mathbf{A}_{i,j}$. For $A0_{i,j}$ we have

$$A0_{i,j} = \mathtt{max}(\mathtt{min}(-\underline{A}_{i,j}, \overline{A}_{i,j}), 0) = \mathtt{max}(-\underline{A}_{i,j}, 0) = 0.$$

  Since the floating-point addition or subtraction of 0 is exact, then

$$\begin{cases} A1_{i,j} &= \triangledown(\underline{A}_{i,j} + A0_{i,j}) = \underline{A}_{i,j} \\ A2_{i,j} &= \triangle(\overline{A}_{i,j} - A0_{i,j}) = \overline{A}_{i,j} \end{cases}$$

  Hence $[-A0_{i,j}, A0_{i,j}] = 0 = \mathbf{A}^0_{i,j}$, and $[A1_{i,j}, A2_{i,j}] = \left[\underline{A}_{i,j}, \overline{A}_{i,j}\right] = \mathbf{A}^*_{i,j}$.

- **If** $\underline{A}_{i,j} \leq \overline{A}_{i,j} \leq 0$, then $\mathbf{A}^0_{i,j} = 0$ and $\mathbf{A}^*_{i,j} = \mathbf{A}_{i,j}$. For $A0_{i,j}$ we have

$$A0_{i,j} = \mathtt{max}(\mathtt{min}(-\underline{A}_{i,j}, \overline{A}_{i,j}), 0) = \mathtt{max}(\overline{A}_{i,j}, 0) = 0.$$

  Therefore $A1_{i,j} = \underline{A}_{i,j}$ and $A2_{i,j} = \overline{A}_{i,j}$. Hence $[-A0_{i,j}, A0_{i,j}] = 0 = \mathbf{A}^0_{i,j}$, and $[A1_{i,j}, A2_{i,j}] = \left[\underline{A}_{i,j}, \overline{A}_{i,j}\right] = \mathbf{A}^*_{i,j}$.

- **If** $\underline{A}_{i,j} < 0 < |\underline{A}_{i,j}| < \overline{A}_{i,j}$, then $\mathbf{A}^0_{i,j} = \left[\underline{A}_{i,j}, -\underline{A}_{i,j}\right]$, and $\mathbf{A}^*_{i,j} = \left[0, \underline{A}_{i,j} + \overline{A}_{i,j}\right]$.

$$\begin{aligned} A0_{i,j} &= \mathtt{max}(\mathtt{min}(-\underline{A}_{i,j}, \overline{A}_{i,j}), 0) = \mathtt{max}(-\underline{A}_{i,j}, 0) = -\underline{A}_{i,j} \\ \Rightarrow A1_{i,j} &= \bigtriangledown(\underline{A}_{i,j} + A0_{i,j}) = \bigtriangledown(\underline{A}_{i,j} - \underline{A}_{i,j}) = 0, \\ A2_{i,j} &= \bigtriangleup(\overline{A}_{i,j} - A0_{i,j}) = \bigtriangleup(\overline{A}_{i,j} + \underline{A}_{i,j}). \end{aligned}$$

It is obvious that $\bigtriangleup(\overline{A}_{i,j} + \underline{A}_{i,j}) \geq \overline{A}_{i,j} + \underline{A}_{i,j} \Rightarrow \left[0, \overline{A}_{i,j} + \underline{A}_{i,j}\right] \subseteq \left[0, \bigtriangleup(\overline{A}_{i,j} + \underline{A}_{i,j})\right]$. Hence, $\mathbf{A}^*_{i,j} \subseteq [A1_{i,j}, A2_{i,j}]$, and $[-A0_{i,j}, A0_{i,j}] = \left[\underline{A}_{i,j}, -\underline{A}_{i,j}\right] = \mathbf{A}^0_{i,j}$.

- **If** $\underline{A}_{i,j} < 0 < \overline{A}_{i,j} < |\underline{A}_{i,j}|$, then $\mathbf{A}^0_{i,j} = \left[-\overline{A}_{i,j}, \overline{A}_{i,j}\right]$, and $\mathbf{A}^*_{i,j} = \left[\underline{A}_{i,j} + \overline{A}_{i,j}, 0\right]$.

$$\begin{aligned} A0_{i,j} &= \mathtt{max}(\mathtt{min}(-\underline{A}_{i,j}, \overline{A}_{i,j}), 0) = \mathtt{max}(\overline{A}_{i,j}, 0) = \overline{A}_{i,j} \\ \Rightarrow A1_{i,j} &= \bigtriangledown(\underline{A}_{i,j} + A0_{i,j}) = \bigtriangledown(\underline{A}_{i,j} + \overline{A}_{i,j}), \\ A2_{i,j} &= \bigtriangleup(\overline{A}_{i,j} - A0_{i,j}) = 0. \end{aligned}$$

Again, it is obvious that $\bigtriangledown(\overline{A}_{i,j}+\underline{A}_{i,j}) \leq \overline{A}_{i,j}+\underline{A}_{i,j} \Rightarrow \left[\overline{A}_{i,j} + \underline{A}_{i,j}, 0\right] \subseteq \left[\bigtriangledown(\overline{A}_{i,j} + \underline{A}_{i,j}), 0\right]$. Hence, $\mathbf{A}^*_{i,j} \subseteq [A1_{i,j}, A2_{i,j}]$, and $[-A0_{i,j}, A0_{i,j}] = \left[-\overline{A}_{i,j}, \overline{A}_{i,j}\right] = \mathbf{A}^0_{i,j}$.

□

Now we have to compute the two sub-products, which are $[-A0, A0]*\mathbf{B}$ and $[A1, A2]*\mathbf{B}$. Following Section 2.3.1,

$$[-A0, A0] * \mathbf{B} = \left[-A0 * \mathtt{mag}\left(\mathbf{B}\right), A0 * \mathtt{mag}\left(\mathbf{B}\right)\right].$$

Hence to compute $[-A0, A0] * \mathbf{B}$, it is only needed to compute its upper bound, which is $A0 * \mathtt{mag}\left(\mathbf{B}\right)$. In the presence of rounding errors, it must be computed in upward rounding mode.

To compute the multiplication $[A1, A2] * \mathbf{B}$ using Proposition 2.3.3, it is needed to get the negative and positive parts of the two bounds of $[A1, A2]$ and $\mathbf{B}$.

For $a \in \mathbb{F}$, it is true that $a^- = \mathtt{min}(a, 0)$ and $a^+ = \mathtt{max}(a, 0)$.

For $\mathbf{a} = [\underline{a}, \overline{a}] \in \mathbb{IF}$ and $\mathbf{a}$ does not contain zero, it is true that

$$\begin{aligned} \overline{\mathbf{a}^+} &= \mathtt{max}(\overline{a}, 0), \\ \underline{\mathbf{a}^+} &= \mathtt{max}(\underline{a}, 0), \\ \overline{\mathbf{a}^-} &= \mathtt{min}(\overline{a}, 0), \\ \underline{\mathbf{a}^-} &= \mathtt{min}(\underline{a}, 0). \end{aligned}$$

Hence, using Proposition 2.3.3 we can compute the multiplication $[A1, A2] * \mathbf{B}$ by:

$$\begin{aligned} \mathtt{inf}([A1, A2] * \mathbf{B}) &= \bigtriangledown(\mathtt{max}(A2, 0) * \mathtt{min}(\underline{B}, 0) + \mathtt{max}(A1, 0) * \mathtt{max}(\underline{B}, 0) \\ &\quad + \mathtt{min}(A2, 0) * \mathtt{min}(\overline{B}, 0) + \mathtt{min}(A1, 0) * \mathtt{max}(\overline{B}, 0)), \\ \mathtt{sup}([A1, A2] * \mathbf{B}) &= \bigtriangleup(\mathtt{max}(A2, 0) * \mathtt{max}(\overline{B}, 0) + \mathtt{max}(A1, 0) * \mathtt{min}(\overline{B}, 0) \\ &\quad + \mathtt{min}(A2, 0) * \mathtt{max}(\underline{B}, 0) + \mathtt{min}(A1, 0) * \mathtt{min}(\underline{B}, 0)). \end{aligned}$$

Finally, the full algorithm is implemented in MatLab as a function called `igemm` using the IntLab library as follows.

```
1  function R = igemm(A, B)
2      A0 = max(0,min(-A.inf,A.sup));
3      setround(-1);
4      A1 = A.inf + A0;
5      setround(1);
6      A2 = A.sup - A0;

8      A2a = max(A2,0);
9      A2i = min(A2,0);
10     A1a = max(A1,0);
11     A1i = min(A1,0);

13     B2 = B.sup;
14     B1 = B.inf;
15     B2a = max(B2,0);
16     B2i = min(B2,0);
17     B1a = max(B1,0);
18     B1i = min(B1,0);

20     zeroA = all(all(A0 == 0));
21     if (zeroA)
22         A0 = zeros(size(A0,1),size(B,2));
23     else
24         A0 = A0 * max(-B.inf, B.sup);
25     end

27     RU = A2a * B2a + A1a * B2i + A2i * B1a + A1i * B1i + A0;
28     setround(-1);
29     RI = A2a * B1i + A1a * B1a + A2i * B2i + A1i * B2a - A0;

31     R = infsup(RI,RU);
32 end
```

The IntLab library uses endpoints format to represent real intervals. Each real interval term (scalar, vector, or matrix) is represented by two floating-point terms of the same size, one for the lower bound and one for the upper bound. One can obtain the lower bound and the upper bound of each interval term by using the getter functions, which are `.inf` and `.sup` respectively. Beside basic functions of MatLab, we use here some additional functions:

- `setround`(rnd) to change the rounding mode according to the input parameter rnd: 1 means upward rounding mode, -1 means downward rounding mode, and 0 means to-nearest rounding mode;

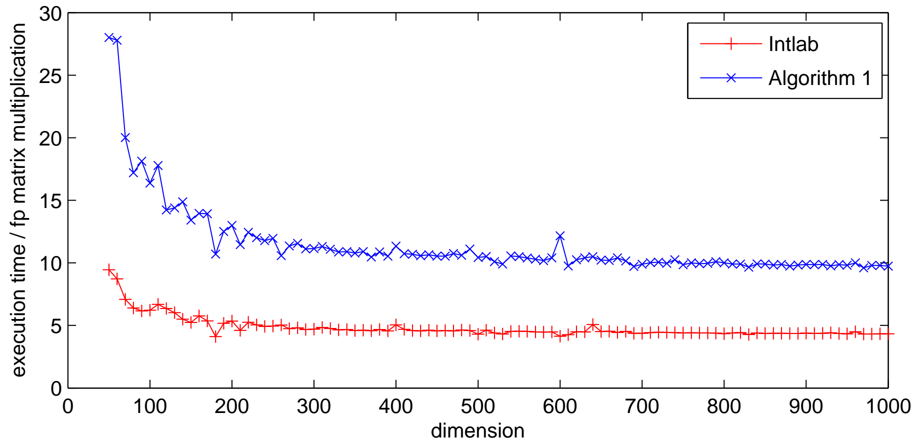
- `infsup`($P1, P2$) to construct an interval matrix whose lower bound and upper bound is respectively $P1$ and $P2$;

- `zeros`(m, n) to construct a matrix of $m \times n$ null components;

- `all`($X == 0$) to test if all elements of X are non-zero. Indeed, if X is a vector, it returns 1 if and only if none of the elements of X is zero. If X is a matrix, `all`(X) operates on the column of X, returning a row vector of logical 1's and 0's.

Lines 2–6 perform the decomposition following Algorithm 2.1, which requires 2 rounding mode changes. Then we do not change the rounding mode and keep the upward rounding

Figure 2.1: Performance of `igemm`.



mode for the following lines: this has no impact on lines 8–22, and it is the required rounding mode for line 24 which consists in computing $A0 * \texttt{mag}(B)$, and line 27 which computes the upper bound of the result. Then we change to downward rounding mode to compute the lower bound of the result in line 29. In total, this implementation uses only 3 changes of rounding mode.

Practical performance

The `igemm` function uses in total 9 floating-point matrix products, along with some matrix operations of order $\mathcal{O}(n^2)$ to manipulate data. Hence, in theory, the factor in terms of execution time between `igemm` and a floating-point matrix product is 9.

In practice, when the matrix dimension is small, the difference between operations of order $\mathcal{O}(n^2)$ and operations of order $\mathcal{O}(n^3)$ is small. That is not only because the theoretical factor n is small, but also that operations of order $\mathcal{O}(n^3)$ better exploit memory locality and parallelism.

Test matrices for this set of experiments as well as other experiments in this section are generated by function $\texttt{rand}(n)$ from MatLab, given n the dimension of the matrix. This function returns a matrix of size $n \times n$ whose components are uniformly distributed random numbers between 0 and 1.

Since the `igemm` function uses only a small constant number of BLAS level 3 routines, the overhead in execution time introduced by the interpretation is negligible. It means that there is almost no difference in performance in terms of execution time between a MatLab implementation and other implementations on C or FORTRAN programming language.

As shown on Figure 2.1, when the matrix dimension is small, i.e smaller than 100, the execution time of `igemm` with respect to a floating-point matrix multiplication, marked by the $\times$ symbol, is high. Nevertheless, when the matrix dimension gets higher, this factor gets smaller and gets close to the theoretical factor.

The same phenomenon can be observed with Rump's algorithm, marked by the $+$ symbol.

In the implementation, for some special cases, we can reduce the number of floating-

point matrix products by inspecting the signs of input matrices. For example, if $\mathbf{A}$ is a non-negative matrix then, following Proposition 2.3.1, the interval multiplication requires only four punctual multiplications. Furthermore, if the upper bound of all the components of $\mathbf{B}$ are of the same sign then either $\overline{B}^-$ or $\underline{B}^+$ is zero. Hence, in total it requires only three punctual matrix multiplications.

The best case is when all the components of $\mathbf{A}$ are of the same sign, and all the components of $\mathbf{B}$ are of the same sign, though it is not a very common case. In this case, we need only two punctual matrix multiplications to implement an interval matrix multiplication.

Implementation issues

Following Proposition 2.3.8, results computed by this algorithm are always included in results computed by the IntLab library. Especially when one of the multiplier does not contain zero, then there is no overestimation with this algorithm. For the implementation using floating-point arithmetic, we face the problem of rounding errors. Therefore, when the input data are of small relative precision, then results computed by the IntLab library are curiously even tighter than results computed by this algorithm. This can be explained by the following example.

Denote by ϵ the relative machine error for double precision, which is 2^{-53}, then $1 + 2\epsilon, 1 + 4\epsilon$ and $1 + 6\epsilon$ are floating-point numbers but $1 + \epsilon, 1 + 3\epsilon$ and $1 + 5\epsilon$ are not. Let's consider the interval multiplication $\mathbf{a} * \mathbf{b}$ where $\mathbf{a} = [1 - 2\epsilon, 1 + 2\epsilon] = \langle 1, 2\epsilon \rangle$, $\mathbf{b} = [1 + 2\epsilon, 1 + 6\epsilon] = \langle 1 + 4\epsilon, 2\epsilon \rangle$.

Using the natural algorithm

$$\begin{aligned} \mathbf{a} * \mathbf{b} &= [\bigtriangledown((1 - 2\epsilon) * (1 + 2\epsilon)), \bigtriangleup((1 + 2\epsilon) * (1 + 6\epsilon))] \\ &= \left[\bigtriangledown(1 - 4\epsilon^2), \bigtriangleup(1 + 8\epsilon + 12\epsilon^2)\right] \\ &= [1 - \epsilon, 1 + 10\epsilon]. \end{aligned}$$

Using the midpoint-radius representation

$$\begin{aligned} \mathtt{mid}(\mathbf{a} \odot \mathbf{b}) &= [\bigtriangledown(1 * (1 + 4\epsilon)), \bigtriangleup(1 * (1 + 4\epsilon))] \\ &= 1 + 4\epsilon, \\ \mathtt{rad}(\mathbf{a} \odot \mathbf{b}) &= \bigtriangleup(2\epsilon * (1 + 4\epsilon + 2\epsilon) + 1 * 2\epsilon) \\ &= \bigtriangleup(4\epsilon + 12\epsilon^2) \\ &= 4\epsilon * (1 + 4\epsilon). \end{aligned}$$

Without having to transform back to endpoints representation, we can see that

$$\mathtt{diam}(\mathbf{a} \odot \mathbf{b}) = 8\epsilon * (1 + 4\epsilon) < 11\epsilon = \mathtt{diam}(\mathbf{a} * \mathbf{b}).$$

This example is somewhat tricky, but its purpose is to show that, taking into account rounding errors, algorithms' behavior is different from that of the mathematical model. Hence, although in theory our algorithm is shown to be always better than Rump's algorithm in terms of accuracy, in practice it is only true when the intervals' width is large enough.

In the following section, we will explore another algorithm which takes advantage of the decomposition proposition, and which is less prone to rounding errors.

2.4 Algorithm 2 using midpoint-radius representation

The interval matrix multiplication algorithm addressed in the previous section is based on the representation of intervals by endpoints. Using this representation, the product between one interval matrix, which does not contain zero in its interior, and another interval matrix requires 8 floating-point matrix products. In general, an interval matrix multiplication as defined by Proposition 2.3.5 requires 9 punctual matrix multiplications. The overestimation in the worst case of this algorithm is $4 - 2\sqrt{2} \approx 1.18$.

In this section, we introduce another algorithm, which is based on the midpoint-radius representation, to compute the product of two interval matrices. This algorithm also uses the decomposition of one multiplier matrices into two parts, one which does not contain zero and the other which is centered in zero.

2.4.1 Scalar interval product

Let us first consider the case of the scalar interval product.

Let $\mathbf{a} = \langle a, \alpha \rangle$ and $\mathbf{b} = \langle b, \beta \rangle$ be two intervals in midpoint-radius representation. Let us split the problem into four cases where $\mathbf{a} \geq 0$, $\mathbf{a} < 0$, $0 \leq a < \alpha$, and $-\alpha < a \leq 0$. For each case, we apply the decomposition proposition to compute the product of $\mathbf{a}$ and $\mathbf{b}$, and to eventually deduce a homogeneous formula in midpoint-radius format for all the four cases.

Case $\mathbf{a} \geq 0$

When $\mathbf{a} \geq 0$, $a + \alpha \geq 0$ and $a - \alpha \geq 0$. Therefore

$$\begin{aligned}
\mathtt{sup}(\mathbf{a} * \mathbf{b}) &= \mathtt{max}((a+\alpha)*(b+\beta), (a-\alpha)*(b+\beta)) \\
&= a*(b+\beta) + \mathtt{max}\{a*(b+\beta), -\alpha*(b+\beta)\} \\
&= a*(b+\beta) + \alpha*|b+\beta|, \\
\mathtt{inf}(\mathbf{a} * \mathbf{b}) &= \mathtt{min}((a+\alpha)*(b-\beta), (a-\alpha)*(b-\beta)) \\
&= a*(b-\beta) + \mathtt{min}(\alpha*(b-\beta), -\alpha*(b-\beta)) \\
&= a*(b-\beta) - \alpha*|b-\beta|.
\end{aligned}$$

Hence

$$\begin{aligned}
\mathtt{mid}(\mathbf{a} * \mathbf{b}) &= \frac{\mathtt{sup}(\mathbf{a} * \mathbf{b}) + \mathtt{inf}(\mathbf{a} * \mathbf{b})}{2} \\
&= \frac{a*(b+\beta) + \alpha*|b+\beta| + a*(b-\beta) - \alpha*|b-\beta|}{2} \\
&= a*b + \alpha*\frac{|b+\beta| - |b-\beta|}{2}, \\
\mathtt{rad}(\mathbf{a} * \mathbf{b}) &= \frac{\mathtt{sup}(\mathbf{a} * \mathbf{b}) - \mathtt{inf}(\mathbf{a} * \mathbf{b})}{2} \\
&= \frac{a*(b+\beta) + \alpha*|b+\beta| - (a*(b-\beta) - \alpha*|b-\beta|)}{2} \\
&= a*\beta + \alpha*\frac{|b+\beta| + |b-\beta|}{2}.
\end{aligned}$$

Case $\mathbf{a} < 0$

When $\mathbf{a} < 0$, $a + \alpha < 0$ and $a - \alpha < 0$. Therefore

$$\begin{aligned}
\texttt{sup}(\mathbf{a} * \mathbf{b}) &= \texttt{max}\{(a+\alpha)*(b-\beta), (a-\alpha)*(b-\beta)\} \\
&= a*(b-\beta) + \texttt{max}\{\alpha*(b-\beta), -\alpha*(b-\beta)\} \\
&= a*(b-\beta) + \alpha*|b-\beta|, \\
\texttt{inf}(\mathbf{a} * \mathbf{b}) &= \texttt{min}\{(a+\alpha)*(b+\beta), (a-\alpha)*(b+\beta)\} \\
&= a*(b+\beta) + \texttt{min}\{\alpha*(b+\beta), -\alpha*(b+\beta)\} \\
&= a*(b-\beta) - \alpha*|b+\beta|.
\end{aligned}$$

Hence

$$\begin{aligned}
\texttt{mid}(\mathbf{a} * \mathbf{b}) &= \frac{\texttt{sup}(\mathbf{a} * \mathbf{b}) + \texttt{inf}(\mathbf{a} * \mathbf{b})}{2} \\
&= \frac{a*(b-\beta) + \alpha*|b-\beta| + a*(b+\beta) - \alpha*|b+\beta|}{2} \\
&= a*b - \alpha*\frac{|b+\beta| - |b-\beta|}{2}, \\
\texttt{rad}(\mathbf{a} * \mathbf{b}) &= \frac{\texttt{sup}(\mathbf{a} * \mathbf{b}) - \texttt{inf}(\mathbf{a} * \mathbf{b})}{2} \\
&= \frac{a*(b-\beta) + \alpha*|b-\beta| - (a*(b+\beta) - \alpha*|b-\beta|)}{2} \\
&= -a*\beta + \alpha*\frac{|b+\beta| + |b-\beta|}{2}.
\end{aligned}$$

Denote by $\texttt{sign}(a)$ the sign of a, these two cases where $\mathbf{a}$ does not contain zero can be rewritten by the same formula

$$\begin{cases}
\texttt{mid}(\mathbf{a} * \mathbf{b}) = a*b + \texttt{sign}(a)*\alpha*\frac{|b+\beta|-|b-\beta|}{2}, \\
\texttt{rad}(\mathbf{a} * \mathbf{b}) = |a|*\beta + \alpha*\frac{|b+\beta|+|b-\beta|}{2}.
\end{cases}$$

Moreover $\beta \geq 0$, hence

- If $b > \beta \geq 0$ then $\begin{cases} |b+\beta| + |b-\beta| = 2*b, \\ |b+\beta| - |b-\beta| = 2*\beta. \end{cases}$

- If $\beta \geq b \geq 0$ then $\begin{cases} |b+\beta| + |b-\beta| = 2*\beta, \\ |b+\beta| - |b-\beta| = 2*b. \end{cases}$

- If $0 \geq b \geq -\beta$ then $\begin{cases} |b+\beta| + |b-\beta| = 2*\beta, \\ |b+\beta| - |b-\beta| = -2*b. \end{cases}$

- If $b < -\beta \leq 0$ then $\begin{cases} |b+\beta| + |b-\beta| = -2*b, \\ |b+\beta| - |b-\beta| = -2*\beta. \end{cases}$

It is easy to deduce that

$$\begin{aligned}
\frac{|b+\beta| + |b-\beta|}{2} &= \texttt{max}(|b|, \beta), \\
\frac{|b+\beta| - |b-\beta|}{2} &= \texttt{sign}(b)*\texttt{min}(|b|, \beta).
\end{aligned}$$

Therefore

$$\begin{cases} \texttt{mid}(\mathbf{a} * \mathbf{b}) &= a * b + \texttt{sign}(a) * \alpha * \texttt{sign}(b) * \texttt{min}(|b|, \beta), \\ \texttt{rad}(\mathbf{a} * \mathbf{b}) &= |a| * \beta + \alpha * \texttt{max}(|b|, \beta). \end{cases} \tag{2.10}$$

Case $0 \le a < \alpha$

We have $a - \alpha < 0 < |a - \alpha| < a + \alpha$. Using the technique of decomposition from section 2.3, $\mathbf{a} = \langle a, \alpha \rangle = [a - \alpha, a + \alpha]$ is decomposed into two parts as follows:

$$\begin{array}{rcccl} \mathbf{a}^0 &=& [a - \alpha, \alpha - a] &=& \langle 0, \alpha - a \rangle, \\ \mathbf{a}^* &=& [0, 2 * a] &=& \langle a, a \rangle. \end{array}$$

As $\mathbf{a}^0$ is centered in zero, $\texttt{mid}(\mathbf{a}^0 * \mathbf{b}) = 0$ and

$$\begin{aligned} \texttt{rad}(\mathbf{a}^0 * \mathbf{b}) &= (\alpha - a) * \texttt{mag}(\mathbf{b}) \\ &= (\alpha - a) * (|b| + \beta). \end{aligned}$$

Moreover, $\mathbf{a}^*$ does not contain zero, and following (2.10)

$$\begin{cases} \texttt{mid}(\mathbf{a}^* * \mathbf{b}) &= a * b + a * \texttt{sign}(b) * \texttt{min}(|b|, \beta), \\ \texttt{rad}(\mathbf{a}^* * \mathbf{b}) &= a * \beta + a * \texttt{max}(|b|, \beta). \end{cases}$$

Due to the sub-distributivity property of interval arithmetic, we have

$$\mathbf{a} * \mathbf{b} \subseteq \mathbf{a}^0 * \mathbf{b} + \mathbf{a}^* * \mathbf{b} \equiv \mathbf{a} \circledast \mathbf{b}$$

where

$$\begin{aligned} \texttt{mid}(\mathbf{a} \circledast \mathbf{b}) &= \texttt{mid}(\mathbf{a}^* * \mathbf{b}) + \texttt{mid}(\mathbf{a}^0 * \mathbf{b}) \\ &= a * b + a * \texttt{sign}(b) * \texttt{min}(|b|, \beta), \\ \texttt{rad}(\mathbf{a} \circledast \mathbf{b}) &= \texttt{rad}(\mathbf{a}^* * \mathbf{b}) + \texttt{rad}(\mathbf{a}^0 * \mathbf{b}) \\ &= a * \beta + a * \texttt{max}(|b|, \beta) + (\alpha - a) * (|b| + \beta) \\ &= a * \beta + \alpha * |b| + \alpha * \beta - a * (|b| + \beta - \texttt{max}(|b|, \beta)) \\ &= a * \beta + \alpha * |b| + \alpha * \beta - a * \texttt{min}(|b|, \beta). \end{aligned}$$

Case $0 \ge a > -\alpha$

We have $a - \alpha < 0 < a + \alpha < |a - \alpha|$. Using the technique of decomposition from section 2.3, $\mathbf{a}$ is decomposed into two parts as follows:

$$\begin{array}{rcccl} \mathbf{a}^0 &=& [-(a + \alpha), \alpha + a] &=& \langle 0, a + \alpha \rangle, \\ \mathbf{a}^* &=& [2 * a, 0] &=& \langle a, -a \rangle. \end{array}$$

As $\mathbf{a}^0$ is centered in zero, $\texttt{mid}(\mathbf{a}^0 * \mathbf{b}) = 0$ and

$$\begin{aligned} \texttt{rad}(\mathbf{a}^0 * \mathbf{b}) &= (\alpha + a) * \texttt{mag}(\mathbf{b}) \\ &= (\alpha + a) * (|b| + \beta). \end{aligned}$$

Again, $\mathbf{a}^*$ does not contain zero, following (2.10)

$$\begin{aligned} \texttt{mid}(\mathbf{a}^* * \mathbf{b}) &= a * b + \texttt{sign}(a) * (-a) * \texttt{sign}(b) * \texttt{min}(|b|, \beta) \\ &= a * b + a * \texttt{sign}(b) * \texttt{min}(|b|, \beta), \\ \texttt{rad}(\mathbf{a}^* * \mathbf{b}) &= |a| * \beta - a * \texttt{max}(|b|, \beta). \end{aligned}$$

Due to the sub-distributive property of interval arithmetic, we have

$$\mathbf{a} * \mathbf{b} \subseteq \mathbf{a}^0 * \mathbf{b} + \mathbf{a}^* * \mathbf{b} \equiv \mathbf{a} \circledast \mathbf{b}$$

where

$$\begin{aligned}
\mathtt{mid}(\mathbf{a} \circledast \mathbf{b}) &= a * b + a * \mathtt{sign}(b) * \mathtt{min}(|b|, \beta), \\
\mathtt{rad}(\mathbf{a} \circledast \mathbf{b}) &= |a| * \beta - a * \mathtt{max}(|b|, \beta) + (\alpha + a) * (|b| + \beta) \\
&= |a| * \beta + \alpha * |b| + \alpha * \beta + a * (|b| + \beta - \mathtt{max}(|b|, \beta)) \\
&= |a| * \beta + \alpha * |b| + \alpha * \beta - |a| * \mathtt{min}(|b|, \beta).
\end{aligned}$$

Hence, the last two cases where $0 \leq a < \alpha$ or $0 \geq a > -\alpha$ can be rewritten in the same formula

$$\begin{cases} \mathtt{mid}(\mathbf{a} \circledast \mathbf{b}) = a * b + a * \mathtt{sign}(b) * \mathtt{min}(|b|, \beta), \\ \mathtt{rad}(\mathbf{a} \circledast \mathbf{b}) = |a| * \beta + \alpha * |b| + \alpha * \beta - |a| * \mathtt{min}(|b|, \beta). \end{cases} \tag{2.11}$$

Let's now compare the two formulae (2.10) and (2.11) to deduce a homogeneous formula for the general case. Let us recall Formula (2.10) for the case where $\mathbf{a}$ does not contain zero:

$$\begin{cases} \mathtt{mid}(\mathbf{a} * \mathbf{b}) = a * b + \mathtt{sign}(a) * \alpha * \mathtt{sign}(b) * \mathtt{min}(|b|, \beta), \\ \mathtt{rad}(\mathbf{a} * \mathbf{b}) = |a| * \beta + \alpha * \mathtt{max}(|b|, \beta). \end{cases}$$

If $\mathbf{a}$ does not contain zero, then $\mathtt{min}(|a|, \alpha) = \alpha$.

Otherwise, if $\mathbf{a}$ contains zero, then $\mathtt{min}(|a|, \alpha) = |a|$. Hence $a = \mathtt{sign}(a) * |a| = \mathtt{sign}(a) * \mathtt{min}(|a|, \alpha)$.

Therefore, the midpoint of the two formulae (2.10) and (2.11) can be rewritten in the same form:

$$\mathtt{mid}(\mathbf{a} \circledast \mathbf{b}) = a * b + \mathtt{sign}(a) * \mathtt{min}(|a|, \alpha) * \mathtt{sign}(b) * \mathtt{min}(|b|, \beta).$$

Moreover, when $\mathbf{a}$ does not contain zero

$$\begin{aligned}
\mathtt{rad}(\mathbf{a} * \mathbf{b}) &= |a| * \beta + \alpha * \mathtt{max}(|b|, \beta) \\
&= |a| * \beta + \alpha * (|b| + \beta - \mathtt{min}(|b|, \beta)) \\
&= |a| * \beta + \alpha * |b| + \alpha * \beta - \alpha * \mathtt{min}(|b|, \beta)) \\
&= |a| * \beta + \alpha * |b| + \alpha * \beta - \mathtt{min}(|a|, \alpha) * \mathtt{min}(|b|, \beta).
\end{aligned}$$

Finally, these two formulae can be rewritten in the same form for both cases, whether $\mathbf{a}$ contains zero or not:

$$\begin{cases} \mathtt{mid}(\mathbf{a} \circledast \mathbf{b}) = a * b + \mathtt{sign}(a) * \mathtt{min}(|a|, \alpha) * \mathtt{sign}(b) * \mathtt{min}(|b|, \beta), \\ \mathtt{rad}(\mathbf{a} \circledast \mathbf{b}) = |a| * \beta + \alpha * |b| + \alpha * \beta - \mathtt{min}(|a|, \alpha) * \mathtt{min}(|b|, \beta). \end{cases} \tag{2.12}$$

Denote by $\begin{cases} \mathtt{mmr}(\mathbf{a}) = \mathtt{sign}(a) * \mathtt{min}(|a|, \alpha) \\ \mathtt{mmr}(\mathbf{b}) = \mathtt{sign}(b) * \mathtt{min}(|b|, \beta) \end{cases}$

then (2.12) is rewritten by:

$$\begin{cases} \mathtt{mid}(\mathbf{a} \circledast \mathbf{b}) = a * b + \mathtt{mmr}(\mathbf{a}) * \mathtt{mmr}(\mathbf{b}), \\ \mathtt{rad}(\mathbf{a} \circledast \mathbf{b}) = |a| * \beta + \alpha * |b| + \alpha * \beta - |\mathtt{mmr}(\mathbf{a})| * |\mathtt{mmr}(\mathbf{b})|. \end{cases} \tag{2.13}$$

This formula differs from Rump's algorithm (2.2) only by the term $\texttt{mmr}(\mathbf{a}) * \texttt{mmr}(\mathbf{b})$. It can be considered as a correction term to Rump's algorithm. Using this correction term and the fact that $|\texttt{mmr}(\mathbf{a}) * \texttt{mmr}(\mathbf{b})| \leq |\texttt{mmr}(\mathbf{a})| * |\texttt{mmr}(\mathbf{b})|$, it is easy to deduce that $\mathbf{a} \circledast \mathbf{b}$ is always included in $\mathbf{a} \odot \mathbf{b}$ for all $\mathbf{a} \in \mathbb{IR}$ and $\mathbf{b} \in \mathbb{IR}$. The following proposition provides an estimation of the accuracy of $\mathbf{a} \circledast \mathbf{b}$.

Proposition 2.4.1. *Let $\mathbf{a}, \mathbf{b}$ be two intervals. The product between $\mathbf{a}$ and $\mathbf{b}$ as defined by (2.13) is:*

- *equal to the exact product when $\mathbf{a}$ or $\mathbf{b}$ does not contain zero,*
- *an enclosure of the exact product with an overestimation factor of less than or equal to $4 - 2\sqrt{2}$ when both $\mathbf{a}$ and $\mathbf{b}$ contain zero in its interior.*

Proof. Using the decomposition proposition to deduce the formula, the overestimation factor can also easily be deduced from the previous section. □

2.4.2 Matrix and vector operations

Formula (2.13) for the product of two intervals can be generalized to matrix and vector operations.

Proposition 2.4.2. *For $\mathbf{A} = \langle A, \Lambda \rangle \in \mathbb{IR}^{m \times n}$, $\mathbf{B} = \langle B, \Sigma \rangle \in \mathbb{IR}^{n \times k}$, the multiplication $\mathbf{A} \circledast \mathbf{B}$ is defined as*

$$\begin{cases} \texttt{mid}(\mathbf{A} \circledast \mathbf{B}) & = & A * B + \texttt{mmr}(\mathbf{A}) * \texttt{mmr}(\mathbf{B}), \\ \texttt{rad}(\mathbf{A} \circledast \mathbf{B}) & = & |A| * \Sigma + \Lambda * |B| + \Lambda * \Sigma - |\texttt{mmr}(\mathbf{A})| * |\texttt{mmr}(\mathbf{B})|. \end{cases} \tag{2.14}$$

Then $\mathbf{A} * \mathbf{B} \subseteq \mathbf{A} \circledast \mathbf{B}$.

Proof. Matrix and vector operations are both based on dot product. Therefore, the isotonicity of (2.14) can be proven by considering the case of the interval dot product.

Let $\mathbf{a}$ and $\mathbf{b}$ be two vectors of n intervals.

$$\begin{aligned}
\mathbf{a} * \mathbf{b}^T &= \sum_{i=1}^{n} \mathbf{a}_i * \mathbf{b}_i \\
&\subseteq \sum_{i=1}^{n} \mathbf{a}_i \circledast \mathbf{b}_i \\
&\subseteq \sum_{i=1}^{n} \langle a_i * b_i + \texttt{mmr}(\mathbf{a}_i) * \texttt{mmr}(\mathbf{b}_i), |a_i| * \beta_i + \alpha_i * |b_i| + \alpha_i * \beta_i - |\texttt{mmr}(\mathbf{a}_i)| * |\texttt{mmr}(\mathbf{b}_i)| \rangle \\
&\subseteq \Bigg\langle \sum_{i=1}^{n} a_i * b_i + \texttt{mmr}(\mathbf{a}_i) * \texttt{mmr}(\mathbf{b}_i), \\
&\qquad \sum_{i=1}^{n} |a_i| * \beta_i + \alpha_i * |b_i| + \alpha_i * \beta_i - |\texttt{mmr}(\mathbf{a}_i)| * |\texttt{mmr}(\mathbf{b}_i)| \Bigg\rangle \\
&\subseteq \left\langle a * b^T + \texttt{mmr}(\mathbf{a}) * \texttt{mmr}(\mathbf{b})^T, |a| * \beta^T + \alpha * |b|^T + \alpha * \beta^T - |\texttt{mmr}(\mathbf{a})| * |\texttt{mmr}(\mathbf{b})|^T \right\rangle \\
&\subseteq \mathbf{a} \circledast \mathbf{b}^T.
\end{aligned}$$

□

Let $\mathbf{a}$ and $\mathbf{b}$ be two vectors of n intervals. Because there is no overestimation in interval addition, and thus $\mathtt{diam}(\mathbf{x}+\mathbf{y}) = \mathtt{diam}(\mathbf{x})+\mathtt{diam}(\mathbf{y})$, by induction arguments, we can prove that

$$\mathtt{diam}(\mathbf{a} \circledast \mathbf{b}) \leq (4-2\sqrt{2}) * \mathtt{diam}(\mathbf{a} * \mathbf{b}).$$

Moreover, if no component of $\mathbf{a}$ contains zero, then $\mathbf{a} \circledast \mathbf{b} = \mathbf{a} * \mathbf{b}$. By consequence, we get the following theorem.

Theorem 2.4.3. *For $\mathbf{A} = \langle A, \Lambda \rangle \in \mathbb{IR}^{m\times n}$, $\mathbf{B} = \langle B, \Sigma \rangle \in \mathbb{IR}^{n\times k}$, the multiplication $\mathbf{A} \circledast \mathbf{B}$ is defined as (2.14). It holds that:*

1. *the overestimation for each component of $\mathbf{A} \circledast \mathbf{B}$ is always bounded by $4-2\sqrt{2}$;*
2. *if no component of $\mathbf{A}$ or $\mathbf{B}$ contains zero, then there is no overestimation.*

2.4.3 Implementation

To implement this algorithm, we first have to implement the function `mmr`. Getting the sign of a matrix is often costly, hence instead of using the original formula we will use an alternate formula which employs only maximum and minimum functions.

Proposition 2.4.4. *For $\mathbf{a} = \langle a, \alpha \rangle \in \mathbb{IR}$*

$$\mathtt{mmr}(\mathbf{a}) = \mathtt{min}(\alpha, \mathtt{max}(a, -\alpha)).$$

Proof. The equivalence of this alternative formula and the original formula can be checked by inspecting all possible relative positions of a with respect to 0 and α.

1. If $a \geq 0$ then $\mathtt{sign}(a) = 1$ and $a = |a|$. Hence

$$\mathtt{min}(\alpha, \mathtt{max}(a, -\alpha)) = \mathtt{min}(\alpha, a) = \mathtt{sign}(a) * \mathtt{min}(|a|, \alpha) = \mathtt{mmr}(\mathbf{a}).$$

2. If $a < 0$ then $\mathtt{sign}(a) = -1$ and $\mathtt{max}(a, -\alpha) = \mathtt{max}(-|a|, -\alpha) = -\mathtt{min}(|a|, \alpha) < 0 < \alpha$. Hence

$$\mathtt{min}(\alpha, \mathtt{max}(a, -\alpha)) = -\mathtt{min}(|a|, \alpha) = \mathtt{sign}(a) * \mathtt{min}(|a|, \alpha) = \mathtt{mmr}(\mathbf{a}).$$

□

The implementation in IntLab of the function `mmr`, using the previous proposition, exhibits some minor improvement. It is faster than the straightforward implementation using the original formula by a factor 4/3.

Although IntLab library supports both endpoints and midpoint-radius representations of interval, for real input data, intervals are exposed to the user in endpoints format. Midpoint-radius format is only used internally for complex data. Hence, to implement our algorithm, we have to transform the input data from endpoints representation to midpoint-radius representation before performing computations, and finally transform results from midpoint-radius representation back to endpoints representation for output.

Using floating-point arithmetic to implement interval arithmetic, rounding errors must be taken care of when transforming interval data. Transforming from midpoint-radius representation to endpoints representation is simple.

Proposition 2.4.5. *For* $\mathbf{a} = \langle a, \alpha \rangle \in \mathbb{IF}$ *an interval in midpoint-radius representation, an enclosing endpoints representation is an interval* $[\underline{a}, \overline{a}]$ *being computed by:*

$$\begin{aligned} \underline{a} &= \triangledown(a - \alpha), \\ \overline{a} &= \triangle(a + \alpha). \end{aligned}$$

It holds that $\mathbf{a} \subseteq [\underline{a}, \overline{a}]$.

To transform intervals from endpoints representation to midpoint-radius representation, we use an algorithm from Intlab library [Rum], which is due to Oishi [Ois98].

Proposition 2.4.6. *For* $\mathbf{a} = [\underline{a}, \overline{a}] \in \mathbb{IF}$ *an interval in endpoints representation, an enclosing midpoint-radius representation is an interval* $\langle a, \alpha \rangle$ *being computed by:*

$$\begin{aligned} a &= \triangle(\underline{a} + \overline{a}) * 0.5, \\ \alpha &= \triangle(a - \underline{a}). \end{aligned}$$

It is true that $\mathbf{a} \subseteq \langle a, \alpha \rangle$.

The isotonicity of this algorithm is given below.

Proof. Using directed rounding mode, we have that:

$$\alpha = \triangle(a - \underline{a}) \geq a - \underline{a}.$$

Hence, $a - \alpha \leq \underline{a}$.
Moreover

$$a = \triangle(\underline{a} + \overline{a}) * 0.5 \geq (\underline{a} + \overline{a}) * 0.5.$$

Hence $a + \alpha \geq a + (a - \underline{a}) \geq 2 * a - \underline{a} \geq (\underline{a} + \overline{a}) - \underline{a} = \overline{a}$.
Combining this with $a - \alpha \leq \underline{a}$, we get $a - \alpha \leq \underline{a} \leq \overline{a} \leq a + \alpha$ and thus $\mathbf{a} \subseteq \langle a, \alpha \rangle$. □

Like in Rump's algorithm, because of rounding errors, the midpoint of the result cannot be computed exactly using formula (2.14). Hence, we have to compute an enclosure of the midpoint. It means that it must be computed twice, once with downward rounding mode and once with upward rounding mode.

What follows is a function named `igemm3` implementing this algorithm in MatLab, using the IntLab library.

```
1  function C = igemm3(A,B)
2      setRound(1);  % switch to upward rounding mode
3      mA = (sup(A) + inf(A)) * 0.5;
4      mB = (sup(B) + inf(B)) * 0.5;
5      rA = mA - inf(A);
6      rB = mB - inf(B);
7      mmA = min(max(mA,-rA),rA);
8      mmB = min(max(mB,-rB),rB);

10     mCu = mA * mB + mmA * mmB;
11     setRound(-1); % switch to downward rounding mode
12     mCd = mA * mB + mmA * mmB;

14     rC = abs(mmA) * abs(mmB);
15     setRound(1);  % switch to upward rounding mode
```

```
16      rC = rA * (rB + abs(mB)) + abs(mA) * rB - rC;

18      mCu = mCu + rC;
19      setRound(-1);  % switch to downward rounding mode
20      mCd = mCd - rC;

22      C = infsup(mCd, mCu);
23  end
```

This implementation is straightforward with respect to the theoretical formula (2.14). It is obvious that rC computed at line 14 of the code snippet above is non-negative. Hence rC computed at line 16 must be smaller or equal to $\mathrm{rA} * (\mathrm{rB} + \mathtt{abs}(\mathrm{mB})) + \mathtt{abs}(\mathrm{mA}) * \mathrm{rB}$, which is the radius of Rump's algorithm.Therefore, apparently, the results computed by this implementation should be tighter than results computed by the IntLab library. However, in practice, when all the components of $\mathbf{A}$ and $\mathbf{B}$ are of small relative precision then the results computed by the IntLab library are slightly tighter. This is in fact the consequence of rounding errors. For example, using double precision with mantissa of length 52 bits for the computations:

$$\triangle(1 + 2^{-54} - 2^{-53}) = \triangle(\triangle(1 + 2^{-54}) - 2^{-53}) = \triangle((1 + 2^{-52}) - 2^{-53}) = 1 + 2^{-52} > 1.$$

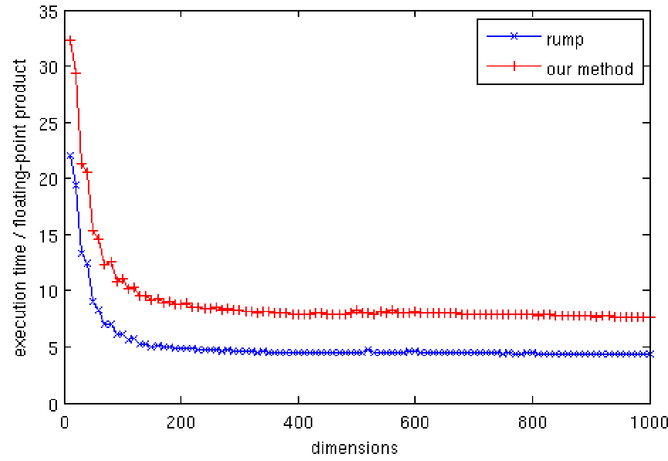
With the presence of rounding errors, computed results depend not only on the input data but also on the order of the computations. With the same example as above, if we change the order of computations then the computed result is better.

$$\triangle(1 + 2^{-54} - 2^{-53}) = \triangle(1 + \triangle(2^{-54} - 2^{-53})) = \triangle(1 - 2^{-54}) = 1.$$

One useful trick to minimize the effect of rounding errors when performing floating-point summation is to sum the smaller terms first. It is shown in [Hig02, chap.4] to be the best strategy (in terms of the accuracy of the final result) when all summands are non-negative. It is often a good heuristic otherwise. By definition of mmA and mmB, $\mathrm{mmA} * \mathrm{mmB}$ and $\mathtt{abs}(\mathrm{mmA}) * \mathtt{abs}(\mathrm{mmB})$ should be the smallest among the other sub-products. Because in the implementation of $\mathbf{A} \circledast \mathbf{B}$, we need to compute the addition and the subtraction between midpoint and radius to get the final result in endpoints representations, we will first compute these two smallest products, then compute the other sub-products to hopefully reduce rounding errors. This leads to another implementation of the formula (2.14).

```
1   function C = igemm4(A,B)
2       setRound(1);
3       mA = (sup(A) + inf(A)) * 0.5;
4       mB = (sup(B) + inf(B)) * 0.5;
5       rA = mA - inf(A);
6       rB = mB - inf(B);

8       mmA = min(max(mA,-rA),rA);
9       mmB = min(max(mB,-rB),rB);

11      setRound(-1);

13      rC = abs(mmA) * abs(mmB);
14      mCd = mmA * mmB + rC;

16      setRound(1);
```

Figure 2.2: Performance of igemm3.




```
17        mCu = mmA * mmB - rC;
18        rC = rA * (rB + abs(mB)) + abs(mA) * rB;

20        mCu = mCu + rC;
21        mCu = mCu + mA * mB;
22        setRound(-1);
23        mCd = mCd - rC;
24        mCd = mCd + mA * mB;

26        C = infsup(mCd, mCu);
27    end
```

With this new implementation, results computed by (2.14) are at least as accurate as results computed by the IntLab library when input intervals are small. On the other hand, when input intervals are thick, this implementation provides results tighter than those computed by the IntLab library.

Using 7 floating-point matrix multiplication, the theoretical factor of this algorithm in terms of execution time with respect to floating-point multiplication is 7. In practice, because of data manipulation which is of order $\mathcal{O}(n^2)$, when the matrix dimension is small, the practical factor is much higher than the theoretical factor. The theoretical factor is only reached when matrix dimension is high enough, i.e larger than 100×100, see Figure 2.2.

2.5 Conclusion

Using solely floating-point matrix operations to implement interval matrix products, the performance of our algorithms depends strongly on the performance of the employed floating-point matrix library. Moreover, these algorithms can be easily parallelized by using parallelized versions of the floating-point matrix library.

We can use whatever library optimized for the running system is available, with only one requirement, which is that this library must support upward and downward rounding

modes. We can cite here for example the ATLAS and GotoBLAS libraries. Unfortunately, this requirement prevents us from using Strassen-like algorithms or even faster ones [Str69, Pan84, Hig02] to reduce the complexity of floating-point matrix product to less than $\mathcal{O}(n^3)$.

Rump's algorithm computes the interval matrix products very fast, with only four floating-point matrix products, but suffers from a factor of overestimation in the worst case of 1.5. Our algorithms provide a trade-off between performance and accuracy. With five more floating-point matrix products for the first algorithm, and three more floating-point matrix products for the second algorithm, we manage to reduce the overestimation factor in the worst case to approximately 1.18. Moreover, when one multiplier does not contain zero, then our algorithms provide exact results in exact arithmetic. In the presence of rounding errors, the second proposed algorithm always provides results of smaller width than those computed by Rump's algorithm.

Chapter 3

Verification of the solution of a linear system

This chapter deals with the verification of the floating-point solutions of linear systems of equations. Our approach is based on the floating-point iterative refinement, applied to the system pre-multiplied by an approximate inverse of the coefficient matrix. In our method, interval arithmetic is used to compute the pre-multiplication and the residual. Then interval iterative improvements are applied, namely the interval Jacobi or Gauss-Seidel iterations. The computed result is a tight enclosure of the correction term and not merely an approximation of if. This method is implemented in MatLab using the IntLab library as a function called `certifylss` *to perform numerical experiments, and to compare our method with an algorithm proposed by S.M. Rump, which is packed in the IntLab library as the function called* `verifylss`*. Numerical results shows the high accuracy of the two verified functions at the cost of a longer execution time in comparison with the MatLab non-verified built-in function. When the condition number of the matrix of the system is small or moderate,* `certifylss` *is faster than* `verifylss`*. Nevertheless,* `certifylss` *gets slower than* `verifylss` *as the coefficient matrix is ill-conditioned.*

3.1 Introduction

In this chapter, our approach is presented for solving a linear system in floating-point arithmetic and at the same time verifying the computed solution. "Verify" means to compute an enclosure of the error by switching from floating-point arithmetic to interval arithmetic to solve the residual system: this yields a guaranteed enclosure of the error between the exact result and the approximate floating-point result. This idea can be found in the `verifylss` function of the IntLab library [Rum]. The function `verifylss` first computes a floating-point approximation of the solution using *iterative refinement*, then it switches to interval arithmetic to compute a "tight" interval error bound using a method due to Neumaier [Neu99]. Our proposal is to use alternately floating-point arithmetic and interval arithmetic to refine simultaneously the approximation and the error bound.

The use of the residual is the basis of iterative refinement methods [Mol67, Ske80, Wil63, Hig02]. An enclosure of the error can be computed, using interval arithmetic, by

adapting one of these iterative refinement methods. These two building blocks, i.e, the floating-point solution of a linear system and the iterative refinement of the error bounds using interval arithmetic, are combined to produce a more accurate solution along with a tight enclosure of the error. In practice, the error bound is tight enough to indicate the number of correct digits of the approximate solution.

3.2 State of the art, `verifylss` function

It is well known that interval arithmetic allows us to obtain verified results. A very naive way to apply interval arithmetic is by taking existing floating-point algorithms, replacing all the input data by punctual intervals, as well as replacing all the arithmetic operations by their corresponding interval operations. Hence the computed result will be an interval which includes the exact result. For example, to solve a lower triangular linear system of equations using Gaussian elimination, performing all the operations in interval arithmetic yields an enclosure of the solution. Some people tend to think of this technique as a "sloppy" way to compute a verified result, "sloppy" meaning here "lazy".

In practice, in some cases, this technique might provide results, which are verified with a large relative error. In the general case, this technique fails to provide a computed result. In such a situation, however, it is incorrect to state that the problem is singular. Consider the following example, taken from [Rum05].

Example 4. Let us solve the triangular linear system

$$\begin{pmatrix} 1-\epsilon & & & & \\ 1 & 1 & & & \\ 1 & 1 & 1 & & \\ 1 & 1 & 1 & 1 & \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

where $\epsilon = 2^{-53}$ is the machine relative error. Hence $1-\epsilon$ and $1+2\epsilon$ are floating-point numbers. Nevertheless, $1+\epsilon$ is not a floating-point number.

Using forward substitution to solve this lower triangular system in interval arithmetic, it is obvious that

$$\begin{aligned} x_1 &= 1/(1-\epsilon) = [1, 1+2\epsilon] \\ x_2 &= 1 - x_1 = 1 - [1, 1+2\epsilon] = [-2\epsilon, 0] \\ x_3 &= 1 - x_1 - x_2 = 1 - [1, 1+2\epsilon] - [-2\epsilon, 0] = [-2\epsilon, 2\epsilon] \\ x_4 &= 1 - x_1 - x_2 - x_3 = 1 - [1, 1+2\epsilon] - [-2\epsilon, 0] - [-2\epsilon, 2\epsilon] = [-4\epsilon, 4\epsilon] \\ x_5 &= 1 - x_1 - x_2 - x_3 - x_4 = 1 - [1, 1+2\epsilon] - [-2\epsilon, 0] - [-2\epsilon, 2\epsilon] - [-4\epsilon, 4\epsilon] = [-8\epsilon, 8\epsilon]. \end{aligned}$$

Note that, except for the computation of the first element which leads to an interval of small width, all the other computations are exact, i.e. no rounding error occurs. By induction one can prove that, for $i \geq 3$,

$$x_i = \left[-2^{i-2}\epsilon, 2^{i-2}\epsilon\right].$$

Hence the width of the computed solution's components explodes exponentially fast with respect to row's index. Meanwhile it is easy to verify that the exact solution is $[1/(1-\epsilon), -\epsilon/(1-\epsilon), 0, 0, 0, 0]^T$. Moreover, the coefficient matrix is well-conditioned, because

$$\begin{pmatrix} 1-\epsilon & & & & \\ 1 & 1 & & & \\ 1 & 1 & 1 & & \\ 1 & 1 & 1 & 1 & \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1/(1-\epsilon) & & & & \\ -1/(1-\epsilon) & 1 & & & \\ & -1 & 1 & & \\ & & -1 & 1 & \\ & & & -1 & 1 \end{pmatrix}.$$

The main reason to that explosion is that there is no cancellation in interval arithmetic. For all $\mathbf{x} \in \mathbb{IR}$, then $\mathbf{x} - \mathbf{x} \neq 0$. Addition and subtraction both increase the width of intervals. Hence, a naive application of interval arithmetic does not provide good results. One important trick when working with interval arithmetic is to use original data, in punctual form, as long as possible [Rum05].

The `verifylss` function of the IntLab library [Rum] verifies the solution of a linear system. Its signature is given below

$$x = \texttt{verifylss}(A, b)$$

where A can be a floating-point or interval square matrix, and b can be a floating-point or interval vector. The result x is an interval vector which encloses the hull of the solution set of the system $Ax = b$, denoted by $\square\Sigma(A, b)$.

First, `verifylss` computes an approximate floating-point solution $\tilde{x}$ of the system $\texttt{mid}(A)x = \texttt{mid}(b)$ using floating-point iterative refinement. Then it switches to interval arithmetic to compute an enclosure of the error bound upon the floating-point solution. The interval part of `verifylss` comprises two stages. The first stage is based on Krawczyk operator [Rum80, Rum91, Rum05] and the second stage is based on a theorem by Ning & Kearfott [Han92, Roh93, NK97].

Theorem 3.2.1 (Krawczyk)**.** *Let $\mathbf{A} \in \mathbb{IR}^{n\times n}, \mathbf{b} \in \mathbb{IR}^n, R \in \mathbb{R}^{n\times n}, \mathbf{e}^K \in \mathbb{IR}^n$ be given and suppose*

$$R(b - \mathbf{A}\tilde{x}) + (I - R\mathbf{A})\mathbf{e}^K \subseteq \mathbf{e}^K. \tag{3.1}$$

Then for all $\tilde{A} \in \mathbf{A}$ and for all $\tilde{b} \in \mathbf{b}$, $\tilde{A}^{-1}\tilde{b} \in \tilde{x} + \mathbf{e}^K$.

Usually, R is set to an approximate inverse of $\texttt{mid}(\mathbf{A})$. By that way, $I - R\mathbf{A}$ is hopefully close to zero, or centered in zero if R is a good approximation of $\texttt{mid}(\mathbf{A})$.

Furthermore, to compute such an interval vector $\mathbf{e}^K$, `verifylss` uses the epsilon-inflation technique [Rum98, OO09]. Denote by $\mathbf{z} = R(b - \mathbf{A}\tilde{x})$, $\mathbf{C} = I - R\mathbf{A}$, then we get a Krawczyk operator: $\mathbf{z} + \mathbf{C}x = x$. The interval $\mathbf{e}^K$ is initialized to $\mathbf{e}^K_0 = \mathbf{z} + 0.1 * \texttt{rad}(\mathbf{z}) * [-1, 1]$. At each step the fixed-point property (3.1) is checked. If (3.1) is satisfied then the iterations are terminated. Otherwise, $\mathbf{e}^K$ is updated simultaneously by Krawczyk operator and by inflation as follows:

$$\mathbf{e}^K_{i+1} = \mathbf{z} + \mathbf{C} * \left(\mathbf{e}^K_i + 0.1 * \texttt{rad}(\mathbf{e}^K_i) * [-1, 1]\right).$$

The maximum number of iterations is set to 7 in `verifylss`. If it is not possible to obtain such an interval vector $\mathbf{e}^K$ after 7 iterations then `verifylss` switches to the second step to compute a tight enclosure of the solution of the system $(R\mathbf{A})e = \mathbf{z}$ according to

[HS81, Han92, Neu99]. The formula for the interval enclosure of the solution set of interval linear systems of equations, in the case where the coefficient matrix is an H-matrix, is given by the following theorem of Ning and Kearfott [NK97].

Theorem 3.2.2 (Ning-Kearfott [NK97])**.** *Let* $\mathbf{A} \in \mathbb{IR}^{n\times n}$ *be an H-matrix,* $\mathbf{b} \in \mathbb{IR}^n$ *a right-hand side,*

$$u = \langle \mathbf{A} \rangle^{-1} |\mathbf{b}|, \quad d_i = (\langle \mathbf{A} \rangle^{-1})_{ii}$$

and

$$\alpha_i = \langle \mathbf{A}_{ii} \rangle - 1/d_i, \quad \beta_i = u_i/d_i - |\mathbf{b}_i|,$$

where $\langle \mathbf{A} \rangle$ *is the comparison matrix of* $\mathbf{A}$*, which is defined by*

$$\langle \mathbf{A} \rangle_{i,j} = \begin{cases} \mathtt{mig}(\mathbf{A}_{i,j}) & \text{for } i = j, \\ -\mathtt{mag}(\mathbf{A}_{i,j}) & \text{for } i \neq j. \end{cases}$$

Then $\square\Sigma(\mathbf{A}, \mathbf{b})$ *is contained in the vector* $\mathbf{x}$ *with components*

$$\mathbf{x}_i = \frac{b_i + [-\beta_i, \beta_i]}{\mathbf{A}_{ii} + [-\alpha_i, \alpha_i]}.$$

Moreover, if the midpoint of $\mathbf{A}$ *is diagonal then* $\square\Sigma(\mathbf{A}, \mathbf{b}) = \mathbf{x}$*.*

To be able to apply the Ning-Kearfott Theorem requires to compute an upper bound for $\langle \mathbf{A} \rangle^{-1}$, which amounts in fact to compute an upper bound for $|\mathbf{A}^{-1}|$ [Neu90, Section 3.6-3.7, pp104–114]. To that end, according to [Neu99], an approximate inverse of $\mathbf{A}$ must be computed, which costs $2n^3$`FLOPs`. This renders the second stage more costly than the first stage. In return, the second stage allows to obtain a tight enclosure of the solution of the residual system, even for ill-conditioned problems.

In the following sections, we will introduce our proposed method to avoid having to compute a tight enclosure of the solution of the residual system without reducing the result accuracy. In order to obtain an accurate solution, our approach is also based on iterative refinements. Therefore, the floating-point iterative refinements will be briefly recalled prior to explaining how to adapt it to interval arithmetic.

3.3 Floating-point iterative refinement

Iterative refinement is a technique for improving a computed solution $\tilde{x}$ of a linear system $Ax = b$. First, the approximate solution $\tilde{x}$ is computed by some method. Then, the residual $r = b - A\tilde{x}$ of the system for this approximation is computed. The exact error e is the solution of a linear system involving the matrix A, with r as the right-hand side: $Ae = r$. By solving, again approximately, the residual system, a correction term $\tilde{e}$ is obtained and is used to update the floating-point approximation. This method is sketched in Algorithm 3.1 using MatLab notations. In particular, $A \setminus b$ means the solution of the linear system of equations $Ax = b$ computed by some method.

If r and $\tilde{e}$ could be computed exactly, then obviously $A(\tilde{x} + \tilde{e}) = A\tilde{x} + r = b$. Hence, the iteration would converge with just one step. Nevertheless, because of rounding errors, none of them are computed exactly.

In the first versions of this method, floating-point iterative refinement was used with Gaussian elimination. Its convergence conditions are provided in [Mol67, Ske80]. Higham

Algorithm 3.1 Classical iterative refinement

$\tilde{x} = A \setminus b$
while (stopping criterion not verified) **do**
  $r = b - A\tilde{x}$
  $\tilde{e} = A \setminus r$
  $\tilde{x} = \tilde{x} + \tilde{e}$
end while

[Hig97] gives a more thorough and general analysis for a generic solver and for both fixed and mixed computing precision. In fixed precision iterative refinement, the working precision is used for all computations. In mixed precision iterative refinement, residuals are computed in twice the working precision.

First, it is stated in [Mol67, Hig97] that the rates of convergence of mixed and fixed precision iterative refinements are similar. It is also stated in [Mol67] that the computing precision used for residual computations does not materially affect the rate of convergence. However, the computing precision used for residual computations affects the accuracy of results. In this section, only forward errors on $\tilde{x}$, namely the relative error $\|x - \tilde{x}\|_\infty / \|x\|_\infty$ and the componentwise relative error $\| \left((x_i - \tilde{x}_i)/x_i \right) \|_\infty$ are of concern.

Indeed, given a matrix A of dimension n which is not too ill-conditioned, the relative error, after convergence, of the mixed precision iterative refinement is of order $\|x - \tilde{x}\|_\infty / \|x\|_\infty \approx \epsilon$, with ϵ being the relative machine error related to working precision [Hig97]. For fixed precision iterative refinement, the relative error, after convergence, is only of order $\|x - \tilde{x}\|_\infty / \|x\|_\infty \approx 2n\ \texttt{cond}(A, x)\epsilon$: a relative error of order ϵ is no longer ensured. Here $\texttt{cond}(A, x)$ denotes the condition number of the linear system when only A is subject to perturbations [Ske79]. However, this relative error bound is the best that can be obtained without using higher precision. Usually, fixed precision iterative refinement is used to get a stable solution of linear systems, such as in [LD03, LLL+06]. Indeed, only one iteration of iterative refinement with only fixed precision accumulation of the residual suffices to make Gaussian elimination componentwise backward stable [Ske80].

3.4 Interval iterative refinement

The idea of the interval version of iterative refinement is to compute an enclosure of the error term instead of approximating it. This enclosure of the error, added to the approximate solution, yields an enclosure of the exact result. The algorithm proceeds as follows: a floating-point approximation $\tilde{x}$ of the solution is computed first. Then the residual is computed using interval arithmetic: it contains the exact residual. The residual system is now an interval linear system. Hence, the solution $\mathbf{e}$ of this interval linear system contains the exact correction of the floating-point approximation. Thus, $\tilde{x} + \mathbf{e}$ contains the exact solution of the original system. Finally, the floating-point approximate solution $\tilde{x}$ is updated by adding the midpoint of $\mathbf{e}$ to it, meanwhile $\mathbf{e}$ is centered to zero.

Actually, when computing the initial solution, the residual system is usually pre-multiplied by R, a floating-point approximate inverse of A, and that multiplication is performed using interval arithmetic. This operation leads to another interval system:

$$\mathbf{K}\mathbf{e} = \mathbf{z}, \text{ where } \mathbf{K} = [RA], \mathbf{z} = [R\mathbf{r}]. \tag{3.2}$$

Algorithm 3.2 Interval iterative refinement

1: $\tilde{x} = A \setminus b$
2: **while** (stopping criterion not verified) **do**
3: $\quad \mathbf{r} = [b - A\tilde{x}]$ % interval computation
4: $\quad \mathbf{e} = A \setminus \mathbf{r}$
5: $\quad \tilde{x} = \tilde{x} + \texttt{mid}(\mathbf{e})$
6: $\quad \mathbf{e} = \mathbf{e} - \texttt{mid}(\mathbf{e})$
7: **end while**

The goal is to make the iteration contractant and thus to ensure its convergence. However, the system now has an interval matrix and solving such a system is NP-hard [RK95, Roh05].

For $\mathbf{A} \in \mathbb{IR}^{n\times n}, \mathbf{b} \in \mathbb{IR}^n$, the solution of the interval linear system $\mathbf{Ax} = \mathbf{b}$, denoted by $\Sigma(\mathbf{A}, \mathbf{b})$, is the set of all the solutions of all the possible punctual linear systems taken from the interval system:

$$\Sigma(\mathbf{A}, \mathbf{b}) = \{x \in \mathbb{R}^n \quad | \quad \exists A \in \mathbf{A}, \exists b \in \mathbf{b} : Ax = b\}.$$

Usually, $\Sigma(\mathbf{A}, \mathbf{b})$ is not an interval vector. It is not even a convex set.

Example 5. Consider the following example

$$\begin{pmatrix} 2 & [-1,1] \\ [-1,1] & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} [-2,2] \\ 0 \end{pmatrix}. \tag{3.3}$$

The solution of the above system is given by

$$\Sigma(\mathbf{A}, \mathbf{b}) = \{x \in \mathbb{R}^2 \quad | \quad 2|x_2| \leq |x_1|, 2|x_1| \leq 2 + |x_2|\}$$

which is depicted in Figure 3.1 by the grayed out region.

If $\mathbf{A}$ is regular, i.e every $A \in \mathbf{A}$ is regular, then $\Sigma(\mathbf{A}, \mathbf{b})$ is also bounded, since $\mathbf{A}$ and $\mathbf{b}$ are bounded intervals. Hence, the hull of the solution set is defined as follows:

$$\mathbf{A}^H\mathbf{b} = \square\Sigma(\mathbf{A}, \mathbf{b}) = [\texttt{inf}(\Sigma(\mathbf{A}, \mathbf{b})), \texttt{sup}(\Sigma(\mathbf{A}, \mathbf{b}))]. \tag{3.4}$$

The hull of the solution set of the system (3.3) is depicted by the dashed rectangular box in Figure 3.1. Let $\mathbf{A}^{-1}$ denote the matrix inverse of $\mathbf{A}$ which is

$$\mathbf{A}^{-1} = \square\{\tilde{A}^{-1} \quad | \quad \tilde{A} \in \mathbf{A}\}$$

then we have the following inclusion [Neu90, Eq. (6), Section 3.4, p.93]

$$\mathbf{A}^H\mathbf{b} \subseteq \mathbf{A}^{-1}\mathbf{b}, \text{ with equality if } \mathbf{A} \text{ is thin.}$$

Algorithms for solving interval linear systems return a box containing the convex hull of the solution set (3.4), which is thus not the minimal enclosure. Direct algorithms to enclose the solution of an interval linear system exist [Han92, Roh93, Neu99]. Here we prefer an iterative refinement method, which reuses some of the previous floating-point computations. In our case, $\mathbf{e}$ is usually relatively small with respect to the approximate solution of the original system $\tilde{x}$. Hence ideally, the enclosure of the error corresponds to the last bits of $\tilde{x}$, and does not need to be too tight. Indeed, as it will be shown in the next section, such an enclosure of that convex hull can be obtained at a cost of $\mathcal{O}(n^2)$ operations. This complexity is asymptotically negligible compared to the cost of solving the original system using floating-point arithmetic.

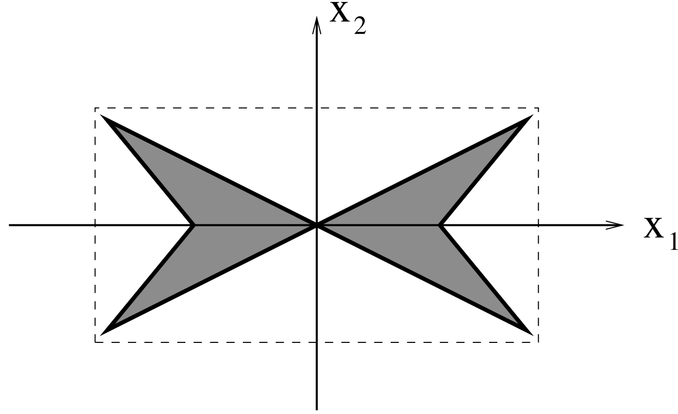


Figure 3.1: Solution set of the interval linear system (3.3).

3.4.1 Numerical behavior

Although switching from floating-point arithmetic to interval arithmetic, in this section we will show that under some conditions, the numerical behavior of $\tilde{x}$ of the interval iterative refinement is similar to that of the floating-point refinement.

Lemma 3.4.1. *Let e^* be the exact solution of the system $Ae = \texttt{mid}(\mathbf{r})$. If $\tilde{e}$ is a solution of the system $A\mathbf{e} = \mathbf{r}$ then $2e^* - \tilde{e}$ is also a solution of the system $A\mathbf{e} = \mathbf{r}$.*

Proof. e^* is the exact solution of the system $Ae = \texttt{mid}(\mathbf{r})$ hence $Ae^* = \texttt{mid}(\mathbf{r})$.

$\tilde{e}$ is a solution of the system $A\mathbf{e} = \mathbf{r}$, which means that there exists a vector $\tilde{r} \in \mathbf{r}$ such that $A\tilde{e} = \tilde{r}$.

Moreover, $\tilde{r} \in \mathbf{r} \Rightarrow |\tilde{r} - \texttt{mid}(\mathbf{r})| \leq \texttt{rad}(\mathbf{r}) \Rightarrow \texttt{mid}(\mathbf{r}) + (\texttt{mid}(\mathbf{r}) - \tilde{r}) \in \mathbf{r} \Rightarrow 2\texttt{mid}(\mathbf{r}) - \tilde{r} \in \mathbf{r}$.

Replace $\texttt{mid}(\mathbf{r})$ and $\tilde{r}$ by Ae^* and $A\tilde{e}$ respectively we get

$$\begin{aligned} 2Ae^* - A\tilde{e} &\in \mathbf{r} \\ \Rightarrow A(2e^* - \tilde{e}) &\in \mathbf{r}. \end{aligned}$$

Therefore $2e^* - \tilde{e}$ is also a solution of the system $A\mathbf{e} = \mathbf{r}$. □

The above lemma means that the exact solution of the system $Ae = \texttt{mid}(\mathbf{r})$ is the midpoint of the exact solution of the system $A\mathbf{e} = \mathbf{r}$. Hence if we can obtain a good enclosure of the solution of the system $A\mathbf{e} = \mathbf{r}$ then

$$\texttt{mid}(A \setminus \mathbf{r}) = e^* \cong A \setminus \texttt{mid}(\mathbf{r}).$$

$\mathbf{r}$ is an enclosure of the residual upon $\tilde{x}$, hence $\mathbf{r}$ is computed by

$$\begin{aligned} \mathbf{r} &= [\bigtriangledown(b + A(-\tilde{x})), \bigtriangleup(b + A(-\tilde{x}))] \\ \Rightarrow \texttt{mid}(\mathbf{r}) &\approx b - A\tilde{x}. \end{aligned}$$

It means that $\texttt{mid}(\mathbf{r})$ is an approximation of $b - A\tilde{x}$.

Therefore, ignoring the interval parts, Algorithm 3.2 can be considered as analogous to the following algorithm

```
x̃ = A \ b
while (stopping criterion not verified) do
  r̂ = b − Ax̃
  ê = A \ r
  x̃ = x̃ + ê
end while
```

One can see that the numerical behavior of $\tilde{x}$ in the above algorithm is similar to that of the floating-point iterative refinement.

3.4.2 Updating the computed solution

After each step the approximate solution is updated, and also the error bound on it (lines 5 and 6 of Algorithm 3.2). The purpose of this update is to ensure that $\tilde{x} + \mathbf{e}$ always contains the exact solution in its interior. It is worth noting here that, using floating-point arithmetic, all the punctual operations are subject to rounding errors. Thus to have a correctly updated solution, interval operations must be used. Lines 5 and 6 of Algorithm 3.2 should be replaced by the following snippet.

```
x̃' = x̃ + mid(e)
e = e − [x̃' − x̃]
x̃ = x̃'
```

Using $[\tilde{x}' - \tilde{x}]$ instead of $\mathtt{mid}(\mathbf{e})$ seems to inflate the interval a little bit. In fact, if $\mathtt{mid}(\mathbf{e})$ is small in magnitude with respect to $\tilde{x}$, i.e $|\mathtt{mid}(\mathbf{e})| \leq 1/2|\tilde{x}|$ which is often the case, then $\tilde{x}$ and $\tilde{x}'$ have the same sign and even satisfy $1/2\tilde{x} \leq \tilde{x}' \leq 2\tilde{x}$, thus following the Sterbenz lemma [Ste74], $\tilde{x}' - \tilde{x}$ is exact in floating-point arithmetic.

Moreover, the following inclusion holds

$$\tilde{x}' + (\mathbf{e} - [\tilde{x}' - \tilde{x}]) \supseteq \tilde{x}' + (\mathbf{e} - \tilde{x}' - \tilde{x}) = \tilde{x} + \mathbf{e}.$$

Hence the inclusion property is ensured by this updating process. This updated error bound will serve as the starting point for the next step, or as the initial solution for the next refinement: there is no need to recompute it.

Still, to start the refinement, an initial solution of the residual system is needed. In the following section, we explain how to compute an initial solution of an interval linear system.

3.4.3 Initial solution

The determination of an initial enclosure of the solution is a main issue in the refinement method. The entire real line could be considered as an initial enclosure of each component, but formulae shown in Section 3.4.4 cannot refine it.

The necessary condition to be able to compute an initial enclosure of the interval linear system $\mathbf{Ke} = \mathbf{z}$ is that $\mathbf{K}$ is invertible, i.e. each real matrix $K \in \mathbf{K}$ is invertible. Nevertheless, checking the invertibility of an interval matrix is NP-hard [Roh96, Roh05].

However, if some sufficient condition is satisfied, an initial enclosure of the solution of an interval linear system can be computed. For this purpose, we apply the following proposition to the original, real, matrix A and to the interval right-hand side $\mathbf{r}$.

Proposition 3.4.2 ([Neu90, Prop. 4.1.9, p.121]). *Let $\mathbf{A} \in \mathbb{IR}^{n\times n}$ be an interval matrix of dimensions $n \times n$ and let $C, C' \in \mathbb{R}^{n\times n}$.*

1. *If $C\mathbf{A}C'$ is an H-matrix then, for all $\mathbf{b} \in \mathbb{IR}^n$, we have*

$$|\mathbf{A}^H\mathbf{b}| \leq |C'|\langle C\mathbf{A}C'\rangle^{-1}|C\mathbf{b}|,$$

2. *if $\langle C\mathbf{A}C'\rangle u \geq v > 0$ for some $u \geq 0$ then*

$$\begin{aligned} |\mathbf{A}^H\mathbf{b}| &\leq \|C\mathbf{b}\|_v \cdot |C'|u, \\ \mathbf{A}^H\mathbf{b} &\subseteq \|C\mathbf{b}\|_v \cdot |C'|[-u, u], \end{aligned}$$

where $\mathbf{A}^H\mathbf{b}$ is the convex hull of the solution set of the system $\mathbf{Ax} = \mathbf{b}$, $\langle\mathbf{A}\rangle$ is the comparison matrix of $\mathbf{A}$, whose components are defined by:

$$\begin{aligned} \langle\mathbf{A}\rangle_{i,i} &= \min(|a_{i,i}|, a_{i,i} \in \mathbf{A}_{i,i}), \\ \langle\mathbf{A}\rangle_{i,j} &= -\max(|a_{i,j}|, a_{i,j} \in \mathbf{A}_{i,j}) \text{ for } j \neq i, \end{aligned}$$

and $\|\mathbf{b}\|_v$ is the scaled norm with respect to v: $\|\mathbf{b}\|_v = \max(|\mathbf{b}_i|/v_i) \quad i \in 1, \dots, n$.

Following this proposition, a sufficient condition to be able to compute an initial solution is that we exhibit C and C' such that $C\mathbf{A}C'$ is an H-matrix. In our algorithm, we use $C = R$, with R an approximate inverse of A, and $C' = I$. If R is a good approximation of the inverse of A then $CAC' = RA \approx I$ is an H-matrix. We also need to exhibit u and v as in the second part of the proposition, to get $\mathbf{e} = \|R\mathbf{r}\|_v[-u, u]$ as an initial enclosure of the system $A\mathbf{e} = \mathbf{r}$.

Because all computations are performed using floating-point (as opposed to exact) arithmetic, it is necessary to compute RA using interval arithmetic in order to get a guaranteed result. So the previous proposition is modified as shown below.

Proposition 3.4.3. *Let $A \in \mathbb{F}^{n\times n}$ and $R \in \mathbb{F}^{n\times n}$ be a floating-point approximate inverse of A. If $\langle[RA]\rangle u \geq v > 0$ for some $u \geq 0$ then:*

$$\begin{aligned} |A^{-1}\mathbf{r}| &\leq \|R\mathbf{r}\|_v u, \\ A^{-1}\mathbf{r} &\subseteq \|R\mathbf{r}\|_v \cdot [-u, u]. \end{aligned}$$

What remains now is to find a positive vector u so that $\langle[RA]\rangle u > 0$. Using [Neu90, Prop. 3.7.3, p.112], u is set to an approximate solution $\tilde{u}$ of the linear system $\langle[RA]\rangle u = e$, where $e = (1, \dots, 1)^T$. Because two important properties of an H-matrix $\mathbf{A}$ are that $\langle\mathbf{A}\rangle$ is regular, and $\langle\mathbf{A}\rangle^{-1}e > 0$, $\tilde{u}$ satisfies $\tilde{u} \approx \langle[RA]\rangle^{-1}e > 0$ and $\langle[RA]\rangle\tilde{u} \approx e > 0$, for sufficiently good approximations $\tilde{u}$.

In our case, A is a floating-point matrix. If A is well preconditioned, or more precisely if RA is close to identity, or diagonally dominant, then it suffices to use the vector $(1, \dots, 1)^T$ as the value of u and the product $\langle[RA]\rangle u$ is positive. If the test of positivity of $\langle[RA]\rangle u$ fails with $u = (1, \dots, 1)^T$, then $\langle[RA]\rangle$ is not a diagonally dominant matrix. However it

does not mean that $\langle [RA] \rangle$ is not an H-matrix. In this case, we apply some floating-point Jacobi iterations in order to compute an approximate solution of the system $\langle [RA] \rangle u = e \equiv (1, \ldots, 1)^T$. Let D, E be the diagonal and off-diagonal parts of $\langle [RA] \rangle$. Jacobi iterations can be expressed by the following formula, where u is the previous iterate, u' is the new iterate, and v is the product between $\langle [RA] \rangle$ and u:

$$u' = D^{-1}(e - Eu) = u + D^{-1}(e - (D + E)u) = u + D^{-1}(e - v).$$

Let $K = \langle [RA] \rangle$ be the comparison matrix of $[RA]$, the algorithm to determine u is detailed in Algorithm 3.3.

Algorithm 3.3 Testing the H-matrix property

$D = \mathtt{diag}(K)$
$e = (1, \ldots, 1)^T$
$u = e$
$v = K * u$
$nbiter = 10$
$i = 0$
while ($\mathtt{any}(v \leq 0)$ && $i < nbiter$) **do**
  $i = i + 1$
  $u = \mathtt{abs}(u + (e - v)./D)$
  $v = K * u$
end while

Note that, in order to ensure that u is a non-negative vector, we take the absolute value of the updated solution after each refinement. Moreover, as it is not necessary to compute an accurate solution, the refinement stops as soon as $Ku > 0$.

If the Jacobi iterative refinement fails to obtain a vector $u \geq 0$ such that $v = K * u > 0$ after 10 iterations, then our algorithm stops and issues a message of failure. Each Jacobi refinement costs only $\mathcal{O}(n^2)$ floating-point operations, hence the overhead of execution time introduced by this refinement is negligible.

Nevertheless, Proposition 3.4.3 does not yield a tight enclosure of the solution. The next section presents methods to improve this interval enclosure.

3.4.4 Interval iterative improvement

There exists several methods of interval iterative improvement, such as the methods of Jacobi and of Gauss-Seidel (introduced below), or the method of Krawczyk. Krawczyk method has the quadratic approximation property [Neu90, Section 2.3, p.56], but Gauss-Seidel iteration always yields tighter intervals than Krawczyk iteration, when applied to a preconditioned system [Neu90, Theorem 4.3.5, p.138].

The essence of interval iterative improvement methods is to compute an improved enclosure of the solution from a given initial enclosure of the solution. Given an initial enclosure $\boldsymbol{e}$ to the interval linear system $\boldsymbol{K}\boldsymbol{e} = \boldsymbol{z}$, an improved approximate enclosure is obtained by writing the linear system satisfied by

$$\tilde{e} \in \boldsymbol{e} : \exists \tilde{K} \in \boldsymbol{K}, \exists \tilde{z} \in \boldsymbol{z} \; : \; \tilde{K}\tilde{e} = \tilde{z}.$$

Developing the i-th line and separating the i-th component, one gets:

$$\tilde{e}_i = \left(\tilde{z}_i - \sum_{j=1}^{i-1} \tilde{K}_{i,j}\tilde{e}_j - \sum_{j=j+1}^{n} \tilde{K}_{i,j}\tilde{e}_j \right) / \tilde{K}_{i,i} \text{ and } \tilde{e}_i \in \boldsymbol{e}_i.$$

Replacing punctual terms by the corresponding interval terms yields the formula of the interval Jacobi iteration:

$$\tilde{e}_i \in \boldsymbol{e}'_i := \left(\boldsymbol{z}_i - \sum_{j=1}^{i-1} \boldsymbol{K}_{i,j}\boldsymbol{e}_j - \sum_{j=j+1}^{n} \boldsymbol{K}_{i,j}\boldsymbol{e}_j \right) / \boldsymbol{K}_{i,i} \cap \boldsymbol{e}_i.$$

Taking into account the fact that for components that have already been refined, $\tilde{e_j}$ belongs to both original value $\boldsymbol{e}_j$ and refined valued $\boldsymbol{e}'_j$, $\boldsymbol{e}_j$ can be replaced by $\boldsymbol{e}'_j$ for $j < i$ to obtain the Gauss-Seidel iteration:

$$\boldsymbol{e}'_i := \left(\boldsymbol{z}_i - \sum_{j=1}^{i-1} \boldsymbol{K}_{i,j}\boldsymbol{e}'_j - \sum_{j=j+1}^{n} \boldsymbol{K}_{i,j}\boldsymbol{e}_j \right) / \boldsymbol{K}_{i,i} \cap \boldsymbol{e}_i. \tag{3.5}$$

Taking the intersection with the former iterate $\boldsymbol{e}_i$ implies the contracting property of both Jacobi and Gauss-Seidel iterations. Hence, both iterations converge. Nevertheless, making full use of the refined values, Gauss-Seidel iterations converge much more quickly than Jacobi iterations. As mentioned in Section 3.4.3, our sufficient condition to compute an initial solution of the interval residual system is that $[RA]$ is an H-matrix. Under this condition, Gauss-Seidel iterations converge very quickly.

Let us now detail the complexity of an iteration. For both Jacobi and Gauss-Seidel iterations, the improvement of each component requires $n - 1$ interval multiplications, n interval additions and one interval division. Thus, in total, each Jacobi/Gauss-Seidel iteration costs $\mathcal{O}(n^2)$ interval operations. Hence, theoretically, the improvement stage should not affect the overall cost of the method, which is $\mathcal{O}(n^3)$.

3.4.5 Detailed algorithm

The complete algorithm we propose is given below, it uses all the building blocks introduced above.

Complexity

The complexity of the whole algorithm is determined by counting the number of operations for each step. To be more precise, we distinguish here several types of operations, namely:

- `FLOP` a floating-point operation in general,
- `FMUL` a floating-point multiplication,
- `FADD` a floating-point addition,
- $\mathbb{I}$`OP` an interval operation in general,
- $\mathbb{I}$`MUL` an interval multiplication,
- $\mathbb{I}$`ADD` an interval addition.

Algorithm 3.4 Solving and verifying a linear system

1: Compute the LU decomposition of A: $A = L \times U$
2: Compute an approximation $\tilde{x}$ with a forward and a backward substitution using L and U
3: Compute R, an approximate inverse of A, by solving $RL = \texttt{inv}(U)$ [Hig02, Chapter 14]
4: Pre-multiply A by R: $\boldsymbol{K} = [RA]$
5: Test if $\boldsymbol{K}$ is an H-matrix by computing a non-negative vector u such that $\langle \boldsymbol{K} \rangle u \geq v > 0$
6: **if** (fail to compute u) **then**
7: **print** *Failed to verify the solution. The system may be either singular or too ill-conditioned.*
8: **return**
9: **end if**
10: Compute the residual $\boldsymbol{r} = [b - A\tilde{x}]$ in double the working precision
11: Pre-multiply the residual by R: $\boldsymbol{z} = R\boldsymbol{r}$
12: Compute an initial error bound $\boldsymbol{e} = \|\boldsymbol{z}\|_v \cdot [-u, u]$
13: **while** (not converged) **do**
14: Apply 5 Gauss-Seidel iterations on $\boldsymbol{K}, \boldsymbol{z}$, and $\boldsymbol{e}$
15: Update $\tilde{x}$ and $\boldsymbol{e}$
16: Recompute $\boldsymbol{r}$ and $\boldsymbol{z}$
17: **end while**

The LU decomposition of a floating-point matrix costs $4/3n^3$ FLOPs.

Forward and backward substitution using upper and lower triangular matrices cost $n^2 + \mathcal{O}(n)$ FLOPs each [Hig02, chap.8].

Matrix inversion using LU decomposition costs $8/3n^3$ FLOPs.

Pre-multiplying A by R, assuming that we use directed rounding modes, requires two floating-point matrix multiplications, one for the upper bound and the other for the lower bound. Hence in total the preconditioning requires $4n^3$ FLOPs.

Computing the comparison matrix of $\mathbf{K}$ requires $2n^2 + \mathcal{O}(n)$ FLOPs.

Testing if $\langle \mathbf{K} \rangle$ is diagonally dominant is equivalent to a matrix-vector operation. Hence it costs $2n^2 + \mathcal{O}(n)$ FLOPs.

Computing the residual in interval arithmetic requires computing the residual twice in floating-point arithmetic, once in downward rounding mode and once in upward rounding mode. Moreover, to increase the result accuracy, the residual is computed in twice the working precision. Therefore, we will count it by $2n^2 + \mathcal{O}(n)$ $\texttt{FLOP}_2\texttt{s}$, where $\texttt{FLOP}_2$ denotes a floating-point operation in twice the working precision.

Pre-multiplying the residual is a floating-point matrix-interval vector multiplication, which costs $2n^2 + \mathcal{O}(n)$ FLOPs.

Computing the scaled norm of a vector costs only $\mathcal{O}(n)$ FLOPs.

For each refinement iteration:

- each Jacobi/Gauss-Seidel iteration costs n^2 $\mathbb{I}$MULs $+ n^2$ $\mathbb{I}$ADDs $+ \mathcal{O}(n)$ $\mathbb{I}$OPs.
- the updating process requires only $\mathcal{O}(n)$ FLOPs.
- recomputing the residual system requires $(2n^2 + \mathcal{O}(n))$ $\texttt{FLOP}_2\texttt{s}$ $+ (2n^2 + \mathcal{O}(n))$ FLOPs.

As will be shown in section 3.5.1, a floating-point operation in twice the working precision can be implemented by some floating-point operations in the working precision, i.e. $1\texttt{FLOP}_2 = \mathcal{O}(1)\texttt{FLOPs}$. One can see that the complexity of the algorithm is $8n^3 + \mathcal{O}(n^2)\texttt{FLOPs} + \mathcal{O}(n^2)\mathbb{I}\texttt{OPs}$. Meanwhile, to solve a linear system using floating-point arithmetic costs $4/3n^3 + \mathcal{O}(n^2)\texttt{FLOPs}$. Therefore, the theoretical slowdown factor of solving a linear system in interval arithmetic with respect to floating-point arithmetic is 6.

Stopping criteria

As mentioned in Section 3.4, the width of the interval error decreases after each step. So it is a non-negative non-increasing series. This property leads to two stopping criteria (for a study of stopping criteria, see [DHK+06]).

Firstly, we stop the computation whenever reaching a required number of accurate bits. For example, working with double floating-point numbers, there is no point of getting a result which is accurate to more than 52 bits. Using interval arithmetic, it is quite easy to compute the number of exact bits in the result via componentwise relative errors:

$$\texttt{nb_bits} = -\texttt{log}_2 \left(\max \left(\frac{\texttt{rad}\,(\boldsymbol{e}_i)}{|\tilde{x}_i|} \right) \right). \tag{3.6}$$

Nevertheless, the program can end up without reaching the required number of exact bits. Hence, we have to detect whether the iteration yields extra correct digits or stagnates. Let $\boldsymbol{e}$ and $\boldsymbol{e}'$ be two successive iterates, a second stopping criterion is that no more correct digit is obtained in any component of the approximation:

$$\max_j \left(\frac{\left|\texttt{rad}\ (\boldsymbol{e}'_j) - \texttt{rad}\ (\boldsymbol{e}_j)\right|}{|\tilde{x}_j^{(i)}|} \right) < \epsilon, \tag{3.7}$$

with ϵ being the machine relative error, which is 2^{-53} for IEEE 754 binary double precision format.

In cases where none of these two criteria above is matched, it is necessary to stop the computation when one has tried enough. Practically, in our implementations, the maximum number of iterations is set to 10. Also, the maximum number of Gauss-Seidel iterations is set to 5 due to its quick convergence property.

3.5 Numerical experiments

In what follows, the implementation in MatLab, using the interval library IntLab, of Algorithm 5.1, is called `certifylss`. The computing precision used for all computations is the IEEE double floating-point precision, except for the residual computation, which is computed in twice the working precision. The matrices used in the tests are generated by the MatLab function `gallery` $('randsvd', dim, cond)$, where dim is the matrix dimension (taken as 1000), and $cond$ is the condition number, which varies between 2^5 and 2^{50}.

3.5.1 Implementation issues

Higher precision for the residual

At the moment, almost no processor supports native quadruple precision. Hence, in order to obtain a higher precision for the residual computation, we use simulated double-double

precision which is based on error-free transforms according to [Dek71, HLB01, LDB+02, ORO05, LL08b, LL08a, Rum09b, Rum09a, NGL09, GNL09].

More precisely, it uses the the error-free transforms for the sum and for the product of two floating-point numbers. The error-free transform of the sum of two floating-points, called `TwoSum`, costs 6`FADDs`. Moreover, the error-free transform of the product of two floating-points, called `TwoProd`, costs 10`FADDs` and 7`FMULs` unless an operation of type Fused-Multiply-Add is available [BM05]. Hence the cost of computing the residual, as accounted by $2n^2 + \mathcal{O}(n)\texttt{FLOP}_2s$ in the previous section, using simulated double-double precision will be translated into `FLOPs` by $7n^2$`FMULs` $+ 16n^2$`FADDs`. Hence, in theory, this cost can still be considered as $\mathcal{O}(n^2)$`FLOPs`.

```
1  function res = residual(A, x, b)
2      setround(0)
3      factor = 2^27 + 1;

5      % split vector -x into two parts x1 and x2
6      % x1 corresponds to the first half of the mantissa, of high
           weight
7      % x2 corresponds to the second half of the mantissa, of low
           weight
8      x = -x;
9      y = factor*x;
10     xbig = y - x;
11     x1 = y - xbig;
12     x2 = x - x1;

14     n = size(A,1);


17     r = b;

19     er2 = zeros(n,1);
20     er1 = zeros(n,1);
21     for i = 1:n
22         setround(0);
23         % split a row of A
24         a = A(:, i);
25         a2 = a * factor;
26         abig = a2 - a;
27         a1 = a2 - abig;
28         a2 = a - a1;

30         % EFT for the product
31         p_h = a * x(i);
32         p_l = a2 * x2(i) - (((p_h - a1 * x1(i) ) - a2 * x1(i)) - a1
               * x2(i));

34         % EFT for the sum of high parts of the products
35         s = r + p_h;
36         v = s - r;
37         e = (r - (s - v)) + (p_h - v);
38         r = s;

40         setround(1);
41         er2 = er2 + e + p_l;
```

```
42            er1 = (er1 - e) - p_l;
43        end

45        res = r + infsup(-er1,er2);
46        end
47    end
```

However, it is much worse in practice. The implementation of error-free transforms needs separate access to columns and rows of matrix, which is disadvantageous in MatLab. That causes the residual computation to be very slow. For example, on a computer Intel(R) Core(TM)2 Quad CPU Q9450 of frequency 2.66GHz, a product between two matrices of dimensions 1000×1000 takes 0.376557 seconds, a product between a matrix of dimensions 1000×1000 and a vector of 1000 elements takes 0.001773 seconds. Meanwhile the residual computed by function `residual` with the same dimensions takes 0.195453 seconds, which is roughly half the execution time of matrix multiplication.

Gauss-Seidel vs Jacobi

As mentioned in Section 3.4.5, each Gauss-Seidel iteration as well as each Jacobi iteration costs $n^2\mathbb{I}\texttt{MULs} + n^2\mathbb{I}\texttt{ADDs} + \mathcal{O}(n)\mathbb{I}\texttt{OPs}$. Nevertheless, making full use of refined values, Gauss-Seidel iterations converge more quickly than Jacobi iterations. In implementation, this observation is still true in terms of number of iterations. On the other hand, in terms of execution time, this observation does not always hold. That is because of parallelism and memory access problem in MatLab. Each Jacobi iteration can be translated directly into a matrix-vector multiplication. Hence its performance relies totally on the underlying library. Meanwhile, a Gauss-Seidel iteration is translated into a series of dot products, which reduces the possibility of parallelism. Moreover, accessing rows individually in MatLab is disadvantageous. Hence, in an implementation in MatLab, a Gauss-Seidel iteration is much slower than a Jacobi iteration.

Therefore, for problems of not too large dimensions, Jacobi iterations are preferable over Gauss-Seidel iterations in MatLab. Its slower convergence speed with respect to Gauss-Seidel iterations can be compensated by augmenting the number of iterations, and still gaining in execution time. This phenomenon has been observed in [BT89, chap.2, Section 2.4 and 2.5, pp.130–142] for parallel implementation of classical iterative methods.

3.5.2 Experimental results

Figure 3.2 depicts results computed by our `certifylss` function, by the function `verifylss` of the IntLab library with the residual being computed in quadruple precision, and non-verified results computed by MatLab. Using higher precision for residual computation, verified results provided by the two verified functions: `certifylss` and `verifylss` are much more accurate than non-verified results provided by MatLab, at the price of a higher execution time. In this experimental set, the execution time of a call to the MatLab's function for computing an approximate solution is about 0.1s.

When the coefficient matrix is not too ill-conditioned, i.e. less than 2^{37} in these experiments, both verified functions provide the same accuracy (52 bits). It means that both verified functions succeed to verify the last bit of the approximate solution. Nevertheless, `certifylss` is better than `verifylss` in terms of execution time.

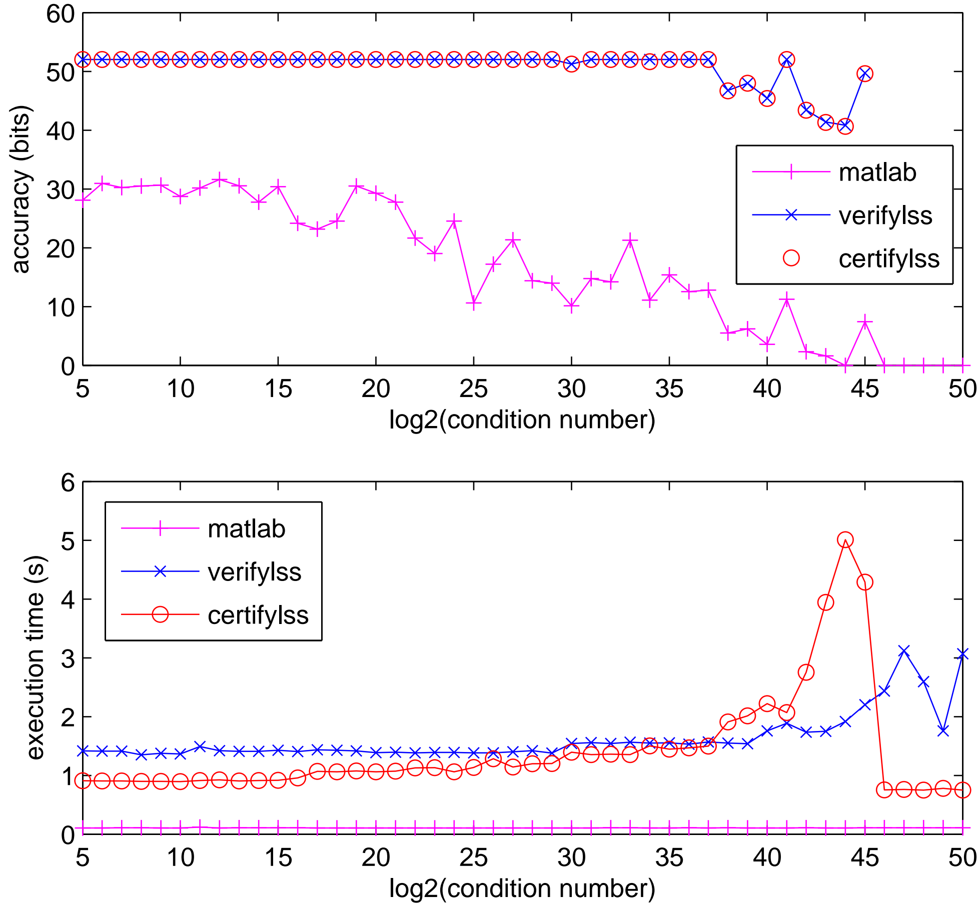


Figure 3.2: MatLab implementation results for 1000×1000 matrices.

When the condition number increases, an accuracy of 52 bits cannot be obtained any more. However, the execution time for `certifylss` becomes higher than the execution time for `verifylss`. This is explained by the fact that `verifylss` uses only floating-point iterations to refine the approximate solutions, while `certifylss` uses alternately floating-point iterations and interval iterations to refine the approximate solutions, and at the same time the error bounds. As the condition number increases, the number of iterations also increases. Interval operations are slower than floating-point operations, hence, the execution time of `certifylss` increases faster than the execution time of `verifylss` along with the increase of the number of iterations.

When the coefficient matrix is not too ill-conditioned, the factor of execution time of `certifylss` with respect to the non-verified function of MatLab is about 8. Nevertheless, when the condition number increases, this theoretical factor is no longer satisfied. It is not only that interval operations are slow, but also that as the condition number increases, the number of iterations as well as the time of recomputing the residual increase. As explained in the previous section, residual computing is costly and it introduces a significant overhead of execution time to both `certifylss` and `verifylss` functions.

When the condition number gets close to $1/\epsilon$, both functions fail to verify the solution. Indeed, a good approximation of the inverse of A cannot be obtained, thus the sufficient condition that RA is an H-matrix does not hold.

3.6 Conclusion

Combining floating-point iterative refinement and interval improvement, we are able to provide accurate results together with a tight error bound upon the approximate solutions. This error bound, which is indeed an interval vector, enables us to state how many guaranteed bits there are in the approximate solutions. When the condition number of the system is small to moderate, our method succeeds in verifying up to the last bit of the approximate solution. It means that the computed solution is the exact solution rounded to a floating-point vector in faithful rounding mode.

Using alternately floating-point iterative refinement and interval improvement also enables us to detect easily when to terminate the refinements, by counting the number of guaranteed bits as well as by checking improvement in the width of the error bound. Moreover, our method also checks the feasibility before performing any further refinement. Therefore our method exhibits better performance than Rump's algorithm when the matrix is either well-conditioned or too ill-conditioned.

On the other hand, when the matrix is ill-conditioned, the computation time of our method increases very fast and becomes much slower than Rump's algorithm. That is because the proposed method uses intensively interval operations. The next chapter will present a technique to overcome this problem.

Chapter 4

Relaxed method to verify the solution of a linear system

This chapter presents an improvement of the method presented in Chapter 3. Due to the intensive use of interval operations, the execution time of the function `certifylss` *increases substantially when the condition number of the coefficient matrix increases. We introduce a relaxation technique to reduce this effect. This technique consists in inflating the matrix of the pre-multiplied system in such a way that it is centered in a diagonal matrix. Then a product of that inflated matrix with another interval vector is equivalent to a floating-point matrix vector product in terms of execution time. The relaxed version of the function* `certifylss`*, also implemented using the IntLab library, is called* `certifylss_relaxed`*. Numerical results show that, when applied to the residual system, the relaxation technique helps to reduce significantly the execution time of the method without deteriorating the accuracy of the final results.*

4.1 Introduction

The interval improvement step is used to improve the error bound upon a computed approximation. Nevertheless, an intensive use of interval operations makes the interval improvement costly in terms of execution time. Especially when the condition number of the system is high, the number of iterations is high. The overhead of execution time introduced by the interval improvement becomes significant and even larger than the execution time of computing a tight error bound enclosure according to [Neu99].

In this chapter, we introduce a technique, which we shall call the *relaxation technique*, to reduce the overhead of execution time introduced by the interval improvement. By nature, the error bound (i) is an enclosure of the difference between the approximation and the exact solution and hence (ii) is used to update the approximation. This error bound should correspond to lost bits in the approximation. Hence, it is not necessary to compute an error bound with high accuracy. Thus we relax the tightness requirement on this error bound to speed up the program.

4.2 Interval improvement: Jacobi and Gauss-Seidel iterations

Let $\mathbf{A} \in \mathbb{IR}^{n\times n}$ be an H-matrix, $\mathbf{b} \in \mathbb{IR}^n$, and $\mathbf{e}^0$ be an initial enclosure of the solution of the interval linear system

$$\mathbf{A}\mathbf{e} = \mathbf{b}. \tag{4.1}$$

The essence of interval improvement is to compute an improved enclosure of the solution from $\mathbf{e}^0$

$$\mathbf{e}' = \{e \in \mathbf{e}^0 \quad | \quad \exists A \in \mathbf{A} \quad \exists b \in \mathbf{b} : Ae = b\}.$$

Let $\mathbf{D}, \mathbf{L}, \mathbf{U}$ be respectively the diagonal, the lower triangular and the upper triangular parts of $\mathbf{A}$, then $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$. Jacobi iteration can be expressed by

$$\mathbf{e}' = \mathbf{D}^{-1}(\mathbf{b} - (\mathbf{L} + \mathbf{U})\mathbf{e}^0).$$

Likewise, Gauss-Seidel iteration can also be expressed symbolically in the same way

$$\mathbf{e}' = \mathbf{D}^{-1}(\mathbf{b} - (\mathbf{L}\mathbf{e}' + \mathbf{U}\mathbf{e}^0)).$$

$\mathbf{D}$ is a diagonal matrix, its inverse is also a diagonal matrix whose diagonal elements are the inverse of their corresponding elements of $\mathbf{D}$. Hence, the most expensive part of Jacobi or Gauss-Seidel iterations is the interval matrix-vector multiplication by $\mathbf{L}$ and $\mathbf{U}$. Even if we use Rump's fast algorithm for interval matrix multiplication, the theoretical factor between an interval matrix vector product in terms of execution time with respect to a floating-point matrix vector product of the same size is 4. However, if we take into account the execution time for transforming data, for example computing midpoint and radius of the matrix, then the practical factor is even higher.

When applied to a preconditioned system, $\mathbf{A}$ is close to the identity, hence $\mathbf{L}, \mathbf{U}$ are close to zero. As it has been observed in Chapter 2 Eq. (2.4), an interval multiplication between a centered-in-zero matrix and another interval matrix or vector is equivalent to its floating-point counterpart in terms of execution time. Hence, to gain in performance, the matrix of the system is enlarged, so as to have its off-diagonal elements centered in 0. Formulas for subsequent operations are thus simplified: the computed intervals are symmetrical around 0 and only one endpoint is computed, using floating-point arithmetic. More precisely, we will work with

$$\begin{cases} \hat{\mathbf{L}} &= [-|L|, |L|], \\ \hat{\mathbf{U}} &= [-|U|, |U|]. \end{cases} \tag{4.2}$$

Lemma 4.2.1. *For* $\mathbf{x} \in \mathbb{IR}$, *let* $\hat{\mathbf{x}} = [-|x|, |x|]$ *then*

$$\mathbf{x} \subseteq \hat{\mathbf{x}} \quad \textit{and} \quad |\mathbf{x}| = |\hat{\mathbf{x}}|.$$

We will call $\hat{\mathbf{x}}$ *the relaxed version of* $\mathbf{x}$.

Proof. From the definition of `mag`, for all $x \in \mathbf{x}$ then

$$|x| \leq |\mathbf{x}| \Rightarrow -|\mathbf{x}| \leq x \leq |\mathbf{x}| \Rightarrow \mathbf{x} \subseteq \hat{\mathbf{x}}.$$

It is obvious that $|\hat{\mathbf{x}}| = |\mathbf{x}|$. □

Hence, $\mathbf{L} \subseteq \hat{\mathbf{L}}$ and $\mathbf{U} \subseteq \hat{\mathbf{U}}$. Denote $\hat{\mathbf{A}} = \hat{\mathbf{L}} + \mathbf{D} + \hat{\mathbf{U}}$, then $\mathbf{A} \subseteq \hat{\mathbf{A}}$, we replace the original system by the relaxed system

$$\hat{\mathbf{A}}\mathbf{e} = \mathbf{b}. \tag{4.3}$$

Proposition 4.2.2. *Let* $\mathbf{A}, \mathbf{A}' \in \mathbb{IR}^{n\times n}$ *be two regular matrices. Let* $\mathbf{b} \in \mathbb{IR}^n$ *be an interval vector. If* $\mathbf{A} \subseteq \mathbf{A}'$ *then* $\Sigma(\mathbf{A}, \mathbf{b}) \subseteq \Sigma(\mathbf{A}', \mathbf{b})$.

Proof. For all $\tilde{x} \in \Sigma(\mathbf{A}, \mathbf{b})$, then

$$\exists \tilde{A} \in \mathbf{A}, \exists \tilde{b} \in \mathbf{b} \quad \text{such that} \quad \tilde{A}\tilde{x} = \tilde{b}.$$

As $\mathbf{A} \subset \mathbf{A}'$, $\tilde{A} \in \mathbf{A}'$. It implies that $\tilde{x} \in \Sigma(\mathbf{A}', \mathbf{b})$. Therefore $\Sigma(\mathbf{A}, \mathbf{b}) \subseteq \Sigma(\mathbf{A}', \mathbf{b})$. □

From Proposition 4.2.2, the solution of the original system (4.1) is included in the solution of the relaxed system (4.3).

Example 6. Consider the following interval linear system

$$\begin{pmatrix} 4 & [-0.5, 1] \\ [-0.5, 1] & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \end{pmatrix}.$$

The relaxed version of this system is

$$\begin{pmatrix} 4 & [-1, 1] \\ [-1, 1] & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \end{pmatrix}. \tag{4.4}$$

The solution set of the original system is depicted in Figure 4.1 by the grayed out region. Meanwhile, the solution set of the relaxed system is depicted by the white region.

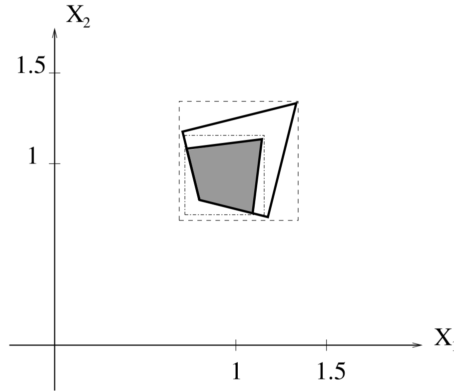


Figure 4.1: Solution set of the relaxed system.

One can see that the solution set of the relaxed system overestimates the solution set of the original system. Nonetheless, if the off-diagonal elements of the original system are almost centered in zero then the grayed out region is close to the white region.

Let us apply the interval improvement to this relaxed system.

The relaxed Jacobi iterations are

$$\mathbf{e}' = \mathbf{D}^{-1}(\mathbf{b} - (\hat{\mathbf{L}} + \hat{\mathbf{U}})\mathbf{e}^0).$$

The relaxed Gauss-Seidel iterations are

$$\mathbf{e}' = \mathbf{D}^{-1}(\mathbf{b} - (\hat{\mathbf{L}}\mathbf{e}' + \hat{\mathbf{U}}\mathbf{e}^0)).$$

Since $\hat{\mathbf{L}}, \hat{\mathbf{U}}$ are centered in zero, $\hat{\mathbf{L}} + \hat{\mathbf{U}}$ is also centered in zero. According to Section 2.3.1, the product between $\hat{\mathbf{L}}, \hat{\mathbf{U}}$ or $\hat{\mathbf{L}} + \hat{\mathbf{U}}$ with any interval vector can be computed by just one punctual matrix-vector product. For $\mathbf{x} \in \mathbb{IR}^n$:

$$\begin{aligned}
\hat{\mathbf{L}}\mathbf{x} &= [-|\mathbf{L}||\mathbf{x}|, |\mathbf{L}||\mathbf{x}|]\,,\\
\hat{\mathbf{U}}\mathbf{x} &= [-|\mathbf{U}||\mathbf{x}|, |\mathbf{U}||\mathbf{x}|]\,,\\
(\hat{\mathbf{L}} + \hat{\mathbf{U}})\mathbf{x} &= [-|\mathbf{L} + \mathbf{U}||\mathbf{x}|, |\mathbf{L} + \mathbf{U}||\mathbf{x}|]\,.
\end{aligned}$$

Therefore the execution time of applying the interval improvement to the relaxed system is much smaller than applying it to the original system. In theory, the factor of execution time between an interval matrix-vector product with respect to a punctual matrix-vector product is 4. The relaxed interval iterations also exhibit some advantages over the original interval iterations regarding storage memory. As $\hat{\mathbf{A}}$ is a matrix whose off-diagonal components are centered in zero, to store $\hat{\mathbf{A}}$, it is only needed to store its upper bound as a floating-point matrix, and its diagonal as an interval vector.

4.2.1 Convergence rate

Let us consider the case of Jacobi iterations

$$\mathbf{e}' = \mathbf{D}^{-1}(\mathbf{b} - (\hat{\mathbf{L}} + \hat{\mathbf{U}})\mathbf{e}^0).$$

Since $\hat{\mathbf{L}} + \hat{\mathbf{U}}$ is centered in zero, then $(\hat{\mathbf{L}} + \hat{\mathbf{U}})\mathbf{e}^0$ is centered in zero and

$$|\mathbf{b} - (\hat{\mathbf{L}} + \hat{\mathbf{U}})\mathbf{e}^0| = |\mathbf{b}| + |(\hat{\mathbf{L}} + \hat{\mathbf{U}})\mathbf{e}^0| = |\mathbf{b}| + |\mathbf{L} + \mathbf{U}||\mathbf{e}^0|.$$

Moreover, $\mathbf{D}^{-1}$ is a positive diagonal matrix, hence

$$|\mathbf{D}_{ii}^{-1}| = |1/\mathbf{D}_{ii}| = 1/\langle \mathbf{D}_{ii} \rangle$$

which implies that $|\mathbf{D}^{-1}| = \langle \mathbf{D} \rangle^{-1}$.

Therefore

$$|\mathbf{e}'| = \langle \mathbf{D} \rangle^{-1}(|\mathbf{b}| + |\mathbf{L} + \mathbf{U}||\mathbf{e}^0|). \tag{4.5}$$

Equation (4.5) is in fact the Jacobi iteration to compute an approximate solution of the system $(\langle \mathbf{D} \rangle - |\mathbf{L} + \mathbf{U}|)x = |\mathbf{b}|$, or $\langle \tilde{\mathbf{A}} \rangle x = |\mathbf{b}|$. Therefore, the convergence rate of the relaxed interval Jacobi iterations is the same as the convergence rate of the real Jacobi iterations when applied to the system $\langle \mathbf{A} \rangle x = |\mathbf{b}|$.

Denote by $u = \langle \mathbf{A} \rangle^{-1}|\mathbf{b}|$ the exact solution of the system $\langle \mathbf{A} \rangle x = |\mathbf{b}|$, then

$$\begin{aligned}
\langle \mathbf{D} \rangle |\mathbf{e}'| &= |\mathbf{b}| + |\mathbf{L} + \mathbf{U}||\mathbf{e}^0|\\
&= \langle \mathbf{A} \rangle u + |\mathbf{L} + \mathbf{U}||\mathbf{e}^0|\\
&= (\langle \mathbf{D} \rangle - |\mathbf{L} + \mathbf{U}|)u + |\mathbf{L} + \mathbf{U}||\mathbf{e}^0|\\
\Rightarrow \langle \mathbf{D} \rangle (|\mathbf{e}'| - u) &= |\mathbf{L} + \mathbf{U}|(|\mathbf{e}^0| - u).
\end{aligned}$$

If $\langle \mathbf{A} \rangle$ is a diagonally dominant matrix, then the Jacobi iterations converge very quickly. Denote

$$\theta = \mathtt{max}_i \left\{ \frac{\sum_{k \neq i} |\mathbf{A}_{ik}|}{\langle \mathbf{A}_{ii} \rangle} \quad i = (1, \ldots, n) \right\} = \| \langle \mathbf{D} \rangle^{-1} |\mathbf{L} + \mathbf{U}| \|_\infty$$

then

$$\begin{aligned} \| |\mathbf{e}'| - u \|_\infty &= \| \langle \mathbf{D} \rangle^{-1} |\mathbf{L} + \mathbf{U}| (|\mathbf{e}^0| - u) \|_\infty \\ &\leq \| \langle \mathbf{D} \rangle^{-1} |\mathbf{L} + \mathbf{U}| \|_\infty \| |\mathbf{e}^0| - u \|_\infty \\ &\leq \theta \| |\mathbf{e}^0| - u \|_\infty . \end{aligned}$$

Therefore, the series $\| |\mathbf{e}^0| - u \|_\infty$ converges more quickly than a geometric series of ratio θ. Moreover, the Jacobi iterations converge even more quickly if the coefficient matrix is close to the identity, since in this case θ is close to 0.

4.2.2 Accuracy

It is evident that using the relaxed system to improve the solution will produce a larger enclosure of the solution than the original system. Nonetheless, in the context of our method, the interval system is a preconditioned system, hence it is nearly centered in a diagonal matrix if the original system is not too ill-conditioned.

This relaxation technique is not really a new idea. Although it is not explicitly stated so, this relaxation technique is already used in [HS81, Han92, Neu99]. Let us recall Theorem 3.2.2: the solution of the original system (4.1) is bounded by

$$\square \Sigma(\mathbf{A}, \mathbf{b}) \subseteq \mathbf{e}^o$$

where

$$u = \langle \mathbf{A} \rangle^{-1} |\mathbf{b}|, \quad d_i = (\langle \mathbf{A} \rangle^{-1})_{ii},$$

$$\alpha_i = \langle \mathbf{A} \rangle_{ii} - 1/d_i, \quad \beta_i = u_i/d_i - |\mathbf{b}_i|,$$

and

$$\mathbf{e}_i^o = \frac{\mathbf{b}_i + [-\beta_i, \beta_i]}{\mathbf{A}_{ii} + [-\alpha_i, \alpha_i]}.$$

As $\hat{\mathbf{L}}$ and $\hat{\mathbf{U}}$ are centered in zero, the midpoint of $\hat{\mathbf{A}}$ is a diagonal matrix. Thus, applying Theorem 3.2.2, the hull of the solution set of the interval system $\hat{\mathbf{A}}\mathbf{e} = \mathbf{b}$ is computed by

$$\square \Sigma(\hat{\mathbf{A}}, \mathbf{b}) = \hat{\mathbf{e}}$$

where

$$\hat{u} = \langle \hat{\mathbf{A}} \rangle^{-1} |\mathbf{b}|, \quad \hat{d}_i = (\langle \hat{\mathbf{A}} \rangle^{-1})_{ii}$$

$$\hat{\alpha}_i = \langle \hat{\mathbf{A}} \rangle_{ii} - 1/\hat{d}_i, \quad \hat{\beta}_i = \hat{u}_i/\hat{d}_i - |\mathbf{b}_i|.$$

and

$$\hat{\mathbf{e}}_i = \frac{\mathbf{b}_i + \left[-\hat{\beta}_i, \hat{\beta}_i\right]}{\hat{\mathbf{A}}_{ii} + [-\hat{\alpha}_i, \hat{\alpha}_i]}.$$

We have that $\hat{\mathbf{A}} = \hat{\mathbf{L}} + \mathbf{D} + \hat{\mathbf{U}}$ and thus $\hat{\mathbf{A}}_{ii} = \mathbf{D}_{ii} = \mathbf{A}_{ii}$. Moreover, following Lemma 4.2.1, $\langle\hat{\mathbf{L}}\rangle = \langle\mathbf{L}\rangle$ and $\langle\hat{\mathbf{U}}\rangle = \langle\mathbf{U}\rangle \Rightarrow \langle\hat{\mathbf{A}}\rangle = \langle\hat{\mathbf{L}}\rangle + \langle\mathbf{D}\rangle + \langle\hat{\mathbf{U}}\rangle = \langle\mathbf{L}\rangle + \langle\mathbf{D}\rangle + \langle\mathbf{U}\rangle = \langle\mathbf{A}\rangle$. It means that: $\hat{u} = u, \hat{d} = d, \hat{\alpha} = \alpha$, $\hat{\beta} = \beta$. Hence $\mathbf{e}^o = \hat{\mathbf{e}}$. In other words, Theorem 3.2.2 computes an enclosure of the solution of (4.1) by computing the hull of the solution set of the relaxed system (4.3).

Take Example 6 again. Applying Theorem 3.2.2 to compute an enclosure of the solution, we have

$$\begin{aligned}
\langle\mathbf{A}\rangle &= \begin{pmatrix} 4 & -1 \\ -1 & 4 \end{pmatrix} & \langle\mathbf{A}\rangle^{-1} &= \begin{pmatrix} 4/15 & 1/15 \\ 1/15 & 4/15 \end{pmatrix} \\
u = \langle\mathbf{A}\rangle^{-1}|b| &= \begin{pmatrix} 18/15 \\ 12/15 \end{pmatrix} & d &= \begin{pmatrix} 4/15 \\ 4/15 \end{pmatrix} \\
\alpha &= \begin{pmatrix} 1/4 \\ 1/4 \end{pmatrix} & \beta &= \begin{pmatrix} 1/2 \\ 1 \end{pmatrix} \\
\mathbf{e}_1^0 = \tfrac{4+[-1/2,1/2]}{4+[-1/4,1/4]} &= [14/17, 6/5] & \mathbf{e}_2^0 &= \tfrac{2+[-1,1]}{4+[-1/4,1/4]} = [4/17, 4/5]\,.
\end{aligned}$$

The interval vector $\mathbf{e}$ is in fact the hull of the solution set of the relaxed system (4.4), which is depicted by the outer dashed box in Figure 4.1. Meanwhile, the exact hull of the solution set of the original system is

$$\mathbf{e}^* = \begin{pmatrix} [14/15, 68/63] \\ [4/15, 40/63] \end{pmatrix}$$

which is depicted by the inner dotted dashed box. It is obvious that $\mathbf{e}^*$ is included in $\mathbf{e}$. Therefore, it is suggested by Neumaier [Neu99] that if the midpoint of the coefficient matrix is not a diagonal matrix, then some Gauss-Seidel iterations should be applied in order to obtain a tighter enclosure of the solution.

Nevertheless, the computed result of the interval improvement is not the hull of the solution of the system $\hat{\mathbf{A}}\mathbf{x} = \mathbf{b}$. Jacobi iterations converge to an interval vector, which is the solution of the following system of equations:

$$\mathbf{e}_i = \left(\mathbf{b}_i - \sum_{k\neq i} \hat{\mathbf{A}}_{ik}\mathbf{e}_k\right) / \tilde{\mathbf{A}}_{ii} \quad i = (1, \ldots, n).$$

The uniqueness of this solution is provided by [Neu90, Theorem 4.4.4, p145], we recall it below.

Theorem 4.2.3. *Let $\mathbf{A} \in \mathbb{IR}^{n\times n}$ be an H-matrix. Then for all $\mathbf{b} \in \mathbb{IR}^n$, the system of equations*

$$\mathbf{e}_i = \left(\mathbf{b}_i - \sum_{k\neq i} \mathbf{A}_{ik}\mathbf{e}_k\right) / \mathbf{A}_{ii} \quad i = (1, \ldots, n)$$

has a unique solution $\mathbf{z} \in \mathbb{IR}^n$.

Denote $\mathbf{A}^F\mathbf{b} = \mathbf{z}$. If $\mathbf{A}$ is thin then $\mathbf{A}^F\mathbf{b} = \mathbf{A}^{-1}\mathbf{b}$.

Applied to the relaxed system, the midpoint of $\hat{\mathbf{A}}$ is a diagonal matrix and the fixed point of the Jacobi iterations can be computed by the following theorem [Neu90, Theorem 4.4.10, p150].

Theorem 4.2.4. *Let $\mathbf{A} \in \mathbb{IR}^{n\times n}$ be an H-matrix and suppose that* $\mathtt{mid}(\mathbf{A})$ *is diagonal. Then*

$$|\mathbf{A}^F\mathbf{b}| = \langle\mathbf{A}\rangle^{-1}|\mathbf{b}|,$$

and $\mathbf{z} = \mathbf{A}^F\mathbf{b}$ can be expressed in terms of $u = \langle\mathbf{A}\rangle^{-1}|\mathbf{b}|$ as

$$\mathbf{z}_i = \frac{\mathbf{b}_i + (\langle\mathbf{A}\rangle_{ii}u_i - |b_i|)\,[-1,1]}{\mathbf{A}_{ii}} \quad i = (1,\ldots,n).$$

Consequently, we can compare the enclosure of the solution set of the system $\hat{\mathbf{A}}\mathbf{x} = \mathbf{b}$, and the enclosure computed by Jacobi iterations.

Corollary 4.2.5. *Let $\mathbf{A} \in \mathbb{IR}^{n\times n}$ be an H-matrix and suppose that* $\mathtt{mid}(\mathbf{A})$ *is diagonal. Then for all $\mathbf{b} \in \mathbb{IR}^n$*

(1) $|\mathbf{A}^F\mathbf{b}| = |\mathbf{A}^H\mathbf{b}|$,

(2) $\mathtt{rad}(\mathbf{A}^F\mathbf{b}) \leq 2 * \mathtt{rad}(\mathbf{A}^H\mathbf{b})$.

Proof. Following Theorem 3.2.2, since $\mathtt{mid}(\mathbf{A})$ is diagonal, we have

$$(\mathbf{A}^H\mathbf{b})_i = \frac{\mathbf{b}_i + [-\beta_i,\beta_i]}{\mathbf{A}_{ii} + [-\alpha_i,\alpha_i]}.$$

Since $\mathbf{A}$ is an H-matrix then $0 \notin \mathbf{A}_{ii}$. Suppose that $\mathbf{A}_{ii} > 0$, then

$$|(\mathbf{A}^H\mathbf{b})_i| = \frac{|\mathbf{b}_i| + \beta_i}{\langle\mathbf{A}\rangle_{ii} - \alpha_i} = \frac{|\mathbf{b}_i| + u_i/d_i - |\mathbf{b}_i|}{\langle\mathbf{A}\rangle_{ii} - (\langle\mathbf{A}\rangle_{ii} - 1/d_i)} = \frac{u_i/d_i}{1/d_i} = u_i = (\langle\mathbf{A}\rangle^{-1}|\mathbf{b}|)_i.$$

Following Theorem 4.2.4, $|\mathbf{A}^F\mathbf{b}| = \langle\mathbf{A}\rangle^{-1}|\mathbf{b}|$. Therefore the first equality holds. Because of the subdistributivity property, we have

$$(\mathbf{A}^F\mathbf{b})_i = \frac{\mathbf{b}_i + (\langle\mathbf{A}\rangle_{ii}u_i - |b_i|)\,[-1,1]}{\mathbf{A}_{ii}} \subseteq \frac{\mathbf{b}_i}{\mathbf{A}_{ii}} + (\langle\mathbf{A}\rangle_{ii}u_i - |b_i|)\frac{[-1,1]}{\mathbf{A}_{ii}} \equiv \mathbf{y}_i,$$

where

$$\begin{aligned}
|\mathbf{y}_i| &\leq \left|\frac{\mathbf{b}_i}{\mathbf{A}_{ii}}\right| + \left|(\langle\mathbf{A}\rangle_{ii}u_i - |b_i|)\frac{[-1,1]}{\mathbf{A}_{ii}}\right| = \frac{|\mathbf{b}_i|}{\langle\mathbf{A}\rangle_{ii}} + (\langle\mathbf{A}\rangle_{ii}u_i - |b_i|)\frac{1}{\langle\mathbf{A}\rangle_{ii}} = u_i,\\
\mathtt{mid}(\mathbf{y}_i) &= \mathtt{mid}(\frac{\mathbf{b}_i}{\mathbf{A}_{ii}}) + \mathtt{mid}((\langle\mathbf{A}\rangle_{ii}u_i - |b_i|)\frac{[-1,1]}{\mathbf{A}_{ii}}) = \mathtt{mid}(\frac{\mathbf{b}_i}{\mathbf{A}_{ii}}).
\end{aligned}$$

Therefore

$$\mathtt{rad}(\mathbf{y}_i) = |\mathbf{y}_i| - |\mathtt{mid}(\mathbf{y}_i)| = u_i - |\mathtt{mid}(\mathbf{b}_i/\mathbf{A}_{ii})|.$$

It is obvious that

$$\mathtt{mid}\left(\frac{\mathbf{b}_i}{\mathbf{A}_{ii}}\right) \in \frac{\mathbf{b}_i}{\mathbf{A}_{ii}} \subseteq \frac{\mathbf{b}_i + [-\beta_i,\beta_i]}{\mathbf{A}_{ii} + [-\alpha_i,\alpha_i]} = (\mathbf{A}^H\mathbf{b})_i.$$

From the definition of the minimal magnitude value, we get:

$$\left|\mathtt{mid}\left(\frac{\mathbf{b}_i}{\mathbf{A}_{ii}}\right)\right| \geq \langle(\mathbf{A}^H\mathbf{b})_i\rangle.$$

Hence

$$2 * \mathtt{rad}(\mathbf{A}^H\mathbf{b})_i \geq |\mathbf{A}^H\mathbf{b}|_i - \langle\mathbf{A}^H\mathbf{b}\rangle_i \geq |\mathbf{A}^F\mathbf{b}|_i - |\mathtt{mid}\,(\mathbf{b}_i/\mathbf{A}_{ii})\,| \\ \geq |\mathbf{A}^F\mathbf{b}|_i - |\mathtt{mid}(\mathbf{A}^F\mathbf{b})|_i \geq \mathtt{rad}(\mathbf{A}^F\mathbf{b})_i.$$

The case $\mathbf{A}_{ii} < 0$ is similar to the case $\mathbf{A}_{ii} > 0$. □

In other words, the above corollary means that when the Jacobi iterations are applied to the relaxed system, the convergent interval of the iterations is at worst twice larger than the exact hull of the solution set of the relaxed system.

4.2.3 Initial solution of the relaxed system

With the assumption of $\mathbf{A}$ being an H-matrix, an initial enclosure of the solution of the original system (4.1) can be computed using Proposition 3.4.3.

Let $w \in \mathbb{R}^n$ be a non-negative vector such that $\langle\mathbf{A}\rangle v \geq w > 0$ then:

$$\mathbf{A}^{-1}\mathbf{b} \subseteq \|\mathbf{b}\|_w \cdot [-v, v]\,.$$

Meanwhile, $\langle\hat{\mathbf{A}}\rangle = \langle\mathbf{A}\rangle \Rightarrow \langle\hat{\mathbf{A}}\rangle v \geq w > 0$. It means that the solution of the relaxed system (4.3) is bounded by the same interval vector as the original system:

$$\hat{\mathbf{A}}^{-1}\mathbf{b} \subseteq |\mathbf{b}|_w \cdot [-v, v]\,.$$

In Example 6, if v is set to $[1,\ 1]^T$ then

$$w = \langle\mathbf{A}\rangle v = \begin{pmatrix} 3 \\ 3 \end{pmatrix} > 0.$$

Hence, the initial solution of both the original system and the relaxed system is computed by

$$\|\mathbf{b}\|_w \cdot [-v, v] = \frac{4}{3}\begin{pmatrix} [-1,1] \\ [-1,1] \end{pmatrix} = \begin{pmatrix} [-4/3, 4/3] \\ [-4/3, 4/3] \end{pmatrix}.$$

This initial solution is much larger then the hull of the solution set computed by Theorem 3.2.2, which is $\begin{pmatrix} [14/17, 6/5] \\ [4/17, 4/5] \end{pmatrix}$.

On the other hand, from Theorem 4.2.4, if we can obtain an initial solution $\mathbf{e}^0 = [-u, u]$, with $u = \langle\mathbf{A}\rangle^{-1}|\mathbf{b}|$ then the interval improvement converges after only one Jacobi iteration:

$$\begin{aligned}
\mathbf{e}_i^1 &= \left(\mathbf{b}_i - \sum_{k\neq i} \mathbf{A}_{ik}\mathbf{e}_k^0\right)/\mathbf{A}_{ii} \\
&= \left(\mathbf{b}_i - \sum_{k\neq i} [-|\mathbf{A}_{ik}|u_k, |\mathbf{A}_{ik}|u_k]\right)/\mathbf{A}_{ii} \\
&= \left(\mathbf{b}_i - (\sum_{k\neq i} |\mathbf{A}_{ik}|u_k)\,[-1,1]\right)/\mathbf{A}_{ii}.
\end{aligned}$$

Since $u = \langle \mathbf{A} \rangle^{-1} |\mathbf{b}|$ then $\langle \mathbf{A} \rangle_{ii} u_i + \sum_{k \neq i} |\mathbf{A}_{ik}| u_k = |\mathbf{b}_i|$ and thus $\sum_{k \neq i} |\mathbf{A}_{ik}| u_k = |\mathbf{b}_i| - \langle \mathbf{A} \rangle_{ii} u_i$. It implies that

$$\mathbf{e}_i^1 = \left(\mathbf{b}_i - \left(|\mathbf{b}_i| - \langle \mathbf{A} \rangle_{ii} u_i\right)[-1, 1]\right) / \mathbf{A}_{ii} = \mathbf{z}_i,$$

where $\mathbf{z}_i$ is defined as in Theorem 4.2.4.

Therefore, instead of $[1, \ldots, 1]^T$, if we set the value of v to an approximation of the solution of the system $\langle \mathbf{A} \rangle x = |\mathbf{b}|$, then we obtain a better initial enclosure.

In this example, if we set v to $\langle \mathbf{A} \rangle^{-1} |\mathbf{b}| = \begin{pmatrix} 1.2 \\ 0.8 \end{pmatrix}$, then

$$w = \langle \mathbf{A} \rangle v = \begin{pmatrix} 4 \\ 2 \end{pmatrix} > 0.$$

Hence, the initial enclosure is

$$\|\mathbf{b}\|_w \cdot [-v, v] = \begin{pmatrix} [-1.2, 1.2] \\ [-0.8, 0.8] \end{pmatrix}.$$

This initial enclosure is better than that for $v = [1, \ldots, 1]^T$ and makes the iterations converge more quickly. Nonetheless, the computation of an approximation of the solution of the system $\langle \mathbf{A} \rangle x = |\mathbf{b}|$ costs $\mathcal{O}(n^3)$ floating-point operations and therefore introduces a significant overhead of execution time to the overall algorithm. Thus we do not use it in our algorithms.

4.3 Implementation and numerical experiments

In this section we implement a relaxed version, called `certifylss_relaxed`, of the function `certifylss`, in MatLab using the IntLab library to explore the performance as well as the accuracy of the relaxation technique.

4.3.1 Implementation

There are only some minor modifications in `certifylss_relaxed` with respect to its predecessor `certifylss`. First, the preconditioned matrix is inflated, hence it is not necessary to store the off-diagonal components of the preconditioned matrix as an interval but to store only their maximal magnitude value. Consequently, to store the product of that matrix with an interval vector, it is sufficient to store only the maximal magnitude of the result.

Algorithm 4.1 describes the interval improvement using Jacobi iterations and the relaxation technique.

Note that D and K are computed only once, and need not to be recomputed each time the interval improvement is applied.

4.3.2 Numerical results

The enclosure computed by the relaxed residual system will be of larger relative error than that of the original system. Nevertheless, in our case, this enclosure is used only to update

Algorithm 4.1 Interval improvement function `refine` using relaxed Jacobi iterations

Require: $\mathbf{e} \in \mathbb{IF}^n$ be an initial enclosure of the solution of the interval system $\mathbf{Ae} = \mathbf{z}$, where $\mathbf{A} \in \mathbb{IF}^{n \times n}$ be an H-matrix, and $\mathbf{z} \in \mathbb{IF}^n$
Ensure: $\mathbf{e}$ is improved using Jacobi iterations
  $D = \mathtt{diag}(\mathbf{A})$
  $K = \mathtt{mag}(\mathbf{A})$
  Set the diagonal components of K to 0
  while (termination criteria not satisfied) **do**
    `setround`(1)
    $tmp = K * \mathtt{mag}(\mathbf{e})$
    $\mathbf{e}1 = (\mathbf{z} + \mathtt{infsup}(-tmp, tmp))\ ./\ D$
    $\mathbf{e} = \mathtt{intersect}(\mathbf{e}, \mathbf{e}1)$
  end while

the approximate solution and to bound the error upon this solution. Hence it is not always true that the solution computed by `certifylss_relaxed` is of larger relative precision than that computed by `certifylss`.

Figure 4.2 depicts numerical results with matrices of dimensions 1000×1000 and condition number varying from 2^5 to 2^{50} for the three verified functions, namely `verifylss` of the IntLab library, `certifylss`, `certifylss_relaxed`, together with results computed by MatLab. Again, the matrices used in the tests are generated by the MatLab function `gallery` $('randsvd', dim, cond)$.

One can see that, although it relaxes the tightness of the error bound enclosure, the function `certifylss_relaxed` always provides results of the same accuracy as those computed by the function `certifylss`. This is also the reason why we do not use the algorithms presented in Chapter 2 for the interval matrix product. Indeed, in this case, it is not necessary to compute a very accurate enclosure of the interval matrix vector product.

Moreover, if `certifylss_relaxed` fails to verify the solution then `certifylss` also fails and vice versa. That is because the two functions share the same initial error bound enclosure. Failure to obtain an initial error bound means failure to verify the solution.

Nevertheless, using the relaxation technique helps to reduce a lot the overhead of execution time introduced by the interval improvement, and therefore it reduces a lot the overall execution time of the function. As it can be seen on Figure 4.2, `certifylss_relaxed` is even almost faster than `verifylss` except for too ill-conditioned systems.

When the condition number gets close to the machine relative error, there is a drop in execution time for both `certifylss_relaxed` and `verifylss`. It corresponds, for both, to a failure to compute an initial error bound. In such a case, there is no need to perform any further iterative refinement and the two functions terminate prematurely without any certification on the computed approximate solution.

Even though the use of the relaxation technique makes the error bound improvement phase become negligible in terms of execution time, when the condition number increases, the execution time increases significantly. Indeed, when the condition number increases, the number of iterations for refining the approximate solution increases. More precisely, the dominant computation, in terms of execution time, is the computation of the residual, as the computing precision is doubled, and thus the computation of the successive residuals is no more negligible compared to the overall execution time.

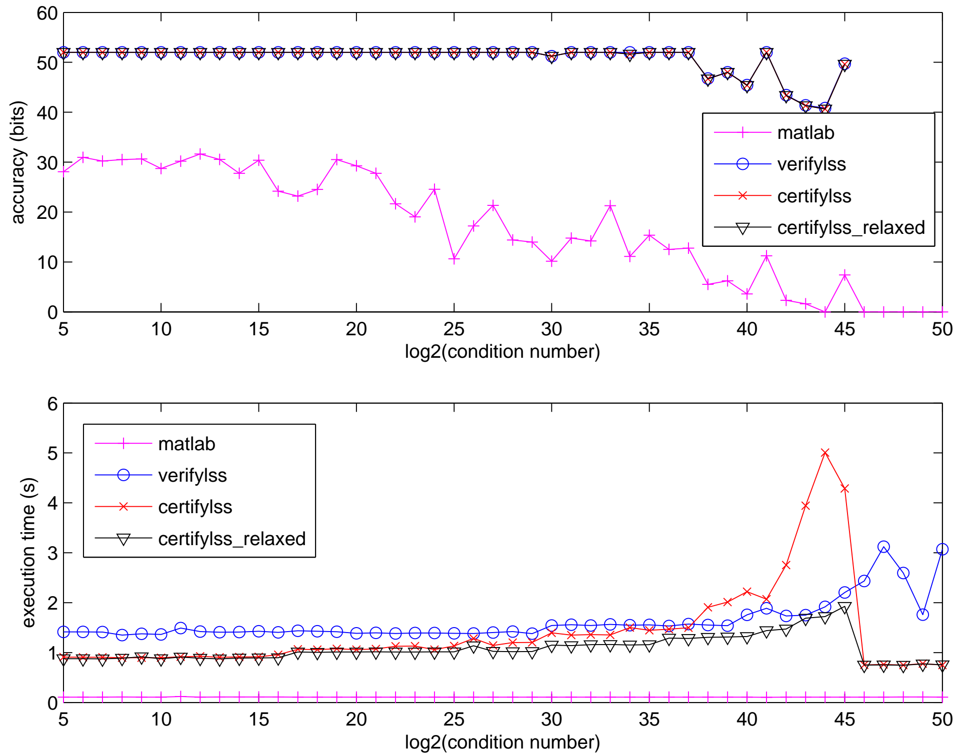


Figure 4.2: Relaxation technique: MatLab implementation results for 1000×1000 matrices.

4.4 Conclusion

By relaxing the exactness to exploit the symmetry of data as well as of operations, the performance has been significantly improved. At the same time, the relaxed method still allows to obtain good accuracy. Following Section 4.2.2, the hull of the solution set of the relaxed system is exactly the enclosure of the solution computed by the Ning-Kearfott method [NK97]. Moreover, Corollary 4.2.5 shows that the enclosure produced by applying interval Jacobi iterations on the relaxed system is at worst 2 times larger than the exact hull of the solution set of the relaxed system. It means that, in comparison with the Ning-Kearfott method, the relaxed method to compute an enclosure of the solution set of an interval linear system looses only 1 bit, whereas it costs only $\mathcal{O}(n^2)$ floating-point operations, instead of $\mathcal{O}(n^3)$ floating-point operations for the Ning-Kearfott method.

In the context of our problem, the relaxation technique is applied to the residual system. Therefore, although deteriorating the tightness of the error bound, in effect, it still allows to recover the lost bits in the approximate solution. Therefore, it does not deteriorate the accuracy of the main result. As shown in the Section 4.3, the relaxed method for the verification of the solution of a linear system is able to guarantee the same number of exact bits as the original method, which is explained in Chapter 3.

Additionally, using the relaxation technique, each interval improvement step in particular and the whole method in general can now be expressed almost solely by punctual operations except for the vectorial componentwise division. Therefore, the performance of the method relies entirely on the floating-point library employed. It also means that, without having to use a dedicated interval library, the method can be implemented using

only floating-point matrix library and appropriate rounding mode control. We can therefore use existing libraries which are well optimized for floating-point matrix operations, such as ATLAS [WP05], GotoBlas [TAC], LAPACK, or the Intel®Math Kernel Library, to implement our method.

The relaxation technique works very well with matrices whose midpoint is nearly diagonal. Hence it can be applied to any algorithm that needs to precondition the original system, for example the Krawczyk iterations, to improve the performance of the algorithm.

Chapter 5

Effects of the computing precision

The effect of the computing precisions used for each step of the function `certifylss` *on the accuracy, i.e. the width, of the final result is studied. The error analysis shows that, not only the computing precision for the residual computation, but also the computing precisions used for inverting the matrix, for pre-multiplying the matrix and for storing the approximate solution influence the accuracy of the final result. Nonetheless, for the sake of performance, we choose to increase the computing precision only for storing the approximate solution. Each approximate solution is stored as an unevaluated sum of two vectors in the working precision. By decoupling the computation of the residual and refining the two parts of the approximate solution successively, we can still obtain results which are guaranteed to almost the last bit with an acceptable overhead in execution time, even when the coefficient matrix is ill-conditioned.*

5.1 Introduction

Classically, in the implementation of iterative refinement using floating-point arithmetic, the residual is computed using twice the working precision in order to obtain a solution accurate to the working precision. The working precision is used for all the other computations, as well as for storing program variables.

In this chapter, we will study the effect of computing precision for each step of our algorithm. The use of a higher precision will increase the result quality but reduce the performance of the program. Conversely, a lower precision will increase the performance of the program at the cost of lower accuracy. Studying the effect of computing precision at each step allows us to find a trade-off between performance and accuracy.

5.2 Implementation using variable computing precisions

MPFR [FHL+07] and MPFI [RR05] are libraries that offer arbitrary precision for floating-point arithmetic and interval arithmetic, respectively. Using these two libraries to implement our algorithms, the computing precision can be tuned at each step in order to study the effect of these precisions on the result accuracy.

The matrices used in the following tests are generated by the MatLab function `gallery` $('randsvd', dim, cond)$, where dim is the matrix dimension (taken as 1000), and $cond$ is the

condition number. The coefficients of the generated matrices are IEEE double floating-point numbers. The condition number varies between 2^5 and 2^{50}.

In order to clarify each experimental set, let us recall the algorithm.

Algorithm 5.1 Solving and verifying a linear system

1: Compute the LU decomposition of A: $A = L \times U$
2: Compute an approximation $\tilde{x}$ with a forward and a backward substitution using L and U
3: Compute R, an approximate inverse of A, by solving $RL = \texttt{inv}(U)$ [Hig02, Chapter 14]
4: Pre-multiply A by R: $\boldsymbol{K} = [RA]$
5: Test if $\boldsymbol{K}$ is an H-matrix by computing a non-negative vector u such that $\langle \boldsymbol{K} \rangle u \geq v > 0$
6: **if** (fail to compute u) **then**
7:    **print** *Failed to verify the solution. The system may be either singular or too ill-conditioned.*
8:    **return**
9: **end if**
10: Compute the residual $\boldsymbol{r} = [b - A\tilde{x}]$ in double the working precision
11: Pre-multiply the residual by R: $\boldsymbol{z} = R\boldsymbol{r}$
12: Compute an initial error bound $\boldsymbol{e} = \|\boldsymbol{z}\|_v \cdot [-u, u]$
13: **while** (not converged) **do**
14:    Apply five Gauss-Seidel iterations on $\boldsymbol{K}, \boldsymbol{z}$, and $\boldsymbol{e}$
15:    Update $\tilde{x}$ and $\boldsymbol{e}$
16:    Recompute $\boldsymbol{r}$ and $\boldsymbol{z}$
17: **end while**
18: **return** $\tilde{x} + \boldsymbol{e}$

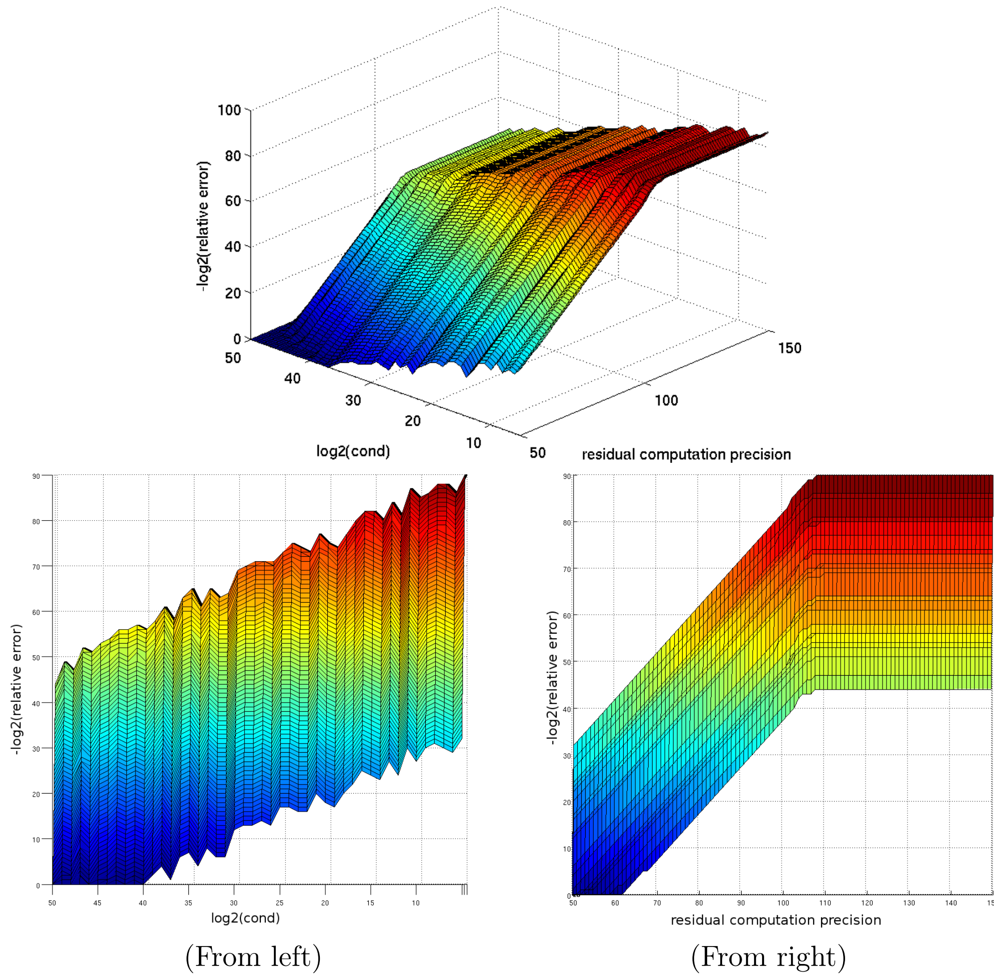
5.2.1 Residual computation

First, the precision for all computations is fixed to the working precision, except for the residual computation, to study its effect on the result accuracy. It means that only line 10 is performed in arbitrary precision, then the result is rounded to the working precision.

Figure 5.1 depicts results obtained with a fixed working precision of 53 bits. When the condition number decreases, the number of guaranteed correct bits (computed as in Eq.(3.6)) increases nearly linearly. When the residual computation precision increases, the number of guaranteed correct bits also increases linearly. However, when the residual computation precision gets a bit higher than twice the working precision, the number of correct bits stops increasing: this precision becomes too high and it does not have any effect when the other precisions remain low. Practically, this phenomenon justifies the use of twice the working precision for residual computations in iterative methods. From now on, the computing precision for the residual will be set to twice the working precision.

5.2.2 Floating-point working precision

Next the working precision varies. Only the output precision is fixed. The purpose of this set of experiments is to check, for a fixed output precision, whether it is mandatory to use



(From left) (From right)

Figure 5.1: Effects of the residual precision.

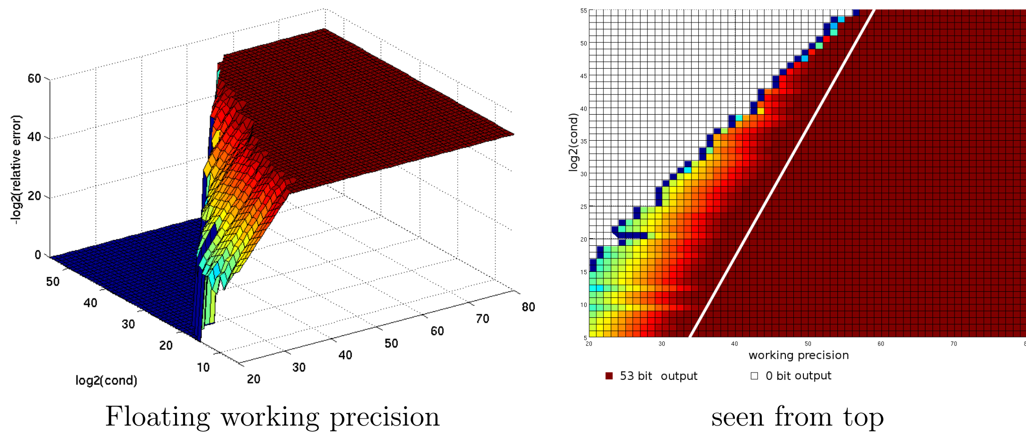
a precision equal to the output precision to get a result which is correct to the last bit. Experimental results are shown in figure 5.2.

For a fixed output precision, the needed working precision depends on the condition number, and this dependency is again linear. As a rule of thumb [Hig02, chap.1, p.10], the solution loses one digit of accuracy for every power of 10 in the condition number. On Figure 5.2, this rule can be interpreted as follows: in order to be able to verify the solution of the system, when the condition number increases by 1 bit, the computing precision should be increased also by 1 bit. Nevertheless, the phenomenon is different when the componentwise relative error is of concern. As revealed on Figure 5.2 by the white line, when the condition number increases by 2 bits, the computing precision should be increased by 1 bit in order to get a result which is correct to the last bit.

5.2.3 Needed working precision

In the final set of experiments, the demanded number of correct bits varies, and the minimal working precision that reaches the prescribed accuracy is determined.

As shown in Figure 5.3, the minimal working precision depends nearly linearly on the condition number and the number of correct bits demanded.



Floating working precision   seen from top

Figure 5.2: Effects of the working precision.

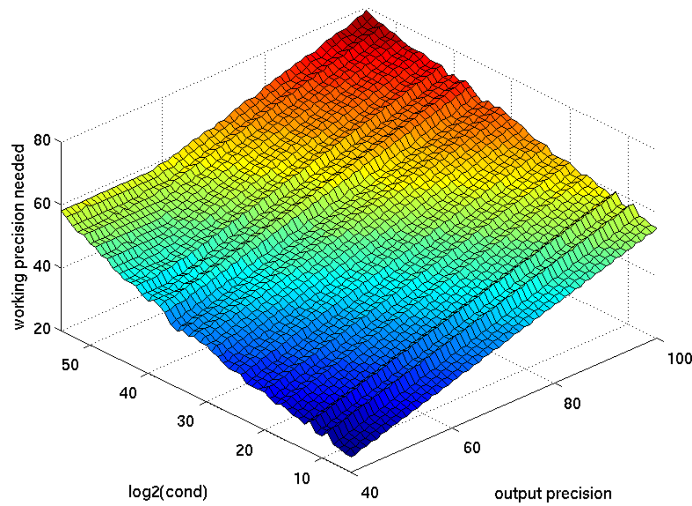


Figure 5.3: Minimal working precision needed.

5.3 Error analysis

As we can observe in the first set of experiments, and on Figure 5.1, the doubling of the working precision should be used for residual computation in order to obtain solutions of smallest relative error. Nevertheless, for ill-conditioned matrices, even if the residual is computed in twice the working precision or higher, we still cannot guarantee the last bits of the solution. This phenomenon has been covered in [DHK+06] for floating-point iterative refinement. For interval iterative refinement, failure to guarantee the last bits of the solution does not necessarily mean that an accurate floating-point solution cannot be obtained, but maybe that the iterations fail to compute a tight enclosure upon this approximate solution.

In this section, the error analysis of the interval iterative refinement is performed in order to study the effect of computing precisions on the result accuracy. For the sake of generality, different precisions will be used for each computation phase and for each variable storage format. Precisions are denoted by machine relative error ϵ. For a floating-point format of radix β where the mantissa is coded by p digits, the machine error is $\epsilon = \beta^{-p+1}$.

Precisely, the following precisions will be used in our analysis:

- ϵ_w is the working precision, if not mentioned explicitly the computations are performed in the working precision and the variables are stored in the working precision,
- ϵ_x is the precision used to store the approximate solution x,
- ϵ_i is the precision used to compute an inverse of A,
- ϵ_p is the precision used to pre-multiply the coefficient matrix,
- $\bar{\epsilon}_r$ and ϵ_r are the precisions used to compute and to store the residual respectively.

Let us assume that R is computed by Gauss elimination, using method B following [Hig02, chap.14, Section 14.3, p.268], which is used by LINPACK's `xGEDI`, LAPACK's `xGETRI`, and Matlab's `inv` function. Then the backward error is bounded by:

$$|RA - I| \leq c'_n \epsilon_i |R| \cdot |\tilde{L}| \cdot |\tilde{U}| \tag{5.1}$$

where c'_n is a constant depending on n, the dimension of the problem.

If $\tilde{L}, \tilde{U}$ are non-negative, from [Hig02, chap.9, Section 9.3, p.164] we have

$$|\tilde{L}| \cdot |\tilde{U}| \leq \frac{1}{1-\gamma_{n,i}} |A|, \text{ where } \quad \gamma_{n,i} = \frac{n\epsilon_i}{1-n\epsilon_i}.$$

For the general case, using GE without pivoting could return an arbitrarily large ratio $\||\tilde{L}||\tilde{U}|\|/\|A\|$. To analyze the stability of GE, we use the growth factor [Hig02, chap.9, Section 9.3, p.165]

$$\rho_n = \frac{\mathtt{max}_{i,j,k}(|a^k_{ij}|)}{\mathtt{max}_{i,j}(|a_{ij}|)}$$

which involves all the elements a^k_{ij} that occur during the elimination. Using this term we have the following result, established in [Hig02, Lemma 9.6, Chap.9, p.165]:

Lemma 5.3.1 ([Hig02]). *If $A = LU \in \mathbb{R}^{n\times n}$ is an LU factorization produced by GE without pivoting then*

$$\||\tilde{L}| \cdot |\tilde{U}|\|_\infty \leq (1 + 2(n^2 - n)\rho_n)\|A\|_\infty.$$

Let us now study the effect of pre-multiplying the coefficient matrix by its approximate inverse in precision ϵ_p. Following [Hig02, chap.3, Section 3.5, p.71], the forward error bound of a floating-point matrix multiplication is bounded by:

$$\tilde{K} = \mathtt{fl}(RA) \text{ verifies } \quad |\tilde{K} - RA| \leq \gamma_{n,p} * |R| \cdot |A|, \quad \gamma_{n,p} = \frac{n\epsilon_p}{1-n\epsilon_p}.$$

Hence, the width of the pre-multiplied matrix is bounded by:

$$\mathtt{diam}(\mathbf{K}) \leq 2\gamma_{n,p} * |R| \cdot |A|.$$

Its maximal magnitude value is bounded by:

$$\begin{aligned} |\mathbf{K} - I| &= |\mathbf{K} - RA + (RA - I)| \\ &\leq |\mathbf{K} - RA| + |RA - I| \\ &\leq \gamma_{n,p}|R| \cdot |A| + c'_n \epsilon_i |R| \cdot |\tilde{L}| \cdot |\tilde{U}|. \end{aligned} \tag{5.2}$$

Let recall that $\overline{\epsilon}_r$ and ϵ_r are the precisions used to compute and to store the residual respectively. Let $\tilde{r} \in \mathbb{F}^n$ be the result of $b - A\tilde{x}$ computed in floating-point arithmetic with some rounding mode, then following [Hig02, chap.12, Section 12.1, p.233]

$$\tilde{r} = b - A\tilde{x} + \Delta r, \quad |\Delta r| \leq [\epsilon_r + (1+\epsilon_r)\overline{\gamma}_{n+1,r}]|A| \cdot |x^* - \tilde{x}| + 2(1+\epsilon_r)\overline{\gamma}_{n+1,r}|A| \cdot |x^*|$$

where $\overline{\gamma}_{n+1,r} = (n+1)\overline{\epsilon}_r/(1-(n+1)\overline{\epsilon}_r)$. Hence

$$\mathbf{r} \subseteq A(x^* - \tilde{x}) + \texttt{midrad}(0, [\epsilon_r + (1+\epsilon_r)\overline{\gamma}_{n+1,r}]|A| \cdot |x^* - \tilde{x}| + 2(1+\epsilon_r)\overline{\gamma}_{n+1,r}|A| \cdot |x^*|).$$

Ignoring rounding errors introduced by pre-multiplying the residual, we have

$$\mathbf{z} = R\mathbf{r} \subseteq RA(x^* - \tilde{x}) + \texttt{midrad}(0, |R|([\epsilon_r + (1+\epsilon_r)\overline{\gamma}_{n+1,r}]|A| \cdot |x^* - \tilde{x}| + 2(1+\epsilon_r)\overline{\gamma}_{n+1,r}|A| \cdot |x^*|))$$

which gives

$$\begin{aligned} |\mathbf{z}| &\leq |RA(x^* - \tilde{x})| + |R|([\epsilon_r + (1+\epsilon_r)\overline{\gamma}_{n+1,r}]|A| \cdot |x^* - \tilde{x}| + 2(1+\epsilon_r)\overline{\gamma}_{n+1,r}|A| \cdot |x^*|) \\ &\leq |RA| \cdot |x^* - \tilde{x}| + [\epsilon_r + (1+\epsilon_r)\overline{\gamma}_{n+1,r}]|R| \cdot |A| \cdot |x^* - \tilde{x}| + 2(1+\epsilon_r)\overline{\gamma}_{n+1,r}|R| \cdot |A| \cdot |x^*|. \end{aligned}$$

Since $\tilde{x}$ is a floating-point approximate solution, the best bound upon the difference between $\tilde{x}$ and the exact solution x^* that can be expected is:

$$|\tilde{x} - x^*| \leq \frac{1}{2}\epsilon_x |x^*|.$$

Hence

$$|\mathbf{z}| \leq \frac{1}{2}\epsilon_x |RA| \cdot |x^*| + [\frac{1}{2}\epsilon_x[\epsilon_r + (1+\epsilon_r)\overline{\gamma}_{n+1,r}] + 2(1+\epsilon_r)\overline{\gamma}_{n+1,r}]|R| \cdot |A| \cdot |x^*|. \tag{5.3}$$

Let $\mathbf{D}$ denote the diagonal of $\mathbf{K}$, and $\mathbf{E}$ denote the off-diagonal components of $\mathbf{K}$. Suppose that with the interval improvement, i.e Jacobi or Gauss-Seidel iterations, the enclosure of the solution of the residual system converges to a fixed point $\mathbf{e}^f$ then $\mathbf{e}^f$ satisfies:

$$\begin{aligned} \mathbf{e}^f &= \mathbf{D}^{-1}(\mathbf{z} - \mathbf{E}\mathbf{e}^f) \\ \Rightarrow |\mathbf{e}^f| &= |\mathbf{D}^{-1}(\mathbf{z} - \mathbf{E}\mathbf{e}^f)| \\ &= |\mathbf{D}^{-1}| \cdot |\mathbf{z} - \mathbf{E}\mathbf{e}^f|. \end{aligned}$$

The necessary condition for Jacobi/Gauss-Seidel to be convergent is that $\mathbf{K}$ is an H-matrix. It implies that $\mathbf{D}$ does not contain zero in its interior. Therefore we can suppose here that $\mathbf{D}$ is positive, if it is not the case we simply multiply the rows of $\mathbf{K}$, and the corresponding element of $\mathbf{z}$, whose diagonal element is negative by -1. Hence

$$\begin{aligned} (\mathbf{D}^{-1})_{ii} &= 1/\mathbf{D}_{ii} \\ \Rightarrow |\mathbf{D}^{-1}|_{ii} &= 1/\langle \mathbf{D}_{ii} \rangle \\ \Rightarrow |\mathbf{e}^f|_i &= \frac{|(\mathbf{z} - \mathbf{E}\mathbf{e}^f|_i}{\langle \mathbf{D}_{ii} \rangle} \\ \Rightarrow |\mathbf{e}^f|_i \langle \mathbf{D}_{ii} \rangle &= |\mathbf{z} - \mathbf{E}\mathbf{e}^f|_i \\ \Rightarrow |\mathbf{e}^f|_i &= |\mathbf{z} - \mathbf{E}\mathbf{e}^f|_i + (I - \langle \mathbf{D} \rangle)_{ii} |\mathbf{e}^f|_i \\ \Rightarrow |\mathbf{e}^f| &\leq |\mathbf{z}| + |\mathbf{E}| \cdot |\mathbf{e}^f| + |\mathbf{D} - I| \cdot |\mathbf{e}^f| \\ &\leq |\mathbf{z}| + |\mathbf{K} - I| \cdot |\mathbf{e}^f|. \end{aligned}$$

Lemma 5.3.2. *If the midpoint of* $\mathbf{K}$ *is equal to the identity then:*

$$|\mathbf{e}^f| = |\mathbf{z}| + |\mathbf{K} - I| \cdot |\mathbf{e}^f|.$$

Proof. The fact that the midpoint of $\mathbf{K}$ is equal to the identity implies that $\mathbf{E}$ is centered in zero and the midpoint of $\mathbf{D}$ is equal to the identity. One can deduce that:

$$\begin{aligned}
\langle \mathbf{D} \rangle + |\mathbf{D}| &= \texttt{inf}(\mathbf{D}) + |\mathbf{D}| = 2I \\
\Rightarrow I - \langle \mathbf{D} \rangle &= |\mathbf{D}| - I = |\mathbf{D} - I|. \\
\mathbf{E}\mathbf{e}^f &= \left[-|\mathbf{E}| \cdot |\mathbf{e}^f|, |\mathbf{E}| \cdot |\mathbf{e}^f|\right] \\
\Rightarrow |\mathbf{z} + \mathbf{E}\mathbf{e}^f| &= |\texttt{inf}(\mathbf{z}) - |\mathbf{E}| \cdot |\mathbf{e}^f|, \texttt{sup}(\mathbf{z}) + |\mathbf{E}| \cdot |\mathbf{e}^f|| \\
&= \texttt{max}(\texttt{inf}(\mathbf{z}), \texttt{sup}(\mathbf{z})) + |\mathbf{E}| \cdot |\mathbf{e}^f| \\
&= |\mathbf{z}| + |\mathbf{E}| \cdot |\mathbf{e}^f|.
\end{aligned}$$

Moreover, $\mathbf{D} - I$ is a diagonal matrix and the diagonal components of $\mathbf{E}$ are zero, hence $\mathbf{E} + (\mathbf{D} - I) = \mathbf{E} + \mathbf{D} - I = \mathbf{K} - I$, and $|\mathbf{E}| + |\mathbf{D} - I| = |\mathbf{E} + \mathbf{D} - I| = |\mathbf{K} - I|$, which gives:

$$\begin{aligned}
|\mathbf{z} - \mathbf{E}\mathbf{e}^f| + (I - \langle \mathbf{D} \rangle)|\mathbf{e}^f| &= |\mathbf{z}| + |\mathbf{E}| \cdot |\mathbf{e}^f| + |\mathbf{D} - I| \cdot |\mathbf{e}^f| \\
&= |\mathbf{z}| + (|\mathbf{E}| + |\mathbf{D} - I|) \cdot |\mathbf{e}^f| \\
\Rightarrow |\mathbf{e}^f| &= |\mathbf{z}| + |\mathbf{K} - I| \cdot |\mathbf{e}^f|
\end{aligned}$$

which is the sought equality. ☐

Because the best bound that we can expect on $|\tilde{x} - x^*|$ is $\frac{1}{2}\epsilon_x|x^*|$, the best bound that we can expect on $|\mathbf{e}^f|$ is also $\frac{1}{2}\epsilon_x|x^*|$, which gives

$$|\mathbf{e}^f| \leq |\mathbf{z}| + \frac{1}{2}|\mathbf{K} - I|\epsilon_x|x^*|.$$

Using the bound for $|\mathbf{K} - I|$ given in (5.2), and the bound for $|\mathbf{z}|$ given in (5.3) we have

$$|\mathbf{e}^f| \leq \frac{1}{2}\epsilon_x|RA||x^*| + d_n|R||A||x^*| + \frac{1}{2}c'_n\epsilon_i\epsilon_x|R||\tilde{L}||\tilde{U}||x^*|$$

where

$$d_n = \frac{1}{2}\epsilon_x[\epsilon_r + (1 + \epsilon_r)\overline{\gamma}_{n+1,r}] + 2(1 + \epsilon_r)\overline{\gamma}_{n+1,r} + \frac{1}{2}\epsilon_x\gamma_{n,p}.$$

From (5.1), one can deduce that

$$|RA| \leq I + c'_n\epsilon_i|R| \cdot |\tilde{L}| \cdot |\tilde{U}|.$$

Hence

$$\begin{aligned}
|\mathbf{e}^f| &\leq \frac{1}{2}\epsilon_x\left(I + c'_n\epsilon_i|R| \cdot |\tilde{L}| \cdot |\tilde{U}|\right)|x^*| + d_n|R| \cdot |A| \cdot |x^*| + \frac{1}{2}c'_n\epsilon_i\epsilon_x|R| \cdot |\tilde{L}| \cdot |\tilde{U}| \cdot |x^*| \\
&\leq \frac{1}{2}\epsilon_x|x^*| + d_n|R| \cdot |A| \cdot |x^*| + c'_n\epsilon_i\epsilon_x|R| \cdot |\tilde{L}| \cdot |\tilde{U}| \cdot |x^*|.
\end{aligned}$$

The first term of the above bound corresponds to the rounding error of the computed result to the working precision, which is unavoidable. Meanwhile the other terms of the above bound are related to different calculations during the computing process, and indeed influence the accuracy of the final result: the second term is influenced by the computing precisions ϵ_r and $\overline{\epsilon}_r$ used for the residual computation and storage and by the computing precision ϵ_p used for the product $\mathbf{K} = [RA]$, and the third term is influenced by the computing precision ϵ_i used to compute the approximate inverse R of A and by the precision ϵ_x used to store the approximate solution $\tilde{x}$.

5.3.1 Normwise relative error

The normwise relative error of the certified solution is defined as $\frac{\|\tilde{x}-x^*\|_\infty}{\|x^*\|_\infty}$. Let us assume that $\tilde{x}$ is the approximate solution after convergence, then $\tilde{x} - x^* \in \mathbf{e}^f$. Therefore the normwise relative error is bounded by:

$$\begin{aligned}
\frac{\|\tilde{x} - x^*\|_\infty}{\|x^*\|_\infty} &\leq \frac{\|\mathbf{e}^f\|_\infty}{\|x^*\|_\infty} \\
&\leq \frac{1}{2}\epsilon_x + d_n \frac{\||R| \cdot |A| \cdot |x^*|\|_\infty}{\|x^*\|_\infty} + c_n' \epsilon_i \epsilon_x \frac{\||R| \cdot |\tilde{L}| \cdot |\tilde{U}| \cdot |x^*|\|_\infty}{\|x^*\|_\infty} \\
&\leq \frac{1}{2}\epsilon_x + d_n \frac{\|R\|_\infty \cdot \|A\|_\infty \cdot \|x^*\|_\infty}{\|x^*\|_\infty} + c_n' \epsilon_i \epsilon_x \frac{\|R\|_\infty \cdot \||\tilde{L}| \cdot |\tilde{U}|\|_\infty \cdot \|x^*\|_\infty}{\|x^*\|_\infty} \\
&\leq \frac{1}{2}\epsilon_x + d_n \|R\|_\infty \cdot \|A\|_\infty + c_n' \epsilon_i \epsilon_x \|R\|_\infty \cdot \||\tilde{L}| \cdot |\tilde{U}|\|_\infty.
\end{aligned}$$

Following Lemma 5.3.1 we get

$$\begin{aligned}
\frac{\|\tilde{x} - x^*\|_\infty}{\|x^*\|_\infty} &\leq \frac{1}{2}\epsilon_x + d_n \|R\|_\infty \cdot \|A\|_\infty + c_n' \epsilon_i \epsilon_x (1 + 2(n^2 - n)\rho_n) \|R\|_\infty \cdot \|A\|_\infty \\
&\leq \frac{1}{2}\epsilon_x + [d_n + c_n' \epsilon_i \epsilon_x (1 + 2(n^2 - n)\rho_n)] \widetilde{\kappa}(A) \\
&\leq \frac{1}{2}\epsilon_x + \frac{1}{2}\epsilon_x [\epsilon_r + (1 + \epsilon_r)\overline{\gamma}_{n+1,r} + \gamma_{n,p}] \widetilde{\kappa}(A) \\
&\quad + 2(1 + \epsilon_r)\overline{\gamma}_{n+1,r} \widetilde{\kappa}(A) + c_n' \epsilon_i \epsilon_x (1 + 2(n^2 - n)\rho_n) \widetilde{\kappa}(A). \qquad (5.4)
\end{aligned}$$

Note that, here, we use $\widetilde{\kappa}(A) = \|R\|_\infty \cdot \|A\|_\infty$ instead of $\kappa(A)$, because by definition $\kappa(A) = \||A^{-1}| \cdot |A|\|_\infty$ meanwhile we only have that R is an approximate inverse of A. Hopefully $R \approx A^{-1}$ hence $\widetilde{\kappa}(A) \approx \kappa(A)$.

(5.4) shows the dependence of the result quality on the condition number of A, as well as on the different precisions employed in the algorithm. Note that, although the precision ϵ_x is used to store x, the final result is still returned in the working precision ϵ_w. Hence, in order to achieve a solution which is guaranteed to the last bit, the following inequality must hold

$$\frac{\|\tilde{x} - x^*\|_\infty}{\|x^*\|_\infty} \leq \epsilon_w.$$

Suppose that $\widetilde{\kappa}(A)$ varies between 1 and $1/\epsilon_w$. It is obvious that x must be stored in a higher precision than the working precision, i.e. $\epsilon_x \leq \epsilon_w$. Therefore the term $\epsilon_x \frac{1}{2}[\epsilon_r + (1 + \epsilon_r)\overline{\gamma}_{n+1,r} + \gamma_{n,p}]\widetilde{\kappa}(A)$ is of order $\mathcal{O}(\epsilon_w)$ and does not affect significantly the result accuracy.

Nevertheless, the term $2(1 + \epsilon_r)\overline{\gamma}_{n+1,r}\widetilde{\kappa}(A)$ in (5.4) influences the result relative error. This is in fact the effect of the precision used for residual computation on the result quality. When A is ill-conditioned, say $\widetilde{\kappa}(A) = \frac{1}{k\epsilon_w}$, with $k > 1$ but not too big, in order that $2(1 + \epsilon_r)\overline{\gamma}_{n+1,r}\widetilde{\kappa}(A) = \mathcal{O}(\epsilon_w)$, we must have $\overline{\gamma}_{n+1,r} = \mathcal{O}(\epsilon_w^2)$. It means that twice the working precision must be used in order to get a result accurate to the last bit in the working precision.

Nevertheless, doubling the working precision for the residual computation does not suffice to obtain the result accurate to the last bit in the working precision. One can see on Figure 4.2 that when the condition number is high, there is a drop in result accuracy.

That is due to the term $c'_n \epsilon_x \epsilon_i (1+2(n^2-n)\rho_n)\widetilde{\kappa}(A)$ in (5.4). When A is ill-conditioned, say again $\widetilde{\kappa}(A) = \frac{1}{k\epsilon_w}$ with $k > 1$ but not too big, if we use the working precision to store the approximate solution and to compute the inverse of A, i.e $\epsilon_x = \epsilon_w$ and $\epsilon_i = \epsilon_w$ then

$$c'_n \epsilon_x \epsilon_i (1+2(n^2-n)\rho_n)\widetilde{\kappa}(A) = \frac{c'_n (1+2(n^2-n)\rho_n)}{k} \epsilon_w.$$

It seems that this term is of order $\mathcal{O}(\epsilon_w)$ and does not affect a lot the result quality. It is true when matrix dimensions are small, but no longer true for matrices of large dimensions. For example, if $n = 1000$ then $2(n^2 - n) \approx 2^{21}$. It makes this term significant and reduces the result accuracy.

In order to reduce the impact of the term $c'_n \epsilon_x \epsilon_i (1+2(n^2-n)\rho_n)\widetilde{\kappa}(A)$ on the result quality, one can either reduce ϵ_x or ϵ_i. That means either increasing the precision to store x or increasing the precision to compute the inverse of A. Nevertheless, inverting A costs $\mathcal{O}(n^3)$ `FLOPs`. Hence, increasing the precision to invert A will introduce a significant overhead to the algorithm execution time.

Therefore, we choose to increase the precision for storage format of x. As it will be shown in the next section, doubling the working precision for the approximate solution will yield results accurate to the last bit without deteriorating significantly the performance of the overall algorithm.

5.3.2 Componentwise relative error

Again, since $\tilde{x} - x^* \in \mathbf{e}^f$, then the componentwise relative error of $\tilde{x}$ is bounded by:

$$\begin{aligned}
\mathtt{max}_i \left(\frac{|\tilde{x} - x^*|_i}{|x^*|_i} \right) &\le \mathtt{max}_i \left(\frac{|\mathbf{e}^f|_i}{|x^*|_i} \right) \\
&\le \frac{1}{2}\epsilon_x + d_n \mathtt{max}_i \frac{(|R| \cdot |A| \cdot |x^*|)_i}{|x^*|_i} + c'_n \epsilon_i \epsilon_x \mathtt{max}_i \frac{(|R| \cdot |\tilde{L}| \cdot |\tilde{U}| \cdot |x^*|)_i}{|x^*|_i} \\
&\le \frac{1}{2}\epsilon_x + d_n \frac{\|R\|_\infty \cdot \|A\|_\infty \cdot \|x^*\|_\infty}{\mathtt{min}_i(|x^*|_i)} + c'_n \epsilon_i \epsilon_x \frac{\|R\|_\infty \cdot \||\tilde{L}| \cdot |\tilde{U}|\|_\infty \cdot \|x^*\|_\infty}{\mathtt{min}_i(|x^*|_i)} \\
&\le \frac{1}{2}\epsilon_x + \left(d_n \|R\|_\infty \cdot \|A\|_\infty + c'_n \epsilon_i \epsilon_x \|R\|_\infty \cdot \||\tilde{L}| \cdot |\tilde{U}|\|_\infty \right) \frac{\mathtt{max}_i(|x^*|_i)}{\mathtt{min}_i(|x^*|_i)}.
\end{aligned}$$

One can see that the componentwise relative error bound differs from the normwise relative error bound only by the factor $\frac{\mathtt{max}_i(|x^*|_i)}{\mathtt{min}_i(|x^*|_i)}$, which corresponds to the variation in absolute value of the components of the exact solution. If all the components of x^* are equal then the componentwise relative error is equal to the normwise relative error.

One problem occurs when using componentwise relative error, when there is a null component in the exact solution. In that case the factor $\frac{\mathtt{max}_i(|x^*|_i)}{\mathtt{min}_i(|x^*|_i)} \sim \infty$, and the relative error of the null component will also be infinite. Hence we suppose that there is no null component in the exact solution.

5.4 Doubling the working precision for approximate solution

To verify the effect of using twice the working precision for the approximate solution, in this section we implement, in MatLab using the Intlab library, a function called `certifylssx`, which follows the Algorithm 5.1 and uses twice the working precision for both computing the residual and storing the floating-point solution.

The use of higher precision for the approximate solution is likely to increase the algorithm execution time. But we will show later in this section that to achieve the same accuracy, there is hardly no difference between `certifylss` and `certifylssx` in terms of execution time. Nonetheless `certifylssx` allows to guarantee more bits when the matrix is ill-conditioned.

5.4.1 Implementation issues

Since there is no quadruple precision, to store the floating-point solution $\tilde{x}$ we will use a pair of two vectors of double precision floating-point numbers $\{\tilde{x}^1, \tilde{x}^2\}$, where $\tilde{x}^1$ represents the high part, i.e the first 53 bits, of $\tilde{x}$, and $\tilde{x}^2$ represents the low part of $\tilde{x}$: $\tilde{x} = \tilde{x}^1 + \tilde{x}^2$. In other words, the approximate solution is now stored as an unevaluated sum $\tilde{x}^1 + \tilde{x}^2$.

Therefore, the residual upon $\tilde{x}$ is computed, always in twice the working precision, by

$$\mathbf{r} = [b - A\tilde{x}]_2 = [b - A(\tilde{x}^1 + \tilde{x}^2)]_2.$$

An arithmetic using twice the working precision and this representation of numbers as sums of two floating-point numbers, is called a *double-double* arithmetic. Implementing a fully supported package of simulated double-double precision operations, namely sum and product, is very costly [Dek71, HLB01, LDB+02]. Instead, we reuse the function for computing the residual in twice the working precision with a minor modification. A new function called `residualxx` is implemented. Instead of returning an interval vector in the working precision which encloses the exact residual, `residualxx` returns a triple $\{r, er1, er2\}$, where $r, er1$, and $er2$ are vectors of double precision floating-point number: r is an approximation of the exact residual, and $er1, er2$ are respectively the lower bound and the upper bound of the difference between r and the exact residual. The pair $\{r, \texttt{infsup}(er1, er2)\}$ is in fact the intermediate result of the function `residual` just before returning the guaranteed residual as $r + \texttt{infsup}(er1, er2) = r + [er1, er2]$. Thus, $b - A * \tilde{x} \in (r + [er1, er2]) - A * \tilde{x}^2 \subseteq [r - A * \tilde{x}^2] + [er1, er2]$. Therefore, with this function, we can compute the residual upon $\tilde{x} = \{\tilde{x}^1, \tilde{x}^2\}$ in twice the working precision by:

$$\begin{aligned} [r1\ er1\ er2] &= \texttt{residualxx}(A, x1, b) \\ r &= \texttt{infsup}(er1, er2) + \texttt{residual}(A, x2, r1) \end{aligned}$$

where $x1$ in the code corresponds to $\tilde{x}^1$ and $x2$ to $\tilde{x}^2$.

Moreover, let $\mathbf{e}^2$ be the error bound upon $\tilde{x} = \{\tilde{x}^1, \tilde{x}^2\}$, then $\tilde{x}^2 + \mathbf{e}^2$ is also an error bound upon $\tilde{x}^1$. On the other hand, if $\mathbf{e}^1$ is the error bound upon $\tilde{x}^1$, then we can use $\texttt{mid}(\mathbf{e}^1)$ as $\tilde{x}^2$ and $\mathbf{e}^1 - \texttt{mid}(\mathbf{e}^1)$ as $\mathbf{e}^2$. This means that we can decouple the interval improvement into two phases to refine $x1$ and $x2$ separately. Suppose that $x1$, an initial floating-point approximate solution in the working precision, and $\mathbf{e}1$, an initial error bound upon this approximation, have been computed according to Section 3.2:

- Phase 1: compute the residual upon $x1$ as a pair of approximation vector and error bound vector $\{r1, \mathbf{er1}\}$. Use the system $\mathbf{Ke} = (r1 + \mathbf{er1})$ to improve $\mathbf{e}1$.

- Phase 2: set $x2 = \texttt{mid}(\mathbf{e}1)$ and $\mathbf{e}2 = \mathbf{e}1 - \texttt{mid}(\mathbf{e}1)$. compute the residual upon $x1+x2$ and use it to refine $\mathbf{e}2$.

A straightforward implementation would exhibit a slowdown with a factor 2. However, even if the approximate solution is computed and stored in twice the working precision, it is possible to implement Phase 2 in such a way that the execution time remains the same as before, where only the residual computation is performed in twice the working precision. Indeed, Phase 2 can be considered as updating $x1$ by $x2$ and refining the error bound upon this updated solution. In other words, by decoupling each interval iteration into two phases, where each one corresponds to one iteration in working precision, there is no overhead of execution time between the algorithm using extended-precision for the approximate solution and the one using working precision for the approximate solution.

Regarding the accuracy, the updating process in phase 2 can be considered as being performed in infinite precision for the approximate solution, which is stored in twice the working precision. Only the error bound is updated using the working precision. Hence, it increases the accuracy of the interval iterative refinement.

During the execution, there might be cases where $x1$ does not change after one iteration, such as, for example, when we obtain a good approximation of the solution but fail to compute a tight enclosure upon it. In those cases, there is no need to recompute the residual upon $x1$.

Additionally, a tight enclosure of the solution can be obtained right after phase 1 of the iteration. In that case, the program can terminate without having to proceed to phase 2.

The algorithm is detailed in Algorithm 5.2. Note that, for the sake of simplicity, Algorithm 5.2 does not use the relaxation technique. In reality, the relaxation technique is also applied to this algorithm in order to improve the performance.

5.4.2 Numerical results

We performed numerical experiments to compare the two functions `certifylssx` and `certifylss` both in terms of execution time and result accuracy. Matrices are again generated by the function `gallery` of MatLab, and the condition number varies between 2^5 and 2^{50}. Numerical results for matrices of dimensions 1000×1000 are depicted on Figure 5.4.

One can see that results computed by `certifylssx` are guaranteed to almost the last bit, except in the case of certification failure. Meanwhile, when the matrix is too ill-conditioned, both functions `certifylss` and `verifylss` fail to guarantee the last bit.

Nonetheless, when the matrix is not too ill-conditioned, there is no overhead of execution time in `certifylssx` with respect to `certifylss`. When the matrix is too ill-conditioned, `certifylssx` is a bit slower than `certifylss` at the benefit of gaining guaranteed bits.

5.4.3 Other extended precisions

From the bound (5.4), to improve the result accuracy one can reduce either ϵ_x or ϵ_i to compensate for the constant $c'_n(1+2(n^2-n))\rho_n$. As shown in the previous section, doubling the working precision for the solution allows to obtain very accurate results. In fact, it is

Algorithm 5.2 Doubling the precision for the approximate solution

Require: $A \in \mathbb{F}^{n\times n}, b \in \mathbb{F}^n$

Ensure: $x \in \mathbb{F}^n$ is an approximate solution of $Ax = b$, $\mathbf{e} \in \mathbb{IF}^n$ is an error bound on x

```
R = inv(A)
x = R * b
K = R * intval(A)
Acom = compmat(K)
Find u ≥ 0 such that v = Acom * u > 0 according to Algorithm 3.3
[r1 er1 er2] = residualxx(A, x1, b)
r = r1 + infsup(er1, er2)
z = R * r
e = max(mag(z)./v) * infsup(−u, u)
while (termination criteria not satisfied) do
  if (Not the first iteration && x1 changes) then
    [r1 er1 er2] = residualxx(A, x1, b)
    r = r1 + infsup(er1, er2)
    z = R * r
  end if
  e = refine(K, e, z)
  if (termination criteria satisfied) then
    break
  end if
  x2 = mid(e)
  e = e − x2
  r2 = residual(A, x2, r1) + infsup(er1, er2)
  e = refine(K, e, R * r2)
  e = x2 + e
  tmp = x + mid(e)
  e = e + (x − intval(tmp))
  x = tmp
  return x and e
end while
```

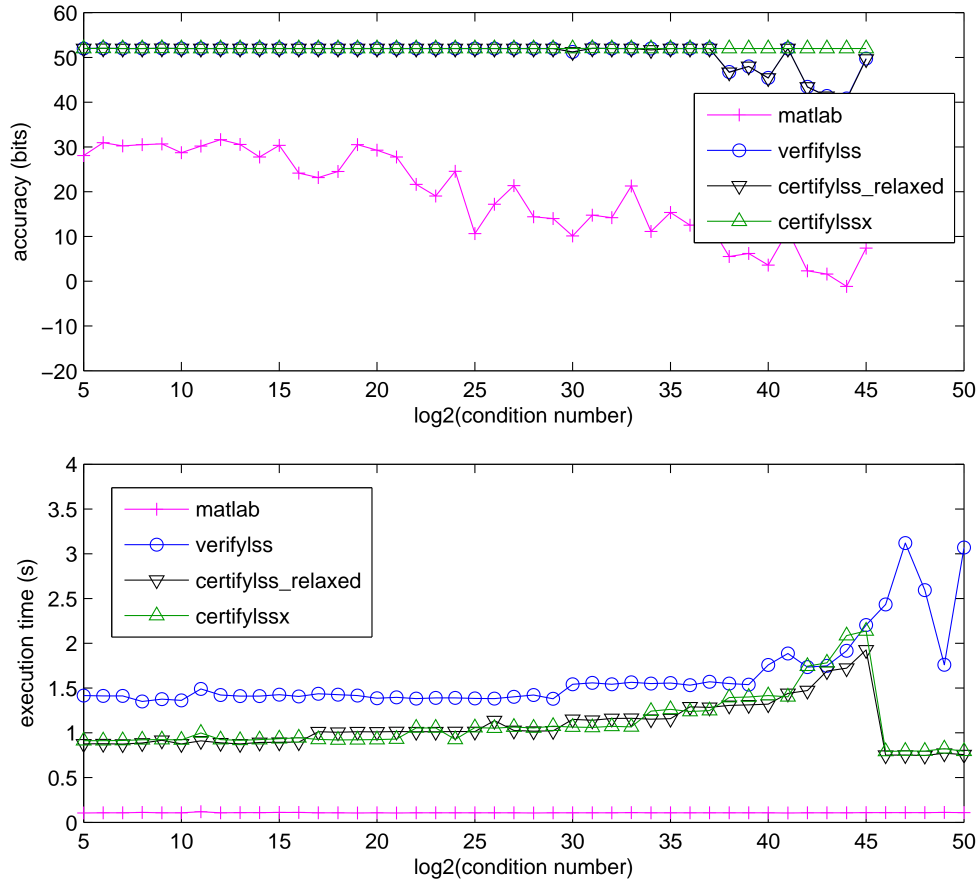


Figure 5.4: Effect of doubling the working precision for the approximate solution.

not really necessary to use twice the working precision. A lower precision is indeed sufficient to achieve results accurate to the last bit in the working precision. For example, $\tilde{x}$ can be coded by a couple $\{\tilde{x}^1, \tilde{x}^2\}$ where $\tilde{x}^1$ is one vector of IEEE double floating-point numbers and $\tilde{x}^2$ is one vector of IEEE single floating-point numbers. Using a lower precision for the solution can help to reduce the computation time of each interval improvement step. Nevertheless, using a lower precision will increase the error upon the residual computation. Hence, it will decrease the rate of convergence, or in other words, increase the number of iterations and thus the number of times residual computation is performed.

Figure 5.5 shows a comparison between the above function `certifylssx` and another version of it called `certifylssxs` which uses IEEE single precision for $\tilde{x}^2$. One can see that `certifylssxs` still succeeds to obtain very accurate results. Moreover, there is not much difference between the two functions in terms of execution time.

5.5 Conclusion

In contrast to the previous chapter, which consists in relaxing the exactness to improve the speed, in this chapter, we relax the speed requirement in order to improve the accuracy by studying the effect of the computing precision on both the accuracy and the speed of the algorithm. Classically, twice the working precision is used for the residual computation

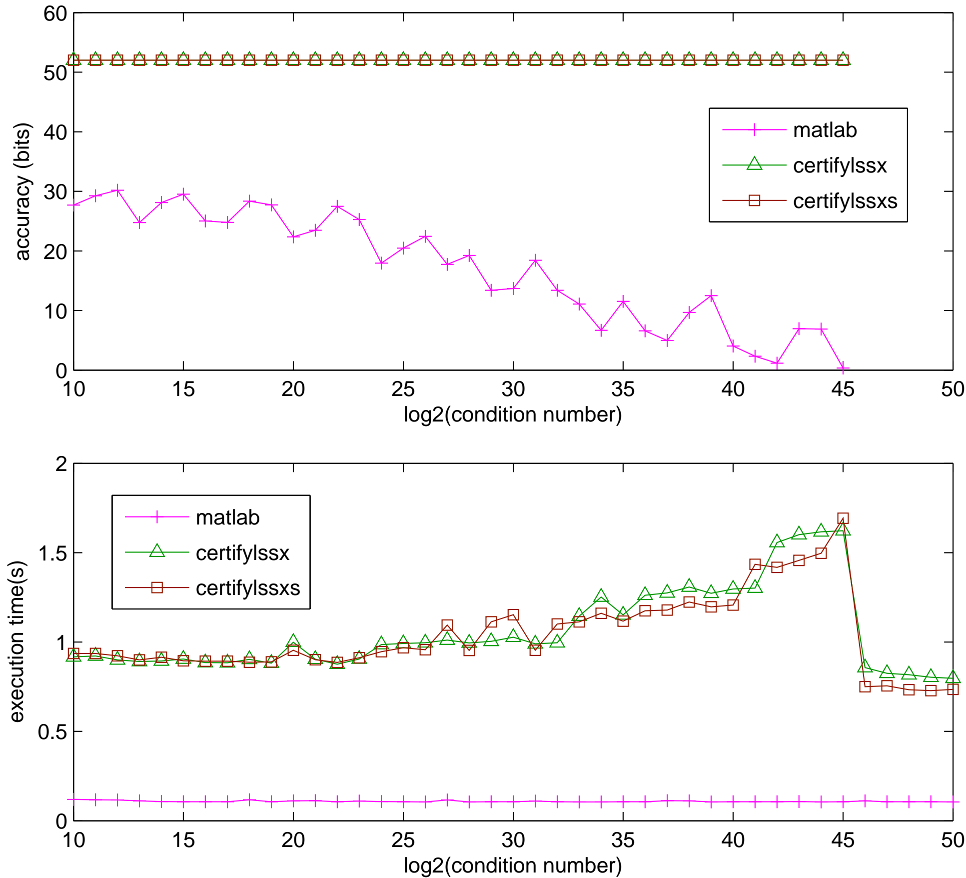


Figure 5.5: Effect of other extended precisions.

in order to obtain results which are accurate in the working precision. Nonetheless, in our case where rigorous error bounds are needed or componentwise relative errors are in question, doubling the working precision only for the residual computation does not suffice to guarantee the last bits of the computed results when the matrix is too ill-conditioned.

By performing the error analysis as well as performing some experiments using arbitrary precision libraries, namely the MPFR and MPFI libraries, to study the effect of the precision for each step on the final accuracy, we observed that in order to verify the last bits of the approximate solution, it is needed to increase either the precision for inverting the coefficient matrix or the precision for storing the intermediate approximate solution. Nevertheless, an increase of precision for inverting the coefficient matrix will lead to a drastic increase of execution time and memory usage. Therefore we chose to increase the precision for storing the approximate solution. By reusing the procedure for computing the residual in twice the working precision, the use of twice the working precision for storing the approximate solution does not harm the algorithm's speed when the condition number is small to intermediate. Furthermore, when the matrix is too ill-conditioned, the use of twice the working precision for the approximate solution helps to obtain very accurate results, up to the last bit, at a cost of acceptable overhead of execution, as revealed in the experimental section.

Conclusions and Perspectives

Conclusion

In this thesis, we concentrated on efficient implementations for verified linear algebra using interval arithmetic. Following the fundamental property of inclusion, interval arithmetic allows to compute an enclosure of the exact result, or a certificate of accuracy. Nevertheless, there are obstacles to the application of interval arithmetic. Firstly, interval arithmetic suffers from low performance in comparison with floating-point arithmetic. Secondly, algorithms using interval arithmetic usually provide results of large overestimation if care is not taken when implementing them. Our main contribution is to find and implement algorithms which run fast and at the same time provide sharp enclosures of the exact results. Two fundamental problems are addressed in this thesis, namely the multiplication of interval matrices and the verification of the solution to a linear system. For each problem, our contributions can be summarized as follows.

Multiplication of interval matrices

Using the natural algorithm to implement the multiplication of interval matrices suffers from very low performance due to the use of too many changes of rounding modes as well as maximal and minimal functions. S. M. Rump proposed an efficient algorithm which is based on midpoint-radius representation of intervals and uses merely floating-point matrix products. This is the fastest implementation till now and costs only 4 floating-point matrix products. Nevertheless, Rump's algorithm always provides overestimated results, and the factor of overestimation in the worst case is 1.5.

We present in this thesis two new algorithms, which are based respectively on endpoints and midpoint-radius representations of intervals and offer new trade-offs between speed and accuracy. In order to reduce the overestimation factor, we introduce a new scheme for decomposing intervals. Each component of one input matrix is decomposed into 2 parts, one which is centered in zero and the other which is the largest interval as possible which does not contain zero in its interior. With this decomposition, both new algorithms can be implemented using merely floating-point matrix products. They cost 9 and 7 floating-point matrix products respectively, and thus they are a bit slower than Rump's algorithm. In return, the factor of overestimation in the worst case of both new algorithms is reduced to less than 1.18. Furthermore, if all the components of one multiplier matrix does not contain zero, then the computed result is exact in the absence of rounding error. In practice, even in the presence of rounding errors, our implementation of the second algorithm yields accurate results, conforming to this theoretical result.

Verify the solution of a linear system of equations

In order to compute a sharp enclosure of the solution of a linear system, we adapt floating-point iterative refinement to interval arithmetic. First, an initial approximate solution is computed using floating-point arithmetic. Then we switch to interval arithmetic to compute the residual. Still in interval arithmetic, the residual system is pre-multiplied by an approximate inverse of the matrix and is used to refine the error bound using either interval Jacobi or interval Gauss-Seidel iterations. The combination of floating-point refinement and interval improvement allows us to obtain accurate results together with a tight enclosure of error bound. However, as the condition number increases, the execution time increases drastically. Moreover, as the matrix gets too ill-conditioned, this method fails to guarantee the last bits of computed result.

To tackle the first problem, we introduce the relaxation technique which consists in relaxing the exactness to gain in performance. The pre-multiplied matrix is indeed inflated in such a way that its midpoint is a diagonal matrix. Exploiting the symmetry of interval data helps to reduce significantly the execution time of the interval improvement phase. Nonetheless, the result accuracy is not affected by this relaxation. Although the error bound becomes larger due to inflated data, it is still able to recover the lost bits in the approximate solution.

The second problem is solved by applying extra-precise refinement, where extended precision is used not only for residual computation but also for storing the approximate solution, since the precision for the approximate solution influences the result accuracy when the matrix is too ill-conditioned. Numerical experiments have shown that by using extended precision for the approximate solution, our method provides very accurate results which are guaranteed to almost the last bit at the cost of an acceptable overhead of execution time.

Perspectives

During my Ph. D. preparation, MatLab is used as experimental environment. It allows to obtain quickly a “proof of concept” for the performance as well as the accuracy of the proposed algorithms. Nevertheless, because of the overhead in execution time introduced by the interpretation, functions implemented in MatLab suffer from low performance when employing loops such as the `for` and `while` loops, or when accessing separately rows and columns of matrices.

Nonetheless, an implementation in other programming languages, such as C or FORTRAN, does not suffer from these problems. By making direct calls to BLAS level 3 routines and accessing directly columns of the matrices, one can obtain a better implementation of the Gauss-Seidel iterations, and especially the residual computation. Although the residual computation costs $\mathcal{O}(n^2)$ floating-point operations, an implementation in MatLab is much more expensive in terms of execution time as explained in Section 3.5.1. Therefore, a better implementation of the residual computation will reduce significantly the execution time of the method.

Our method for verifying the solution of a linear system allows to compute a very accurate solution together with a tight enclosure of the error bound when the matrix is not too ill-conditioned. Nevertheless, when the matrix is too ill-conditioned, i.e. when the condition number of the matrix is close to $1/\epsilon$ with ϵ being the machine relative error,

then both our functions and the function `verifylss` of the IntLab library fail to verify the solution. It is because that a good approximation of the inverse of the coefficient matrix cannot be obtained.

To extend the range for condition number of verifiable systems, we can use a technique proposed by Rump [Rum09b] in order to inverse extremely ill-conditioned matrices. This technique is based on multiplicative correction terms and K-fold precision. For example, using 2-fold precision, i.e. twice the working precision, one can obtain a good approximation for matrices whose condition number falls between $1/\epsilon$ and $1/\epsilon^2$. This technique has also been used in [OOR09] to refine the solution of ill-conditioned linear equations, as well as in [Ogi10] to factorize ill-conditioned matrices.

A research direction that I am also interested in is the implementation of interval arithmetic on new hardware architectures. Till now, there is no dedicated processor for interval arithmetic. There was in fact a hardware chip called XPA 3233, which has been developed by Ulrich W. Kulisch in 1993 [Kul08], which supports long interval arithmetic through the exact dot product. However, this chip has never been put to production. The implementation of interval arithmetic on traditional processors suffers from some limits such as changing rounding mode, testing and branching, etc. Meanwhile new hardware architectures, for example the Cell processor [GLRM08], the Graphics Processing Unit (GPU) [CFD08], or the Field-Programmable Gate Array (FPGA), offer new computational capacities as well as new optimization options. On these new hardware architectures, numerous computations can be run simultaneously, for example the SIMD instructions on the Cell processor and the GPU. However, they have also some restrictions. For example, the Cell processor supports only the rounding mode toward zero for the single precision, meanwhile the GPU supports only the rounding modes toward nearest and toward zero. Therefore, care must be taken in order to obtain an efficient implementation of interval arithmetic on these architectures.

Currently, the midpoint-radius representation is less used than the endpoints representation except for matrix operations because of its disadvantage of lower accuracy. However, when the intervals are of small width then the midpoint-radius representation is almost of the same accuracy as the endpoints representation. As observed in Section 2.3.3, in some specific cases, the midpoint-radius representation is even of better accuracy. In implementing on new hardware architecture, the midpoint-radius exhibits another advantage in comparison with the endpoints representation, that is the midpoint and the radius can be stored in different precisions. For intervals of small width, a smaller precision can be used to represent the radius. That allows to reduce not only the memory usage but also the cost of data transfer.

Bibliography

[BM05] Sylvie Boldo and Jean-Michel Muller, *Some Functions Computable with a Fused-MAC*, Proceedings of the 17th Symposium on Computer Arithmetic (Cape Cod, USA) (Paolo Montuschi and Eric Schwarz, eds.), 2005, pp. 52–58.

[BT89] D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and distributed computation*, Prentice Hall, 1989.

[CFD08] Sylvain Collange, Jorge Flóres, and David Defour, *A GPU interval library based on Boost interval*, Real Numbers and Computers, July 2008, pp. 61–72.

[CK04] Martine Ceberio and Vladik Kreinovich, *Fast Multiplication of Interval Matrices (Interval Version of Strassen's Algorithm)*, Reliable Computing **10** (2004), no. 3, 241–243.

[Dek71] T.J. Dekker, *A floating-point technique for extending the available precision*, Numer. Math. **18** (1971), 224–242.

[DHK+06] James Demmel, Yozo Hida, William Kahan, Xiaoye S. Li, Sonil Mukherjee, and E. Jason Riedy, *Error bounds from extra-precise iterative refinement*, ACM Trans. Mathematical Software **32** (2006), no. 2, 325–351.

[FHL+07] Laurent Fousse, Guillaume Hanrot, Vicent Lefèvre, Patrick Pélissier, and Paul Zimmermann, *MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding*, ACM Trans. Mathematical Software **33** (2007), no. 2.

[GLRM08] Stef Graillat, Jean-Luc Lamotte, Siegfried M. Rump, and Svetoslav Markov, *Interval arithmetic on the Cell processor*, 13th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numerical Computations SCAN'08, 2008, pp. 54–54.

[GNL09] Stef Graillat, Hong Diep Nguyen, and Jean-Luc Lamotte, *Error-Free Transformation in rounding mode toward zero*, Lecture Notes in Computer Science (LNCS) **5492** (2009), 217–229, Numerical Validation in Current Hardware Architecture.

[Han92] E. R. Hansen, *Bounding the Solution of Interval Linear Equations*, SIAM Journal on Numerical Analysis **29** (1992), no. 5, 1493–1503.

[Hig90] N. J. Higham, *Is fast matrix multiplication of practical use ?*, SIAM News (1990), 12–14.

[Hig97] Nicholas J. Higham, *Iterative refinement for linear systems and LAPACK*, IMA Journal of Numerical Analysis **17** (1997), no. 4, 495–509.

[Hig02] ______, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM Press, 2002.

[HKKK08] Chenyi Hu, R. Baker Kearfott, Andre de Korvin, and Vladik Kreinovich, *Knowledge processing with interval and soft computing*, 1 ed., Springer Publishing Company, Incorporated, 2008.

[HLB01] Yozo Hida, Xiaoye S. Li, and David H Baily, *Algorithms for quad-double precision floating point arithmetic*, 15th IEEE Symposium on Computer Arithmetic, 2001, pp. 155–162.

[HS81] Eldon Hansen and Saumyendra Sengupta, *Bounding solutions of systems of equations using interval analysis*, BIT Numerical Mathematics **21** (1981), 203–211.

[IEE85] IEEE Task P754, *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*, IEEE, New York, NY, USA, August 1985.

[IEE08] ______, *IEEE 754-2008, Standard for Floating-Point Arithmetic*, IEEE, New York, NY, USA, August 2008.

[Int07] Intel Corporation, *Intel®64 and IA-32 Architectures Software Developer's Manual*, Specification, 2007, `http://www.intel.com/products/processor/manuals/index.htm`.

[JKDW01] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter, *Applied Interval Analysis*, Springer, 2001.

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, *Introduction to the cell multiprocessor*, IBM J. Res. Dev. **49** (2005), 589–604.

[Knu06] 0. Knuppel, *PROFIL/BIAS–A fast interval library*, Computing **53** (2006), no. 3–4, 277–287.

[Kul08] Ulrich W. Kulisch, *Computer Arithmetic and Validity: Theory, Implementation, and Applications*, de Gruyter Studies in Mathematics, Berlin, New York (Walter de Gruyter), 2008.

[Lan04] Philippe Langlois, *More accuracy at fixed precision*, ournal of Computational and Applied Mathematics **162** (2004), no. 1, 57–77.

[LD03] Xiaoye S. Li and James W. Demmel, *SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems*, ACM Trans. Mathematical Software **29** (2003), no. 2, 110–140.

[LDB+02] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo, *Design, implementation and testing of extended and mixed precision blas*, ACM Trans. Mathematical Software **28** (2002), 152–205.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Trans. Mathematical Software **5** (1979), 308–323.

[LL08a] Philippe Langlois and Nicolas Louvet, *Accurate solution of triangular linear system*, 13th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics, El Paso (TX), USA, September 2008.

[LL08b] ______, *Compensated Horner algorithm in K times the working precision*, RNC-8, Real Numbers and Computer Conference, Santiago de Compostela, Spain (J.D. Brugera and M. Daumas, eds.), July 2008.

[LLL+06] Julie Langou, Julien Langou, Piotr Luszczek, Jakub Kurzak, Alfredo Buttari, and Jack Dongarra, *Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy (Revisiting Iterative Refinement for Linear Systems) - Article 113 (17 pages)*, Proc. of the 2006 ACM/IEEE conference on Supercomputing, 2006.

[MB79] Ramon E. Moore and Fritz Bierbaum, *Methods and Applications of Interval Analysis (SIAM Studies in Applied and Numerical Mathematics) (Siam Studies in Applied Mathematics, 2.)*, Soc for Industrial & Applied Math, 1979.

[MBdD+10] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres, *Handbook of Floating-Point Arithmetic*, Birkhäuser Boston, 2010.

[MKC09] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud, *Introduction to Interval Analysis*, SIAM, Philadelphia, PA, USA, 2009.

[Mol67] Cleve B. Moler, *Iterative Refinement in Floating Point*, J. ACM **14** (1967), no. 2, 316–321.

[Moo63] Ramon Edgar Moore, *Interval arithmetic and automatic error analysis in digital computing*, Ph.D. thesis, Stanford University, Stanford, CA, USA, 1963.

[Neu90] Arnold Neumaier, *Interval Methods for Systems of Equations*, Cambridge University Press, 1990.

[Neu99] ______, *A Simple Derivation of the Hansen-Bliek-Rohn-Ning-Kearfott Enclosure for Linear Interval Equations*, Reliable Computing **5** (1999), no. 2, 131–136.

[NGL09] Hong Diep Nguyen, Stef Graillat, and Jean-Luc Lamotte, *Extended precision with a rounding mode toward zero environement. Application on the Cell processor*, Int. J. Reliability and Safety **3** (2009), no. 1/2/3, 153–173.

[Ngu] Hong Diep Nguyen, *Efficient implementation of interval matrix multiplication*, Proceedings of PARA 2010: State of the Art in Scientific and Parallel Computing.

[NK97] S. Ning and R. B. Kearfott, *A comparison of some methods for solving linear interval equations*, SIAM J. Numer. Anal. **34** (1997), no. 4, 1289–1305.

[NR] Hong Diep Nguyen and Nathalie Revol, *Resolve and Certify a Linear System*, to appear on Reliable Computing (Special volume for SCAN 2008).

[NR10] ______, *High performance linear algebra using interval arithmetic*, PASCO '10: Proceedings of the 4th International Workshop on Parallel and Symbolic Computation (Marc Moreno Maza and Jean-Louis Roch, eds.), ACM, 2010, pp. 171–172.

[Ogi10] Takeshi Ogita, *Accurate matrix factorization: inverse lu and inverse qr factorizations*, SIAM J. Math. Anal. Appli. **31** (2010), no. 5, 2477–2497.

[Ois98] Shin'ichi Oishi, *Finding all solutions of nonlinear systems of equations using linear programming with guaranteed accuracy*, J. Universal Computer Science **4** (1998), no. 2, 171–177.

[OO05] Takeshi Ogita and Shin'ichi Oishi, *Fast inclusion of interval matrix multiplication*, Reliable Computing **11** (2005), no. 3, 191–205.

[OO09] ______, *Fast verified solutions of linear systems*, Japan Journal of Industrial and Applied Mathematics **26** (2009), 169–190, 10.1007/BF03186530.

[OOO10] K. Ozaki, Takeshi Ogita, and Shin'ichi Oishi, *Tight and efficient enclosure of matrix multiplication by using optimized BLAS*, preprint available at `http://onlinelibrary.wiley.com/doi/10.1002/nla.724/pdf`.

[OOR09] Shi'ichi Oishi, Takeshi Ogita, and Segfried M. Rump, *Iterative Refinement for Ill-conditioned Linear Equations*, Japan J. Indust. Appl. Math. **26** (2009), no. 2, 465–476.

[ORO05] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi, *Accurate Sum and Dot Product*, SIAM J. Sci. Comput. **26** (2005), no. 6, 1955–1988.

[Pan84] Victor Pan, *How to multiply matrices faster*, Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[RK95] Jiří Rohn and Vladik Kreinovich, *Computing Exact Componentwise Bounds on Solutions of Linear Systems with Interval Data is NP-Hard*, SIAM J. Matrix Anal. Appl. **16** (1995), no. 2, 415–420.

[Roh93] Jiří Rohn, *Cheap and Tight Bounds: The Recent Result by E. Hansen Can Be Made More Efficient*, Interval Computations (1993), no. 4, 13–21.

[Roh96] ———, *Checking Properties of Interval Matrices*, Tech. Report 686, Czech Academy of Sciences, 1996, `http://uivtx.cs.cas.cz/~rohn/publist/92.pdf`.

[Roh05] ———, *A Handbook of Results on Interval Linear Problems*, 2nd ed., Czech Academy of Sciences, 2005, `http://www.cs.cas.cz/~rohn/publist/handbook`.

[RR05] Nathalie Revol and Fabrice Rouillier, *Motivations for an Arbitrary Precision Interval Arithmetic and the MPFI Library*, Reliable Computing **11** (2005), no. 4, 275–290.

[Rum] Siegfried M. Rump, *INTLAB - INTerval LABoratory*, `http://www.ti3.tu-hamburg.de/rump/intlab`.

[Rum80] ———, *Kleine Fehlerschranken bei Matrixproblemen*, Ph.D. thesis, Universität Karlsruhe, 1980.

[Rum91] ———, *On the solution of interval linear systems*, Computing **47** (1991), 337–353.

[Rum98] ———, *A note on epsilon-inflation*, Reliable Computing **4** (1998), 371–375.

[Rum99a] ———, *Fast And Parallel Interval Arithmetic*, BIT Numerical Mathematics **39** (1999), no. 3, 539–560.

[Rum99b] S.M. Rump, *INTLAB - INTerval LABoratory*, Developments in Reliable Computing (Tibor Csendes, ed.), Kluwer Academic Publishers, Dordrecht, 1999, `http://www.ti3.tu-harburg.de/rump/`, pp. 77–104.

[Rum05] Siegfried M. Rump, *Handbook on Accuracy and Reliability in Scientific Computation (edited by Bo Einarsson)*, ch. Computer-assisted Proofs and Self-validating Methods, pp. 195–240, SIAM, 2005.

[Rum09a] ———, *Error-Free Transformations and ill-conditioned problems*, International Workshop on Verified Computations and Related Topics, 2009.

[Rum09b] ———, *Inversion of extremely ill-conditioned matrices in floating-point*, Japan J. Indust. Appl. Math (JJIAM) **26** (2009), 249–277.

[RZBM09] Siegfried M. Rump, Paul Zimmermann, Sylvie Boldo, and Guillaume Melquiond, *Computing predecessor and successor in rounding to nearest*, BIT Numerical Mathematics **49** (2009), no. 2, 419–431.

[Ske79] Robert D. Skeel, *Scaling for Numerical Stability in Gaussian Elimination*, J. ACM **26** (1979), no. 3, 494–526.

[Ske80] ———, *Iterative Refinement Implies Numerical Stability for Gaussian Elimination*, Mathematics of Computation **35** (1980), no. 151, 817–832.

[Ste74] Pat H. Sterbenz, *Floating-point computation*, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

[Str69] Volker Strassen, *Gaussian elimination is not optimal*, Numer. Math. **13** (1969), 354–356.

[TAC] TACC, *GotoBLAS2*, `http://www.tacc.utexas.edu/tacc-projects/gotoblas2/`.

[vzGG03] J. von zur Gathen and J. Gerhard, *Modern computer algebra*, 2nd edition, Cambridge University Press, 2003.

[Wil60] J. H. Wilkinson, *Error analysis of floating-point computation*, Numerische Mathematik **2** (1960), no. 1, 319–340.

[Wil63] James H. Wilkinson, *Rounding errors in algebraic processes*, Prentice-Hall. Reprinted in 1994 by Dover Publications, Incorporated, 1963.

[WP05] R. Clint Whaley and Antoine Petitet, *Minimizing development and maintenance costs in supporting persistently optimized BLAS*, Software: Practice and Experience **35** (2005), no. 2, 101–121, `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.