PhD dissertation

József Smidla

Veszprém

2021

PhD dissertation

Numerically Stable Simplex Method Implementation

József Smidla

Supervisors:
István Maros, DSc
Tamás Terlaky, DSc

Doctoral School of Information Science and Technology
University of Pannonia

Veszprém

2021

NUMERICALLY STABLE SIMPLEX METHOD IMPLEMENTATION

Thesis for obtaining a PhD degree in the Doctoral School of Information Science and Technology of the University of Pannonia

in the branch of Information Sciences

Written by: József Smidla

Supervisors:
István Maros, DSc
Tamás Terlaky, DSc

Proposes acceptance (yes / no)

....
Supervisor(s)

As reviewer, I propose acceptance of the thesis:

Name of Reviewer: .... (yes / no)

....
Reviewer

Name of Reviewer: .... (yes /no)

....
Reviewer

The PhD-candidate has achieved .... % at the public discussion:
Veszprém, ...

....
Chairman of the Committee

The grade of the PhD Diploma ....
Veszprém, ...

....
Chairman of the UDHC

Abstract

The simplex method is one of the most important algorithms used to solve linear optimization problems (LO). Its development since the 1950s has been closely connected with the development of computer hardware and algorithms. While the first versions were only capable of solving small-scale problems, today's software occasionally can handle problems even with millions of decision variables and constraints. However, solution of large scale optimization models may be problematic due to several reasons. Some of them can be attributed to the unavoidable use of floating-point arithmetic required by the solution algorithms. Though considerable progress has been made to alleviate these problems it is not uncommon that we encounter models that resist to the solution attempt. Great progress has also been achieved in dealing with models involving various numerical difficulties, such as scaling, perturbation, anti-degeneration strategies. It is important that these methods do not significantly slow down the completion of the entire task. But even the current software is unable to solve all the linear optimization problems correctly, as there are numerical errors that we cannot handle with traditional number representation. Existing software can also give poor results, such as cannot solve a problem due to numerical difficulties or not converging to the optimum. This gave us the motivation to incorporate automatic features into our software (Pannon Optimizer) that can detect that the default floating-point number representation cannot solve the given model, so it is worth switching to more accurate arithmetic. This is a great help to the user as there is less doubt as to the correctness of the program output. In connection with this, we developed stable adder and dot-product implementations that do not contain conditional branching, so their computational overhead will be smaller. Moreover, we describe the acceleration method of one of the time consuming elementary steps of the simplex method. When discussing numerical algorithms, we provide information also on their speed efficiency.

Absztrakt

A szimplex módszer az egyik legfontosabb algoritmus, amelyet a lineáris optimalizálási problémák (LO) megoldására használnak. Fejlesztése az 1950-es évek óta szorosan összefügg a számítógépes hardverek és algoritmusok fejlesztésével. Míg az első verziók csak kisméretű problémák megoldására voltak képesek, a mai szoftverek alkalmanként képesek kezelni olyan problémákat is, amelyek döntési változók millióiból és korlátozásaiból állnak. A nagyméretű optimalizálási modellek megoldása azonban több okból is problematikus lehet. Ezek egy része a lebegőpontos aritmetika elkerülhetetlen használatának tulajdonítható, amelyet a megoldó algoritmusok megkövetelnek. Bár jelentős előrelépés történt e problémák enyhítése érdekében, nem ritka, hogy olyan modellekkel találkozunk, amelyek ellenállnak a megoldási kísérletnek. Nagy előrelépés történt a különféle numerikus nehézségekkel járó modellek kezelése terén is, például skálázás, perturbálás, degeneráció-ellenes stratégiák terén. Fontos, hogy ezek a módszerek ne lassítsák jelentősen az egész feladat megoldási folyamatát. De még a jelenlegi szoftverek sem képesek minden lineáris optimalizálási problémát helyesen megoldani, mivel vannak olyan numerikus hibák, amelyeket a hagyományos számábrázolással nem tudunk kezelni. A meglévő szoftverek gyenge eredményeket is adhatnak, például numerikus nehézségek miatt nem tudnak megoldani egy problémát, vagy nem konvergálnak az optimumhoz. Ez adta a motivációt arra, hogy szoftverünkbe (Pannon Optimizer) beépítsünk olyan automatikus eljárásokat, amelyek észlelhetik, hogy az alapértelmezett lebegőpontos ábrázolás nem tudja megoldani az adott modellt, ezért érdemes átállni a pontosabb számábrázolásra. Ez nagy segítség a felhasználó számára, mivel kevesebb kétség merül fel a program kimenetének helyességével kapcsolatban. Ennek kapcsán olyan stabil összeadó és skalár szorzat implementációkat fejlesztettünk ki, amelyek nem tartalmaznak feltételes elágazást, így számítási költségük kisebb lesz. Ezenkívül leírjuk a szimplex módszer egyik időigényes elemi lépésének gyorsítási módszerét. A numerikus algoritmusok tárgyalásakor ismertetjük a mérési eredményeinket is.

Abstrakt

Simplexová metóda je jedným z najdôležitejších algoritmov používaných na riešenie problémov s lineárnou optimalizáciou (LO). Jeho vývoj od 50. rokov 20. storočia bol úzko spätý s vývojom počítačového hardvéru a algoritmov. Zatiaľ čo prvé verzie boli schopné riešiť iba problémy malého rozsahu, dnešný softvér občas dokáže zvládnuť problémy aj s miliónmi rozhodovacích premenných a obmedzení. Riešenie veľkých optimalizačných modelov však môže byť problematické z niekoľkých dôvodov. Niektoré z nich možno pripísať nevyhnutnému použitiu aritmetiky s pohyblivou rádovou čiarkou, ktorú vyžadujú algoritmy riešenia. Napriek tomu, že na zmiernenie týchto problémov bol urobený značný pokrok, nie je neobvyklé, že sa stretávame s modelmi, ktoré odolávajú pokusu o riešenie. Veľký pokrok sa dosiahol aj pri riešení modelov zahŕňajúcich rôzne numerické ťažkosti, ako sú škálovanie, poruchy, stratégie proti degenerácii. Je dôležité, aby tieto metódy výrazne nespomaľovali dokončenie celej úlohy. Dokonca ani súčasný softvér nie je schopný správne vyriešiť všetky problémy s lineárnou optimalizáciou, pretože existujú numerické chyby, ktoré si s tradičným zobrazovaním čísel nevieme rady. Existujúci softvér môže tiež poskytovať zlé výsledky, napríklad nemôže vyriešiť problém z dôvodu numerických problémov alebo nesúladu s optimom. To nás motivovalo začleniť do nášho softvéru (Pannon Optimizer) automatické funkcie, ktoré dokážu zistiť, že predvolená reprezentácia čísla s pohyblivou rádovou čiarkou nemôže vyriešiť daný model, preto stojí za to prejsť na presnejšiu aritmetiku. Je to veľká pomoc pre užívateľa, pretože už nie sú žiadne pochybnosti o správnosti výstupu programu. V súvislosti s tým sme vyvinuli stabilné implementácie sčítačky a skalárneho súčinu, ktoré neobsahujú podmienené vetvenie, takže ich výpočtová réžia bude menšia. Okrem toho popisujeme akceleračnú metódu jedného z časovo náročných elementárnych krokov simplexovej metódy. Pri diskusii o numerických algoritmoch poskytujeme informácie aj o ich rýchlosti.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

Notations

Linear optimization models

m	Number of constraints
n	Number of decision variables, dimension of $\mathbf{x}$
$\mathbf{x}$	Vector of decision variables
x_j	Decision variable j
ℓ	Vector of lower bounds of variables
$\mathbf{u}$	Vector of the upper bounds of variables
$\mathbf{c}$	Coefficient vector of the objective function
z	Value of objective function
$\mathbf{L}$	Vector of the lower bounds of constraints
$\mathbf{U}$	Vector of the upper bounds of constraints
a_j^i	The jth coefficient in the ith constraint
$\mathbf{A}$	Matrix of constraints
χ	Set of feasible solutions

Simplex method

$\mathbf{y}$	Vector of logical variables
$\mathbf{I}$	Identity matrix
$\mathbf{b}$	Right hand side of constraints in transformed computational form
$\mathbf{B}$	Basis part of $\mathbf{A}$
$\mathbf{R}$	Nonbasis part of $\mathbf{A}$
$\mathcal{N}$	Set of indices of variables
$\mathcal{B}$	Set of indices of basic variables

$\mathcal{R}$ Set of indices of nonbasic variables

$\mathcal{U}$ Set of indices of nonbasic variables, at upper bound

$\mathbf{x}_{\mathcal{B}}$ Vector of basic variables

$\mathbf{x}_{\mathcal{R}}$ Vector of nonbasic variables

$\mathbf{c}_{\mathcal{B}}$ Coefficient vector of basic variables

$\mathbf{c}_{\mathcal{R}}$ Coefficient vector of nonbasic variables

β Vector of values of basic variables

$\mathbf{B}^{-1}$ Inverse of the basis

$\boldsymbol{\alpha}_j$ The jth transformed column of matrix $\mathbf{A}$

θ Steplength of a nonbasic variable

w Measure of infeasibility

$\mathcal{M}$ Set of indices of basic variables below their lower bound

$\mathcal{P}$ Set of indices of basic variables above their upper bound

d_j Reduced cost of the jth variable

$\bar{z}$ Value of updated objective value

η Non-unit column of an elementary transformation matrix

$\mathbf{E}$ Elementary transformation matrix

s number of the ETMs

$\hat{d}_j$ Normalized reduced cost of the jth variable

$\boldsymbol{\sigma}_j$ Direction of changing $\mathbf{x}$, when x_j changes by 1

$\mathbf{h}$ Auxiliary vector

ϕ Simplex multiplier in phase-1

π Simplex multiplier in phase-2

$\boldsymbol{\rho}^p$ The pth row of inverse of the basis

$\bar{\boldsymbol{\rho}}^p$ The updated p^{th} row of basis inverse

Numerical stability

$\kappa(\mathbf{M})$ Condition number of the matrix $\mathbf{M}$

Other symbols

$\mathbb{R}$ Set of real numbers

$\|\mathbf{v}\|_2$ Euclidean norm of vector $\mathbf{v}$

$\mathbf{e}_j$ Unit vector

Text formatting

In this dissertation we follow the following text formatting conventions:

- The first occurrences of important keywords are in *italics*.

- C ++ source codes, source code snippets, program variables, and program outputs are formatted by `typewriter` font:

```cpp
1  #include <iostream>
2
3  int sum(int a, int b) {
4      if (!a) {
5          return b;
6      }
7      return sum((a & b) << 1, a ^ b);
8  }
9
10 int main() {
11     std::cout << "The sum: " << sum(10, 20) << std::endl;
12     return 0;
13 }
```

The output:

```
The sum: 30
```

- The mathematical expressions, formulas, variables, symbols are in mathematical form: $e^{i\pi} + 1 = 0$

Preface

Over the past 70 years, linear optimization has remained one of the essential tools in operations research. Its use is critical in proper economic decision-making processes, so its research results in significant benefits. Its development is closely related to the development of computer technology, so with more advanced technology we can solve more problems more efficiently. This supports faster decision-making process.

I have come across a few times a mathematical model (Rump's matrices, discrete moment problems, or some industrial models) for which existing solver software (like GLPK and COIN-OR) gave the result that it is infeasible or unbounded while examining the models revealed that this is not the case: the software gave a wrong result. The problem was caused by the numerical instability of the coefficient matrix. This gave me the motivation to incorporate automatic features into our software (Pannon Optimizer) that can detect that the default floating-point number representation cannot solve the given model, so it is worth switching to more accurate arithmetic. This is a great help to the user as there is less doubt as to the correctness of the program output. Related to this, I also turned to speed up stable operations: during a professional conversation, a developer of a well-known software explained to me that they don't use stable addition algorithms because they interfere with the CPU pipeline mechanism. The idea of a C-language implementation of a branchless dot-product came to me at this time, and then I developed further implementations as well.

Introduction

The simplex method is one of the most important algorithms used to solve linear optimization problems. While the first computer implementations in the 1950s were only capable of solving small-scale problems, today's software occasionally can handle problems even with millions of decision variables and constraints. But even though many algorithms have been added to the original simplex method with which we can handle many numerical problems, even today's solver software is not able to solve all the problems. Therefore, our primary goal was to create a new algorithm that allows the software to detect that it is working on a numerically too unstable problem and can inform the user accordingly. To present our work, we discuss the simplex method to the necessary depth in Chapter 1, and we shortly introduce our simplex implementation named Pannon Optimizer. This method is also forced to use fractional numbers, and processors support floating-point number representation, and it is best suited for scientific calculations. Since this number representation is the cause of the numerical problems examined in the dissertation, we will describe them in Chapter 2. In the later chapters, we will build heavily on these properties and number representation: To understand the numerical problems described in the later chapters, it is necessary to know the properties and problems of floating-point numbers and the IEEE 754 Standard. In Chapter 3, we present the acceleration of the most time-consuming and numerically critical step of the simplex method. In Chapter 4, we describe the condition numbers of the matrices and then prove that it is not enough for our solving software to draw conclusions from the numerical properties of the input matrix. We then propose a method by which the solving software adapts to the numerical properties of the problem that has to be solved and, if necessary, checks the numerical quality of the current inverse of the basis representation. If the checker finds that the current inverse of the basis is numerically unusable, this indicates that the software can then switch to a slower but a more reliable number representation. As a result, unlike the solving software that is used currently, the user does not get false results. Finally, in Chapter 5, we describe a few low-level, numerically stable vector-vector adder and dot-product implementations based on Intel's SIMD instruction set which are much faster than the simple C++ implementations. We also describe and examine a simplified but faster version of Orchard-Hays's classic stable adder. We use measurements to prove that our own simplified algorithm is faster, and we still get correct result.

Appendix A

3. Improvement techniques of
pricing

1. The simplex method   2. Floating-point numbers

4. Numerical stability

Appendix B   5. Low-level optimizations

Appendix C   Appendix D   Appendix E
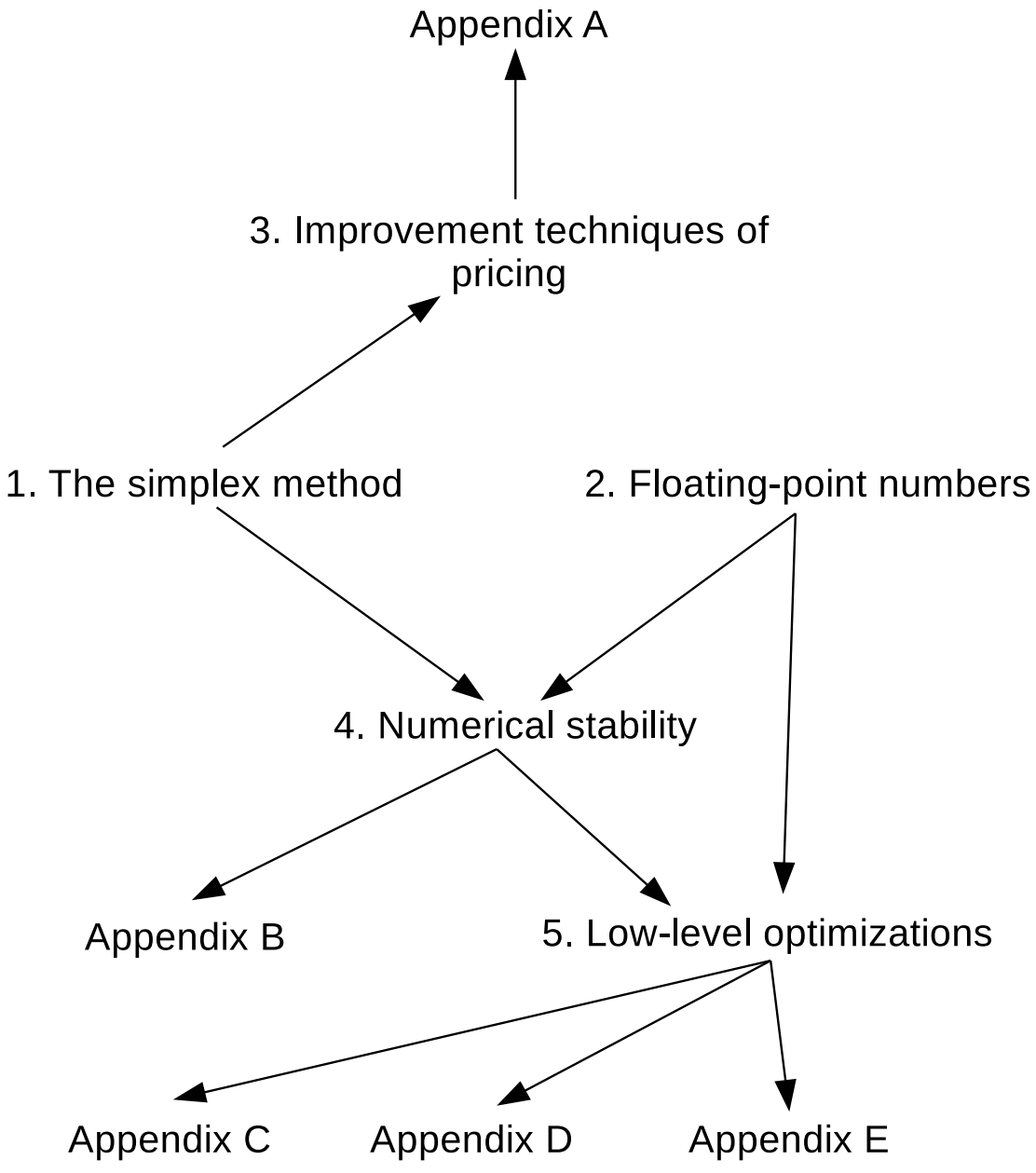
Figure 1: Connections between the chapters

Chapter 1

The simplex method

In this chapter, we introduce the linear programming problems and the simplex method.

1.1 Short history

The history of linear programming began in 1939 [21], when the Soviet *Kantorovich* published his method that solved a general linear programming problem [43] He used this method to optimize military operations during the war. However, his work was neglected in the Soviet Union, his results only became known in the West in 1959 [16]. Meanwhile, *Dantzig* independently developed the formulation of the general linear programming problems in the US Air Force [18] in 1947 and he later published the simplex method [14]. John von Neumann had conjectured the theorem of duality at a meeting with Dantzig [18].

It seemed that the simplex method is a polynomial algorithm, but in 1972 Klee and Minty proved that in specific problems the simplex method is exponential [47]. Khachiyan published his ellipsoid method in 1979, which was polynomial algorithm [45]. However, this algorithm was slower in most of the linear programming problems. In 1984 Karmarkar introduced the interior-point method which was a highly efficient, polynomial algorithm [44]. Later, in 1985 and 1987 Terlaky published the Criss-Cross algorihm [96], [97].

1.2 Linear optimization problems

In *linear optimization* (or *linear programming*), modelers describe problems with linear expressions, linear inequalities. The solution to the problem is represented by numerical values, so we need an *n*-dimensional vector that contains the *decision variables*, denoted by $\mathbf{x}$. Each variable has an individual *lower bound* and *upper bound* from the set of $\{-\infty, \mathbb{R}\}$ and $\{+\infty, \mathbb{R}\}$ respectively. If the variable satisfies its bounding criterion, then it is a *feasible variable*. The n dimensional ℓ vector contains the lower bounds and $\mathbf{u}$ consists of upper bounds. We can say that the variable has no lower / upper bound if the corresponding bound is $-\infty$ / $+\infty$. Obviously, $\ell_j \leq u_j$.

$$\ell_j \leq x_j \leq u_j, j = 1 \ldots n \tag{1.1}$$

The value we wish to minimize or maximize is the *objective function* which is the linear combination of the decision variables. The n dimensional *cost vector* **c** contains the coefficients of the variables in the objective function, so formally the objective function is the following:

$$\min z = \sum_{j=1}^{n} c_j x_j \tag{1.2}$$

or we can define a maximization problem:

$$\max z = \sum_{j=1}^{n} c_j x_j \tag{1.3}$$

But it is known, that there is a simple connection between minimization and maximization problems:

$$\min z = -\max - z \tag{1.4}$$

Therefore, throughout the rest of this dissertation we can talk about minimization problems without loss of generality.

If we would like to solve real-life problems, we have to take into consideration certain restrictions: The problems have constraints too, which are expressed by linear inequalities constructed from **x**: For the linear combination of **x** we can give lower and upper bounds. Let m denote the number of constrains. We can write the constraints formally as follows:

$$L_1 \leq x_1 a_1^1 + \ldots + x_n a_n^1 \leq U_1$$

$$\vdots$$

$$L_i \leq x_1 a_1^i + \ldots + x_n a_n^i \leq U_i$$

$$\vdots$$

$$L_m \leq x_1 a_1^m + \ldots + x_n a_n^m \leq U_m$$

where the m dimensional vectors **L** and **U** contain the lower and upper bounds of the constraints. The a_j^i is the coefficient of jth variable in i^{th} constraint where $1 \leq i \leq m$ and $1 \leq j \leq n$. The matrix $\mathbf{A}^{m \times n}$ contains these coefficients. The *linear optimization model* is composed of the variables, the objective function and the constraints:

$$\min z = \quad \mathbf{c}^\mathbf{T}\mathbf{x} \tag{1.5}$$

$$s.t. \quad \mathbf{L} \le \mathbf{Ax} \le \mathbf{U} \tag{1.6}$$

$$\ell \le \mathbf{x} \le \mathbf{u} \tag{1.7}$$

If $\mathbf{x}$ satisfies the inequality system (1.6), it is a *solution*. If $\mathbf{x}$ is a solution and satisfies (1.7), then it is a *feasible solution*. The χ denotes the set of feasible solutions:

$$\chi = \{\mathbf{x} : \mathbf{L} \le \mathbf{Ax} \le \mathbf{U}, \ell \le \mathbf{x} \le \mathbf{u}\} \tag{1.8}$$

The problem has no feasible solution, if and only if $\chi = \emptyset$. The solution $\mathbf{x} \in \chi$ is *optimal*, if: $\forall \mathbf{x}' \in \chi$: $\mathbf{c}^T\mathbf{x} \le \mathbf{c}^T\mathbf{x}'$.

1.3 The simplex method

In this section, we introduce the *simplex method*. First, we describe the computational form of the linear optimization model as well as the principle of solver algorithm, and finally, we briefly explain the modules . From a mathematical point of view, there are two versions of the simplex method: The *primal* [14] and *dual* [55]. We notice that *Maros* introduced a general dual phase-2 algorithm in 1998 [60], and later in 2003 the general dual phase-1 [59], which iterate faster to the optimum. In this dissertation we only discuss the primal method in details.

Previously, we saw the general form of linear optimization problems in (1.5) - (1.7), but we need to transform this into a simpler but equivalent form. There are two reasons for this transformation: First, it makes the introduction of the algorithms easier to discuss, and second, the source code of the software becomes simpler.

Several *computational forms* exist, and they have a common property: The constraints are transformed into equality by introducing *logical variables*. These special variables are contained by the m dimensional vector $\mathbf{y}$. In some forms, we need to translate the lower and upper bounds of variables, too. The transformed computational form we will use in this dissertation is the following:

$$\min z = \quad \mathbf{c}^\mathbf{T}\mathbf{x} \tag{1.9}$$

$$s.t. \quad \mathbf{Ax} + \mathbf{Iy} = \mathbf{b} \tag{1.10}$$

$$\text{type}(x_j), \text{type}(y_i) \in \{0, 1, 2, 3\}, 1 \le j \le n, 1 \le i \le m \tag{1.11}$$

In the applied computational form, we allow the following types of variables: Hereafter, we will use as a *coefficient matrix* as follows:

$$\mathbf{A} := [\mathbf{A} \,|\, \mathbf{I}]$$

Feasibility range	Type	Reference
$y_j, x_j = 0$	0	*Fixed variable*
$0 \leq y_j, x_j \leq u_j$	1	*Bounded variable*
$0 \leq y_j, x_j \leq +\infty$	2	*Nonnegative variable*
$-\infty \leq y_j, x_j \leq +\infty$	3	*Free variable*

Table 1.1: Types of variables

and moreover the

$$\mathbf{x} := \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix}$$

vector will contain the variables, and the cost coefficients belonging to the logical variables are zero:

$$\mathbf{c} := \begin{bmatrix} \mathbf{c} \\ \mathbf{0} \end{bmatrix}$$

For the simplex method, we partition the $\mathbf{A}$ matrix into two sections, the $\mathbf{B}$, *basis* and the $\mathbf{R}$ submatrix, which contains columns from $\mathbf{A}$ that are not in $\mathbf{B}$:

$$\mathbf{A} = [\mathbf{B} \mid \mathbf{R}]$$

For notational simplicity, we need some index sets. Let $\mathcal{N}$ denote the index set of the variables. Each variable is associated with a column of matrix $\mathbf{A}$. We say that when a column is in the $\mathbf{B}$ basis, then the corresponding variable is also in the basis . These are the *basic variables*, others are the *nonbasic variables*. The set $\mathcal{B}$ contains the indices of basic variables, while $\mathcal{R}$ contains indices of nonbasic variables. Moreover, set $\mathcal{U}$ denotes the indices of nonbasic variables, which are at their upper bound. From the previous observation, we can split $\mathbf{x}$ into basic variables and nonbasic variables, respectively:

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_{\mathcal{B}} \\ \mathbf{x}_{\mathcal{R}} \end{bmatrix}$$

Similarly, we also split $\mathbf{c}$:

$$\mathbf{c} = \begin{bmatrix} \mathbf{c}_{\mathcal{B}} \\ \mathbf{c}_{\mathcal{R}} \end{bmatrix}$$

We can show that in an optimal solution, the nonbasic variables are at their bounds, and the values of basic variables are determined by the nonbasic variables as follows:

$$\beta = \mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}_{\mathcal{R}}) \tag{1.12}$$

We need the following notation: Let $\boldsymbol{\alpha}_j$ denote the product of jth matrix column and basis inverse:

$$\boldsymbol{\alpha}_j = \mathbf{B}^{-1}\mathbf{a}_j \tag{1.13}$$

Two bases are called neighboring bases when they differ from each other in only one column. We can move from a basis to a neighboring one by a so-called *basis change*. The principle of the simplex method is the following: Determine a *starting basis* and search the optimal solution through a finite number of basis changes. The optimality of a solution can be verified easily; the simplex method guarantees that it is a global optimum. Usually, when the algorithm starts from an infeasible solution, the so-called *primal phase-1* controls the basis changes: Its task is to find a feasible solution. Maros introduced a phase-1 algorithm in 1986 [58]. After a feasible solution is found, variables are kept in feasible ranges by *primal phase-2*, while searching for an optimal solution. For the sake of simplicity, we focus on phase-2 in this dissertation.

In each iteration, the simplex algorithm affects the value of a chosen nonbasic variable, and in many cases, this variable moves into the basis. Let x_q be the chosen nonbasic variable, and θ denotes the *steplength* of the change in x_q. In this case, the basic variables change as shown by the following formula:

$$\overline{\beta} = \beta - \theta\boldsymbol{\alpha}_q \tag{1.14}$$

Since θ affects feasibility, during the computation of θ we have to take into consideration that the solution has to stay feasible.

Now we introduce the major steps of the simplex algorithm. Figure (1.1) shows these steps and their logical connections.

MPS reader

Most of the linear optimization problems can be found in standardized text files. There are numerous file formats, and the most common file format is the *MPS*. Every simplex implementation has to support the MPS (Mathematical Programming System) format. The MPS was the first format designed for linear optimization systems at the age of punched cards, so this caused a very rigid storage method. Figure (1.2) shows a simple example LO problem and its MPS representation. Notice that the cost coefficients are multiplied by -1 according to (1.4) because the MPS format describes only maximization problems. Later on, technical development allowed using other, advanced, and more sophisticated formats, but there were too many models in MPS, so this legacy format remained the de-facto standard format of LP models. However, the usage of this format has declined, because of the wide acceptance of the algebraic modeling languages, like AMPL [42] and GAMS [24]. The task of the MPS reader is to read these files, detect their errors, and compute statistics. The result of this module is the raw model, which is stored in the memory. The following software modules will perform transformation operations and computations on this raw model.
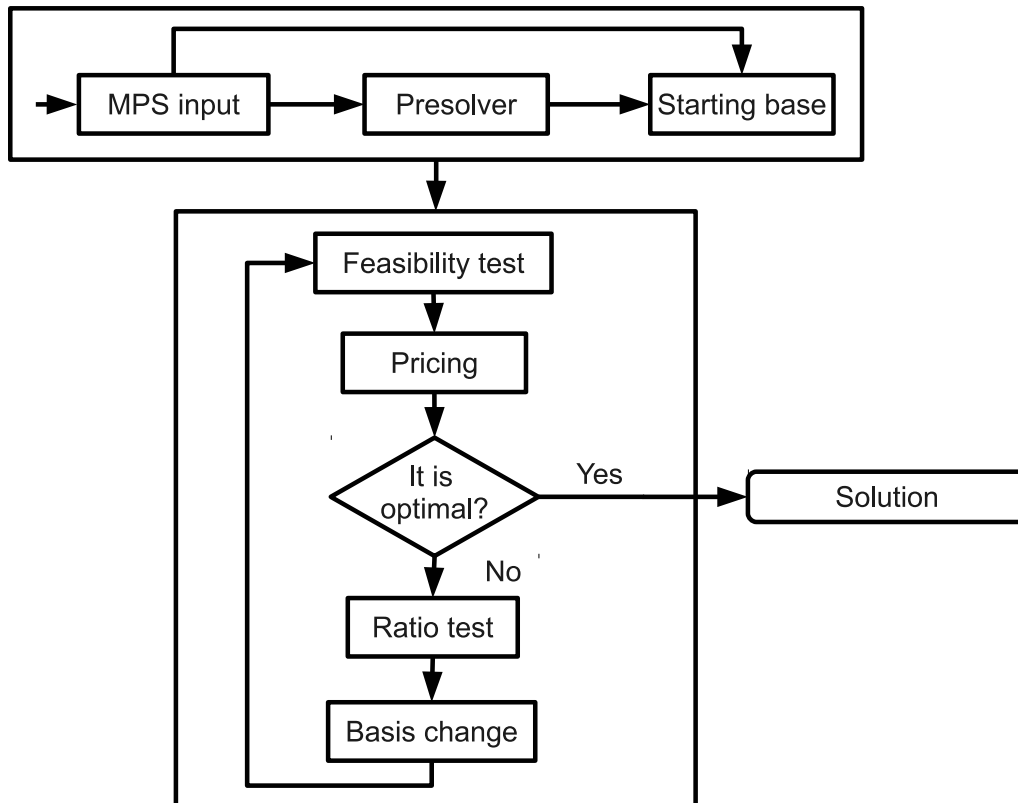
Figure 1.1: Flow chart of the simplex method

Presolver

From the mathematical point of view, the *presolver* is not a necessary module of the simplex algorithm. The task of the presolver is simplification and improvement of the model. Since some generator algorithms create many models, the model may be redundant. The presolver algorithm has to recognize the removable columns and rows in such a way that the resulting model is equivalent to the original one.

For example, a basic presolver can simplify the LO model introduced formerly in Figure (1.2); it recognizes that the 3rd row is unnecessary; that is, it defines an upper bound for the variable x_2. The presolver removes the 3rd constraint and modifies the upper bound of x_2, as it can be seen in Figure (1.3). In the example, the coefficient matrix's size decreased; therefore, the simplex algorithm can work faster.

After the solver algorithm found an optimal solution, the so-called postsolver (the pair of presolver) has to insert the removed variables back into the model to represent the solution for the user by the original model. The first publication about the presolver was the work of *Brearley*, *Mitra* and *Williams* in 1975 [5], it was followed by *Tomlin* and *Welch*'s article in 1983 [101, 100], Andersen and *Andersen* in 1995 [3], and *Gondzio* in 1997 [29]. The last two results apply to interior-point methods.

$$\max z : 600x_1 + 800x_2$$
$$40x_1 + 20x_2 \leq 550$$
$$32x_1 + 64x_2 \leq 150$$
$$48x_2 \leq 120$$
$$x_1, x_2 \geq 0$$

```
NAME            demo
ROWS
 L   c1
 L   c2
 L   c3
 N   z
COLUMNS
     x1         c1                    40   c2                        32
     x1         z                    600
     x2         c1                    20   c2                        64
     x2         c3                    48   z                        800
RHS
     RHS1       c1                   550   c2                       150
     RHS1       c3                   120
ENDATA
```

Figure 1.2: A simple linear optimization problem, and its MPS representation

Scaler

Like the presolver, the *scaler* is also an optional module, executed before the simplex algorithm. There are LO problems, where the variance of the entries of **A** is too large, so this can increase the numerical instability of the algorithm. The scaler created to stabilize the numerically unstable problems; scales the rows and columns of the matrix to decrease the variance as much as possible. We scaled our example model in a way that is using powers of 2, to avoid additional numerical errors, using the method that was proposed by Benichou in [4]; the variance of matrix elements is reduced. Figure (1.4) shows the result.

Finding the starting basis

As we saw in Figure (1.1), before the iterative part of the algorithm, we need to select a reasonable starting basis. There are numerous ways to select a starting basis. The most straightforward way is to select the identity matrix; namely, we select only logical variables into the basis. This is called a *logical basis*. The logical basis is simple and easy to implement, but we can construct bases that are probably closer to an optimal solution, so researchers have designed many starting basis finder algorithms, with different ad-

$$\begin{aligned}
\max z : 600x_1 + 800x_2 \\
40x_1 + 20x_2 &\leq 550 \\
32x_1 + 64x_2 &\leq 150 \\
48x_2 &\leq 120 \\
x_1, x_2 &\geq 0
\end{aligned}
\implies
\begin{aligned}
\max z : 600x_1 + 800x_2 \\
40x_1 + 20x_2 &\leq 550 \\
32x_1 + 64x_2 &\leq 150 \\
x_1 \geq 0, 0 \leq x_2 &\leq 2.5
\end{aligned}$$

Figure 1.3: The simplified model after the presolver

$$\begin{aligned}
\max z : 600x_1 + 800x_2 \\
40x_1 + 20x_2 &\leq 550 \\
32x_1 + 64x_2 &\leq 150 \\
x_1 \geq 0, 0 \leq x_2 &\leq 2.5
\end{aligned}
\implies
\begin{aligned}
\max z : 600x_1 + 800x_2 \\
1.25x_1 + 0.625x_2 &\leq 17.1875 \\
0.5x_1 + x_2 &\leq 2.34375 \\
x_1 \geq 0, 0 \leq x_2 &\leq 2.5
\end{aligned}$$

Figure 1.4: The scaled model

vantages; minimizing the infeasibility, degeneracy, and so on. The algorithm also creates the logical variables at least here, and converts $\leq$ and $\geq$ constraints into = equations, as Figure (1.5) shows. Finally, this module initializes the values of nonbasic variables into one of their bound.

$$\begin{aligned}
\max z: \quad & 600x_1 + 800x_2 \\
\underline{y_1} \quad &+ \quad 1.25x_1 \quad + \quad 0.625x_2 \quad = \quad 17.1875 \\
\underline{y_2} \quad &+ \quad 0.5x_1 \quad + \quad x_2 \quad = \quad 2.34375 \\
& x_1 \geq 0, 0 \leq x_2 \leq 2.5, y_1, y_2 \geq 0
\end{aligned}$$

Figure 1.5: The model with logical variables. In this example, y_1 and y_2 are the initial basic variables.

The simplex algorithm executes the following modules in every iteration, so their implementations have to be very efficient.

Feasibility test

The values of the nonbasic variables determine the values of basic variables, as (1.12) shows. The solver has to check whether these values violate their feasibility range or not; that is, whether the solution is feasible or not. When the solution is infeasible, this module prepares special data structures for phase-1 algorithms. In theory, we skip this step in phase-2, but practical experiences show that we have to perform a *feasibility test* in both phases: Numerical errors can occur and lead to infeasible solutions, pushing back the solver to phase-1.

Let $\mathcal{M}$ and $\mathcal{P}$ denote the index sets of infeasible basic positions, where $i \in \mathcal{M}$, if the ith basic variable is below its lower bound, and $i \in \mathcal{P}$ if the ith basic variable is above its

upper bound. We need a phase-1 objective function, which summarizes these feasibility range violations:

$$w = \sum_{i \in \mathcal{M}} \beta_i - \sum_{i \in \mathcal{P}} (\beta_i - v_i)$$

When $w < 0$, then the actual solution is infeasible, because there is a basic variable, which is below its lower bound or above its upper bound. The task of pricing is to choose a nonbasic variable for increasing the value of w. We can calculate a *phase-1 reduced cost* for each nonbasic variable:

$$d_j = \sum_{i \in \mathcal{M}} \alpha^i_j - \sum_{i \in \mathcal{P}} \alpha^i_j \tag{1.15}$$

The meaning of this reduced cost is the following: When x_q is increased by t, but $\mathcal{M}$ and $\mathcal{P}$ still remain unchanged, then the value of w changes by $-td_q$: $\Delta w = -td_q$. When t is negative, pricing has to choose x_q in such a way that $d_q < 0$, but when t is positive, we need a positive d_q. We summarized the choosing rules for different cases in table (1.2): When the pricing cannot find an appropriate nonbasic variable and $w < 0$, then the

Type	Value	d_j	Improving displacement	Remark
0	$x_j = 0$	Irrelevant	0	Never enters to basis
1	$x_j = 0$	< 0	$+$	
1	$x_j = u_j$	> 0	$-$	$j \in \mathcal{U}$
2	$x_j = 0$	< 0	$+$	
3	$x_j = 0$	$\neq 0$	$+/-$	

Table 1.2: Rules of finding an improving variable in phase-1

algorithm halts, the problem has no solution.

Pricing

In the primal simplex, the *pricing* selects a nonbasic variable x_k candidate to enter the basis; we say that it is an incoming variable. With moving the incoming variable, the objective function has to improve. We compute a *phase-2 reduced cost* for each $j \in \mathcal{R}$ nonbasic variable based on the following formula:

$$d_j = c_j - \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \mathbf{a}_j \tag{1.16}$$

The Formula (1.16) comes from these:

$$\mathbf{z} = \mathbf{c}_{\mathcal{B}}^T \mathbf{x}_{\mathcal{B}} + \mathbf{c}_{\mathcal{R}}^T \mathbf{x}_{\mathcal{R}} \tag{1.17}$$

$$\mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}_{\mathcal{R}}) \tag{1.18}$$

If we mix these formulas and rearrange them, we get the following:

$$z = \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \mathbf{b} + (\mathbf{c}_{\mathcal{R}}^T - \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \mathbf{R}) \mathbf{x_R} \tag{1.19}$$

Since according to (1.19) $z = \mathbf{c}^T \mathbf{x}$, , if $x_k (k \in \mathcal{R})$ changes by θ, then

$$z(\theta) = z + \theta d_k \tag{1.20}$$

As (1.20) shows, the reduced costs determine the direction of the objective function's change. Therefore, we can introduce the so-called optimality conditions; if and only if there are no more improving variables (see Table (1.3)), the current solution is optimal.

Type	Value	d_j	Feasible displacement	Remark
0	$x_j = 0$	Irrelevant	$t = 0$	
1	$x_j = 0$	< 0	$0 \leq t \leq u_k$	
1	$x_j = u_j$	> 0	$-u_k \leq t \leq 0$	$j \in \mathcal{U}$
2	$x_j = 0$	< 0	$t \geq 0$	
3	$x_j = 0$	$\neq 0$	$-\infty \leq t \leq \infty$	

Table 1.3: Rules of finding an improving variable in phase-2, with a minimization objective function. Obviously, the maximization problems have opposite rules.

In our example, the starting basis consists of the logical variables:

$$\mathcal{B} = (y_1, y_2) \Rightarrow \mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \mathbf{c}_{\mathcal{B}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\mathcal{R} = (x_1, x_2) \Rightarrow \mathbf{R} = \begin{bmatrix} 1.25 & 0.625 \\ 0.5 & 1 \end{bmatrix}, \mathbf{c}_{\mathcal{R}} = \begin{bmatrix} 600 \\ 800 \end{bmatrix}$$

The right-hand side vector is constant:

$$\mathbf{b} = \begin{bmatrix} 17.1875 \\ 2.34375 \end{bmatrix}$$

The nonbasic variables are in their bound:

$$\mathbf{x}_{\mathcal{R}} = \begin{bmatrix} x_1 = 0 \\ x_2 = 0 \end{bmatrix}$$

Based on Formula (1.12), we can compute the basic variables:

$$\beta = \mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}_{\mathcal{R}}) = \begin{bmatrix} y_1 = 17.1875 \\ y_2 = 2.34375 \end{bmatrix}$$

Obviously, the initial z value is 0.

The variables satisfy their bound criterion, so this is a feasible solution; we are in phase-2, we can calculate the phase-2 reduced costs based on Formula (1.16):

$$\mathbf{d}^T = \mathbf{c}_{\mathcal{R}}^T - \mathbf{c}_{\mathcal{B}}^T\mathbf{B}^{-1}\mathbf{R} = \begin{bmatrix} d_{x_1} = 600 \\ d_{x_2} = 800 \end{bmatrix}$$

As Table (1.3) shows (recall that we have a maximization problem now), both nonbasic variables are good candidates to enter the basis. There are numerous variants of pricing strategies, the simplest one is the Dantzig pricing [14]: We choose the variable to have the largest reduced cost. However, there are other advanced strategies as well. The most common methods are the normalized pricing strategies; Harris introduced the Devex in 1973 [30], and Goldfarb and Reid developed the Steepest-edge in 1977 [28].

In our case, we simply choose x_2 to enter the basis, its reduced cost is 800, and let denote $q = 2$ the incoming variable's index. As the corresponding

$$\alpha_q = \mathbf{B}^{-1}\mathbf{a}_q = \begin{bmatrix} 0.625 \\ 1 \end{bmatrix},$$

we have to choose a basic variable, which leaves the basis.

Ratio test

The pricing selects a variable to enter the basis. The *ratio test* has to determine the outgoing basic variable in a way that the feasibility ranges are maintained in phase-2, or decrease feasibility violations in phase-1. But sometimes it happens that the basis change is not possible. In this case, we perform a bound flip operation; change the incoming variable from its lower bound to the upper one, or vice versa.

Recall Formula (1.14), and keep in mind that the simplex algorithm has to modify the basic variables in a way that they still have to stay inside their feasible range:

$$l_i \le \beta_i - \Theta\alpha_q^i \le u_i$$

where l_i and u_i are the lower and upper bounds of the ith basic variable. We have several such inequalities, but only one Θ that we are looking for; we have to solve these inequalities to obtain different possible Θ values, and finally we choose their minimum. If we select this value properly, this displacement moves one basic variable into its bound, and that variable leaves the basis.

In our example there are 2 basic variables, the y_1 and y_2, and they have only a lower bound, so there are 2 inequalities:

$$0 \le \beta_1 - \Theta_1 \alpha_q^1 \to 0 \le 17.1875 - \Theta_1 0.625 \to \Theta_1 = 27.5$$

$$0 \le \beta_2 - \Theta_2 \alpha_q^2 \to 0 \le 2.34375 - \Theta_2 \to \underline{\Theta_2 = 2.34375}$$

The Θ is the Θ_2, and the outgoing variable is the y_2. We say that, this variable is the pivot element, and $p = 2$.

Basis change

This module performs a *basis change*: It removes the outgoing variable from the basis, and inserts the incoming variable. Finally, it updates a few data structures, for example, the values of basic variables. When it is necessary, this module performs the *reinversion* of the basis. For higher efficiency, we can integrate the feasibility test in this module.

As saw in the previous subsections, in the example problem, the incoming variable is the x_2, and y_2 leaves the basis. So this step modifies the list of basic and nonbasic variables, and updates other data structures:

$$\mathcal{B} = (y_1, x_2) \Rightarrow \mathbf{B} = \begin{bmatrix} 1 & 0.625 \\ 0 & 1 \end{bmatrix}, \mathbf{c}_{\mathcal{B}} = \begin{bmatrix} 0 \\ 800 \end{bmatrix}$$

$$\mathcal{R} = (x_1, y_2) \Rightarrow \mathbf{R} = \begin{bmatrix} 1.25 & 0 \\ 0.5 & 1 \end{bmatrix}, \mathbf{c}_{\mathcal{R}} = \begin{bmatrix} 600 \\ 0 \end{bmatrix}$$

We update the basic variables according to Formula (1.14), and the basic variable at the pivot position changed by Θ :

$$\overline{\beta} = \overline{\mathbf{x}}_{\mathcal{B}} = \begin{bmatrix} y_1 = 15.72266 \\ x_1 = 0 \to 2.34375 \end{bmatrix}$$

The objective function's value grows from zero: $\overline{z} = z + \Theta d_q = 2.34375 \cdot 800 = 1875$. As one can see, the algorithm has to make logical decisions in several places, namely based on reduced costs, α, and β vectors. In Chapter 4, we will see that if we make a significant numerical error in any of its calculations, the algorithm will continue the calculation in the wrong direction. This can result in cycling or an erroneous result.

Finishing the example

Fortunately, the variables are in their feasible range, so we can compute the new reduced costs:

$$\mathbf{d}^T = \begin{bmatrix} 600 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 800 \end{bmatrix} \begin{bmatrix} 1 & 0.625 \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1.25 & 0 \\ 0.5 & 1 \end{bmatrix} = \begin{bmatrix} 200 & -800 \end{bmatrix}$$

The only one improving variable is the x_1, which reduced cost is 200, so $q = 1$. The associated $\alpha_1 = \mathbf{B}^{-1}\mathbf{a}_1 = \begin{bmatrix} 1 & 0.625 \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1.25 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0.9375 \\ 0.5 \end{bmatrix}$.

In the ratio test 2 inequalities have to be solved:

$$0 \leq 15.72266 - \Theta_1 0.9375 \rightarrow \Theta_1 = 16.77083$$

$$0 \leq 2.34375 - \Theta_2 0.5 \rightarrow \underline{\Theta_2} = \underline{4.6875}$$

Notice that x_2 has an upper bound, but the corresponding α_i value is positive, so this variable moves towards its lower bound. The current $\Theta = 4.6875$, and $p = 2$. The new basic variables after the basis change:

$$\bar{\beta} = \begin{bmatrix} 11.32813 \\ 4.6875 \end{bmatrix}$$

The objective function grows again:

$$\bar{z} = 1875 + 4.6875 * 200 = 2812.5$$

The new basis, and the other data structures changes as follows:

$$\mathcal{B} = (y_1, x_1) \Rightarrow \mathbf{B} = \begin{bmatrix} 1 & 1.25 \\ 0 & 0.5 \end{bmatrix}, \mathbf{c}_{\mathcal{B}} = \begin{bmatrix} 0 \\ 600 \end{bmatrix}$$

$$\mathcal{R} = (x_2, y_2) \Rightarrow \mathbf{R} = \begin{bmatrix} 0.625 & 0 \\ 1 & 1 \end{bmatrix}, \mathbf{c}_{\mathcal{R}} = \begin{bmatrix} 800 \\ 0 \end{bmatrix}$$

The new reduced costs:

$$\mathbf{d}^T = \begin{bmatrix} 800 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 600 \end{bmatrix} \begin{bmatrix} 1 & 1.25 \\ 0 & 0.5 \end{bmatrix}^{-1} \begin{bmatrix} 0.625 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} -400 & -1200 \end{bmatrix}$$

Finally, as the reduced costs are -400 and -1200, there are no more improving variables,

an optimal solution has been found.

1.4 Product form of the inverse

During the solution algorithm the vector-matrix and matrix-vector products with the $\mathbf{B}^{-1}$ matrix are widely used:

$$\alpha = \mathbf{B}^{-1}\mathbf{a} \tag{1.21}$$

$$\pi^T = \mathbf{h}^T\mathbf{B}^{-1} \tag{1.22}$$

We can update $\mathbf{B}^{-1}$ in each iteration, and perform the multiplications, but keeping $\mathbf{B}^{-1}$ up to date is very time consuming. The so-called *Product Form of Inverse* (PFI) representation (PFI) of the basis developed by Dantzig and *Orchard-Hays* is a proper way to handle this problem [17, 61, 15]. We notice that, later on *Markowitz* published the LU decomposition method [57], but we omit that in this dissertation. The principle of PFI is that $\mathbf{B}^{-1}$ can be represented by the product of *elementary transformation matrices* (ETMs): We store the ETMs in a list, and at basis change the algorithm appends a new ETM to the end of list. An ETM can be determined quickly. When the algorithm has to compute (1.21), we use the *FTRAN* algorithm, and for (1.22) we need the *BTRAN* algorithm.

Let $\mathbf{B}$ be the original basis matrix, and $\mathbf{B}_\alpha$ is the neighbor of $\mathbf{B}$, the bases differ from each other in the pth column, because the algorithm changed the pth column of $\mathbf{B}$ by an $\mathbf{a}$ vector.

$$\mathbf{B} = \left[\mathbf{b}_1, \ldots, \mathbf{b}_{p-1}, \mathbf{b}_p, \mathbf{b}_{p+1}, \ldots, \mathbf{b}_m\right]$$

$$\Downarrow$$

$$\mathbf{B}_\alpha = \left[\mathbf{b}_1, \ldots, \mathbf{b}_{p-1}, \mathbf{a}, \mathbf{b}_{p+1}, \ldots, \mathbf{b}_m\right]$$

For constructing the new ETM, we will need the $\mathbf{v} = \mathbf{B}^{-1}\mathbf{a}$ vector, which can be computed by the ratio test. Computing the η vector from $\mathbf{v}$ is as follows:

$$\eta = \left[-\frac{v_1}{v_p}, \ldots, -\frac{v_{p-1}}{v_p}, \frac{1}{v_p} - \frac{v_{p+1}}{v_p}, \ldots, -\frac{v_m}{v_p}\right]$$

The $\mathbf{E}$ ETM is the following:

$$\mathbf{E} = \begin{bmatrix} 1 & & \eta^1 & & \\ & \ddots & \vdots & & \\ & & \eta^p & & \\ & & \vdots & \ddots & \\ & & \eta^m & & 1 \end{bmatrix}$$

By using $\mathbf{E}$ matrix, $\mathbf{B}_\alpha^{-1}$ can be computed with the following formula:

$$\mathbf{B}_\alpha^{-1} = \mathbf{E}\mathbf{B}^{-1}$$

After the ith basis change the basis inverse is the following:

$$\mathbf{B}_i^{-1} = \mathbf{E}_i\mathbf{E}_{i-1}\ldots\mathbf{E}_1,$$

where

$$\mathbf{B}_0^{-1} = \mathbf{I}$$

For FTRAN and BTRAN it is sufficient to store the η vectors and the corresponding p indices from ETMs.

FTRAN

The FTRAN (Forward Transformation) operation implements the (1.21) formula. Suppose that, the actual basis is represented by product of s ETMs. We know that

$$\alpha = \mathbf{B}^{-1}\mathbf{a} = \mathbf{E}_s\mathbf{E}_{s-1}\ldots\mathbf{E}_1\mathbf{a} \tag{1.23}$$

$$\alpha_0 = \mathbf{a}$$

$$\alpha_i = \mathbf{E}_i\alpha_{i-1}, i = 1,\ldots,s$$

$$\alpha_s = \alpha$$

The $\alpha = \mathbf{E}\mathbf{a}$ can be performed by the following:

$$\alpha = \mathbf{E}\mathbf{a} = \begin{bmatrix} 1 & & \eta^1 & & \\ & \ddots & \vdots & & \\ & & \eta^p & & \\ & & \vdots & \ddots & \\ & & \eta^m & & 1 \end{bmatrix}\begin{bmatrix} a_1 \\ \vdots \\ a_p \\ \vdots \\ a_m \end{bmatrix} = \begin{bmatrix} a_1 + \eta^1 a_p \\ \vdots \\ \eta^p a_p \\ \vdots \\ a_m + \eta^m a_p \end{bmatrix}$$

Notice that, when $a_p = 0$, then the we can omit the related ETM from (1.23).

BTRAN

We use the BTRAN (Backward Transformation) for the (1.22) formula. The inverse of the basis is represented by s ETMs. We know that

$$\alpha^T = \mathbf{a}^T \mathbf{B}^{-1} = \mathbf{a}^T \mathbf{E}_s \mathbf{E}_{s-1} \ldots \mathbf{E}_1 \tag{1.24}$$

$$\alpha_0^T = \mathbf{a}^T$$

$$\alpha_i^T = \alpha_{i-1}^T \mathbf{E}_{s-i+1}, i = 1, \ldots, s$$

$$\alpha_s^T = \alpha^T$$

The $\alpha^T = \mathbf{a}^T \mathbf{E}$ can be performed as follows:

$$\alpha^T = \mathbf{a}^T \mathbf{E} = \left[\begin{array}{ccc} a_1, \ldots, a_p, \ldots, a_m \end{array} \right] \begin{bmatrix} 1 & & \eta^1 & & \\ & \ddots & \vdots & & \\ & & \eta^p & & \\ & & \vdots & \ddots & \\ & & \eta^m & & 1 \end{bmatrix}$$

$$\alpha^T = \mathbf{a}^T \mathbf{E} = \left[\begin{array}{ccc} a_1, \ldots, \sum_{i=1}^{m} a_i \eta^i, \ldots, a_m \end{array} \right]$$

1.5   Computation of the reduced costs

In this section we describe how to compute the reduced costs in primal phase-1 and phase-2 efficiently.

1.5.1   Reduced costs in phase-1

According to (1.15), the definition of reduced costs in phase-1 is the following:

$$d_j = \sum_{i \in \mathcal{M}} \alpha_j^i - \sum_{i \in \mathcal{P}} \alpha_j^i$$

It can be seen that in order to compute reduced cost d_j we need the vector $\alpha_j = \mathbf{B}^{-1} \mathbf{a}_j$. However, we have to compute more reduced costs, so it is not practical to compute all α_j vectors directly, because the algorithm will be very slow. We can solve this problem with

introducing an auxiliary vector $\mathbf{h} \in \{-1, 0, 1\}^m$, which has the following components:

$$
h_i = \begin{cases} 1, & \text{if } i \in \mathcal{M} \\ -1, & \text{if } i \in \mathcal{P} \\ 0, & \text{otherwise} \end{cases}
$$

The reduced cost of x_j becomes:

$$
d_j = \mathbf{h}^T \boldsymbol{\alpha}_j = \mathbf{h}^T \mathbf{B}^{-1} \mathbf{a}_j
$$

Note that $\mathbf{h}^T \mathbf{B}^{-1}$ is the same for each nonbasic variable, thus we can introduce the *phase-1 simplex multiplier*:

$$
\boldsymbol{\phi}^T = \mathbf{h}^T \mathbf{B}^{-1} \tag{1.25}
$$

$$
d_j = \boldsymbol{\phi}^T \mathbf{a}_j
$$

First, we have to compute the simplex multiplier with a BTRAN operation. In the second step, the dot products are used to obtain the reduced costs, so the computation will be faster than computing $\boldsymbol{\alpha}_j$ vectors first.

To improve the speed of computation, it is recommended to use this approach:

$$
\mathbf{d}_{\mathcal{R}}^T = \boldsymbol{\phi}^T \mathbf{R} \tag{1.26}
$$

If we store $\mathbf{R}$ rowwise, we can obtain reduced costs by multiplying the rows of $\mathbf{R}$ with the corresponding entries of $\boldsymbol{\phi}$ and summarize the multiplied rows. If $\boldsymbol{\phi}$ has only a few nonzero components then this is a very fast operation.

1.5.2   Reduced costs in phase-2

As saw in (1.16), the definition of reduced costs in phase-2 is the following:

$$
d_j = c_j - \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \mathbf{a}_j
$$

We can introduce the *phase-2 simplex multiplier*:

$$
\boldsymbol{\pi}^T = \mathbf{c}_{\mathcal{B}}^T \mathbf{B}^{-1} \tag{1.27}
$$

$$
d_j = c_j - \boldsymbol{\pi}^T \mathbf{a}_j \tag{1.28}
$$

This simplex multiplier has the same advantages like the phase-1 simplex multiplier Before we compute the reduced costs, we perform a BTRAN operation to obtain $\boldsymbol{\pi}^T$. After
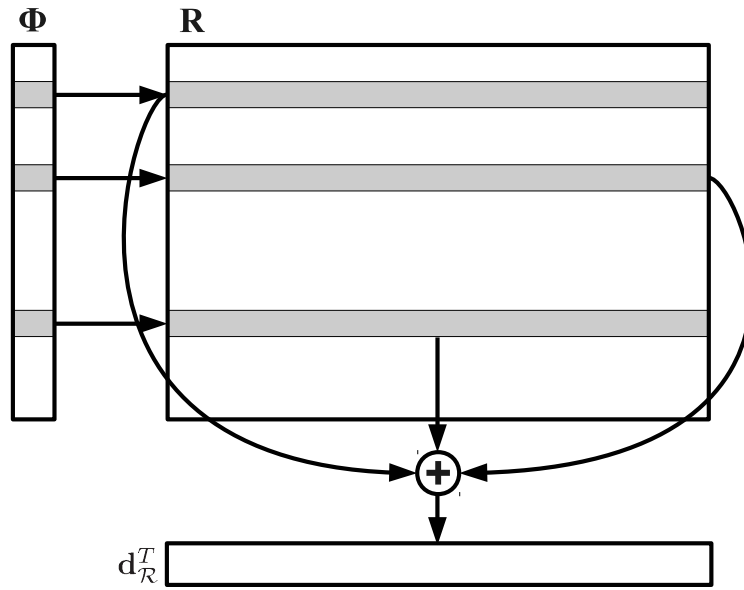
Figure 1.6: Computing phase-1 reduced cost vector using simplex multiplier rowwise matrix representation

this we obtain the reduced costs using dot products and one subtraction. It is known that when we replace the pth basis column with another one, we can update the simplex multiplier ($\bar{\pi}$) using the following method [99]:

$$\bar{\rho}^p = \mathbf{e}_p^T \bar{\mathbf{B}}^{-1}$$

$$\bar{\pi}^T = \pi^T + d_q \bar{\rho}^p, \tag{1.29}$$

where d_q is the reduced cost of the previous incoming variable. The vector $\mathbf{e}_p$ contains only one nonzero component, so using an enhanced BTRAN algorithm for computing $\bar{\rho}^p$ can accelerate this update. We can update the reduced costs using the following method:

$$\bar{d}_j = d_j - d_q \bar{\rho}^p \mathbf{a}_j$$

Usually $\bar{\rho}^p$ contains fewer nonzero components than π^T, so it is practical to compute reduced costs using this formula:

$$\bar{\mathbf{d}}_{\mathcal{R}}^T = \mathbf{d}_{\mathcal{R}}^T - d_q \bar{\rho}^p \mathbf{R} \tag{1.30}$$

Similarly to the phase-1, if $\mathbf{R}$ is stored rowwise, computing reduced costs can be a fast operation using vector additions and multiplication. In the beginning of the third chapter,

we describe the details of the update of phase-1 reduced costs.

1.6 The dual simplex method

In this section, we briefly describe the second phase *dual simplex* algorithm. It can be used when there is at least one infeasible basic variable, but the optimality conditions meet. We notice that the literature describes the theory of duality, but it is unnecessary for this discussion.

Algorithmically, this method differs from primal phase-2 in the choosing of incoming and outgoing variables. While the primal selects an incoming variable first, the dual pricing chooses an outgoing basic variable and in the dual ratio test, we select the incoming variable. Let us consider the following LO problem:

$$\min z = \quad \mathbf{c}^{\mathsf{T}}\mathbf{x} \tag{1.31}$$

$$s.t. \quad \mathbf{Ax} = \mathbf{b} \tag{1.32}$$

$$\mathbf{0} \leq \mathbf{x} \tag{1.33}$$

As we know, $\mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1}\mathbf{b}$. If this vector is feasible, that is $\mathbf{x}_{\mathcal{B}} \geq \mathbf{0}$, the solution is optimal, and the algorithm terminates. Otherwise, we select an $x_p < 0$ from $\mathbf{x}_{\mathcal{B}}$, p is the index of the pivot row. In the second step, we have to select the incoming variable carefully; We have to keep the optimality of the current solution, namely $\mathbf{d} \geq \mathbf{0}$. Recall the formula (1.30), and let denote α^p the row p of the transformed matrix:

$$\alpha^p = \rho^p \mathbf{R}$$

We have to choose an incoming column such a way that the corresponding d_q reduced cost has to reach the 0 value ; the incoming variable will be basic, and the reduced costs of basic variables are zero. So the incoming q index is determined by the following formula:

$$\theta = \frac{d_q}{\alpha_q^p} = \max\left\{\frac{d_j}{\alpha_j^p} : \alpha_j^p < 0, j \in \mathcal{R}\right\}$$

In the basis change, the new reduced cost on the incoming index is $-\theta$.

1.7 Pannon Optimizer

The simplex method described in the dissertation and the various innovations and accelerations were implemented in the software called Pannon Optimizer. It is an open source software written in C ++, with a modular structure, developed by colleagues from the Operations Research Laboratory of the University of Pannonia. The primary purpose of

the software is to make it easy to implement and test a variety of research ideas. The software is based on the book by Professor István Maros [61].

The main component of Pannon Optimizer is a library that can be linked into software for various purposes, such as a command line program. The upper layer of the library is the Solver framework, which controls the steps and strategies of the solving process. This includes the heuristic error detection procedure described in Chapter 4. It includes a modeling subsystem that handles the matrix, vectors, and other data needed for the algorithm, as well as some special algorithms used only in the simplex method. The Chapter 3 accelerated, row-wise BTRAN algorithm is part of this module. These are served by the Linear algebraic kernel, which contains various linear algebraic operations. The accelerated procedures described in Chapter 5 are included. In addition to the Linear algebraic kernel, there is a collection of other procedures in the Utilities module. The lower layers of the Pannon Optimizer are connected to GLIBC, the operating system, and some of its components are directly connected to the hardware. The structure of the system is shown in the Figure 1.7.

Pannon Optimizer is currently able to solve NETLIB and other model collections at an acceptable speed and is currently under development. The current stable version is available at:

```
https://sourceforge.net/projects/pannonoptimizer/
```
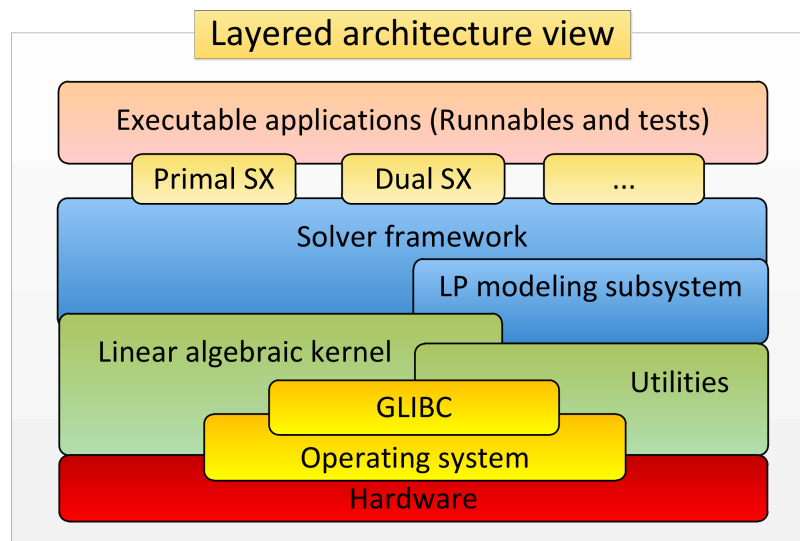

Figure 1.7: Pannon Optimizer architecture

Chapter 2

Floating-point numbers

In this chapter, the definitions and the essential properties of the *floating-point* numbers will be discussed. Most of the simplex implementations, including the topic of this dissertation, use this number system. One of the keys to implementing a stable software is using floating-point arithmetic appropriately.

2.1 A brief preview

There is a long list of representing real numbers with different advantages and disadvantages. The simplest systems are the *fixed-point* numbers, where a fixed number of bits are used for the integer and fractional part. The main advantage is that it is straightforward to implement basic mathematical operations using integer numbers. The fixed-point numbers can be very efficient in low-performance environments [87], where hardware support for floating-point numbers is not available. They were also very useful tools even the videogame industry, for instance, John Carmack used them in his revolutionary graphics engines of Wolfenstein 3D [80] and Doom [79]. However, fixed-point numbers have their limitations, namely the range of the representable numbers is fixed and narrow. There are a lot of interesting ideas to represent numbers, such as logarithmic [46] [39] and semi-logarithmic [68] systems, which can be used in digital signal processing to improve the dynamic range. Some applications can also use the rational numbers well [49]. The list is long, we can mention continued fractions [50] [105], possibly infinite strings of rational numbers [66], level-index numbers [9] [71], fixed-slash and floating-slash numbers [65], and 2-adic numbers [106].

Strictly speaking, the floating-point numbers can be represented in the following way:

$$(-1)^s \times m_0.m_1m_2\ldots m_{p-1} \times \beta^e \tag{2.1}$$

Where the meaning of the letters is as follows:

- $s \in \{0, 1\}$: The *sign* of the number.

- p: The *precision* of the number, that is the number of used digits, and $p \geq 2$.

- $m_i, 0 \leq i \leq p - 1$: The digits of the number, where $m_i < \beta$.

- β: The *radix* (or *base*) of the floating-point number, where $\beta \geq 2$

- e: The *exponent* of the number.

Throughout history, different radix values were used, like the radix-60 number system at the Babylonians [48]. The $\beta = 10$ value is surprisingly widely used, not only in daily human calculations, but it is used in financial calculations and in pocket calculators also. In 2001, 55% of numerical data was stored on commercial databases in decimal numbers [7]. Nowadays this is a very actively researched area [11, 12, 13, 22, 2, 103, 102, 104, 56, 108].

The first described electro-mechanical implementation of Babbage's Analytical Engine was done by Leonardo Torres y Quevedo in 1914 [76]. Later, Konrad Zuse built the Z3 computer in 1941, which used a radix-2 floating-point number system [8].

In 1985, the IEEE 754-1985 Standard of Binary Floating-Point Arithmetic was released [35], which was extended in 1987 (IEEE 854-1987 Standard Radix Independent Floating-Point Arithmetic) [36] for radix-2 and radix-10 number systems. In the current dissertation, we focus only on the radix-2 number systems.

2.2 The floating-point format

The floating-point format design aimed at creating a widely available and efficient number system. Designers have realized that engineering and other applications require a specific precision only; for example, we don't need to know the Earth-Sun distance in millimeters, but we want to describe the weight of an atom. There are five important desirable properties of this number-system in an efficient simplex method implementation:

- Speed: The simplex method has to perform mathematical operations very quickly to produce the result as soon as possible.

- Accuracy: The wrong result of an operation can lead the simplex method in the wrong direction.

- Range: The simplex method has to handle large and small numbers as well.

- Portability or platform-independency: The simplex method has to work on different computer architectures, so the current hardware has to support the floating-point numbers.

- Easy to use and implement: The number system has to be as simple as possible because it would be more challenging to design a stable simplex method implementation.

Representation	s	M	m	β	p	e	Remark
$123625 \cdot 10^{-3}$	0	123625	-	10	6	-3	Decimal
$1.23625 \cdot 10^2$	0	-	1.23625	10	6	2	Decimal Normalized
$1111011101_{(2)} \cdot 2^{-3}$	0	$1111011101_{(2)}$	-	2	10	-3	Binary
$1.111011101_{(2)} \cdot 2^6$	0	-	$1.111011101_{(2)}$	2	10	6	Binary Normalized

Table 2.1: Different representations of the number $x = 123.625$.

Now, we complete the definition of floating-point numbers showed in (2.1):

$$x = (-1)^s \times M \times \beta^{e-p+1} \tag{2.2}$$

Here M is an integer number, which is less than or equal to $\beta^p - 1$. This value is called the *integral significand* of the x. The digits of M are the *significant digits*. We have to note that, the significand is often called *mantissa* in literature. The mantissa is the fractional part of the logarithm of a number, and as Goldberg showed his great work [27], the term significand was created by Forsythe and Moler in 1967 [26].

Moreover, the e exponent has bounds like $e_{\min} \leq e \leq e_{\max}$.

The other representation of a floating-point number is the following:

$$x = (-1)^s \cdot m \cdot \beta^e \tag{2.3}$$

The notation of m is the new term of this definition, where

$$m = |M| \cdot \beta^{1-p}, \tag{2.4}$$

that is, m is the *normal significand*, or *significand* of the representation of x. The consequence of this definition is that there is only one digit before the radix point, and $p - 1$ digit after, and $0 \leq m < \beta$.

For example, the number of $x = 123.625$ has many different possible representations, as Table (2.1) shows.

The *infinitely precise significand* of x, with the radix β is the following:

$$\frac{x}{\beta^{\lfloor \log_\beta |x| \rfloor}} \tag{2.5}$$

As the examples in Table (2.1) show, there are several representations of a floating-point number. However, it is advised to use unique representations. If we have a representation, where $e \geq e_{\min}$, it is called a *normalized representation*. As we will see later, this has an additional pleasing property in radix-2 systems.

There are two variants, if we talk about normalized floating-point numbers: There are *normal* and *subnormal* or *denormal* numbers.

- Normal numbers: In this variant $1 \leq |m| < \beta$, or otherwise $\beta^{p-1} \leq |M| < \beta^p$. The radix point is after the first digit of the significand, which is 1 in radix-2. Notice that if x is a normalized floating point number, its significand is equal to its infinitely precise significand.

- Subnormal numbers: In this variant $e = e_{\min}$, and $|m| < 1$, or $|M| \leq \beta^{p-1} - 1$. The leading digit of the significand is always zero in radix-2.

The so-called *hidden bit* (or *leading bit*, *implicit bit*) convention says that we do not store the first bit in the case of radix-2. This method can improve the accuracy if there is a limited number of bits to represent the significand. Thus, the normal numbers can be written as

$$1.m_1 m_2 \ldots m_{p-1},$$

while subnormal number:

$$0.m_1 m_2 \ldots m_{p-1},$$

The digits in $.m_1 m_2 \ldots m_{p-1}$ consist of the *trailing significand*, or the *fraction*. After this we can define some additional terms:

- $\beta^{e_{\min}}$: The smallest positive normal number.

- $\Omega = (\beta - \beta^{1-p}) \cdot \beta^{e_{\max}}$: The largest finite floating-point number.

- $\alpha = \beta^{e_{\min}-p+1}$: The smallest positive subnormal number, and of course, the smallest positive floating-point number.

The set of the normal range are built by the numbers whose absolute value is between $\beta^{e_{min}}$ and Ω. The set of the subnormal range consists of the numbers whose magnitude is less than $\beta^{e_{min}}$. A simple example to demonstrate these values is the $\beta = 2, p = 6, e_{\min} = -10, e_{\max} = 10$ system:

- The most significant bit of the smallest positive number is 1, and the fraction part is zero: $1.00000_{(2)}$. The radix point is shifted by $e_{\min}$, so the number is: $0.0000000001_{(2)} = 2^{-10} = 9.765625 \cdot 10^{-4}$.

- Every digit of the largest finite number is 1: $1.11111_{(2)}$, so the largest number is: $1.11111_{(2)} \cdot 2^{10} = 2016$.

- The least significant bit of the smallest subnormal positive number is 1, and the leading bits are 0: $0.00001_{(2)}$, so the smallest subnormal positive number is: $0.00001_{(2)} \cdot 2^{-10} = 3.0517578125 \cdot 10^{-5}$.

2.2.1 Rounding

Because of the significand's limited digits, not every number is representable in a floating-point format. The Algorithm (2.1) converts a decimal fractional number to binary, and it prints the series of digits to the output.

Algorithm 2.1 Convert decimal fractional number to binary

Input: x in decimal, where $0 \leq x < 1$

1: **while** $x > 0$ **do**
2:     $x := x \times 2$
3:     **if** $x \geq 1$ **then**
4:             **print** "1"
5:             $x := x - 1$
6:     **else**
7:             **print** "0"
8:     **end**
9: **end**

Let's convert the 0.1_{10} to binary. Figure (2.1) shows the progress and the output of the algorithm. It is clear that after the 6th iteration, the algorithm gets stuck in an infinite loop. It means that we cannot store the exact value of 0.1_{10} in a 2-radix floating-point number, we have to use its approximation. But there are numbers in which binary representation consists of a finite number of digits, but the used floating-point format has fewer digits. In both cases, a rounded number is stored.

Iteration	x	Output
1	0.1	0
2	0.2	0
3	0.4	0
4	0.8	0
5	$1.6 \rightarrow 0.6$	1
6	$1.2 \rightarrow 0.2$	1

Figure 2.1: Convert 0.1_{10} to binary.

There are four types of *rounding* strategies specified by the IEEE 754 standard:

- RD: Round toward $-\infty$ (or round downward). The stored number is the greatest floating-point number, which is less than the original value.

- RU: Round toward $+\infty$ (or round upward). The stored number is the lowest floating-point number, which is greater than the original value.

- RZ: Round toward zero. If $x < 0$, then round toward $+\infty$, and if $x > 0$, round toward $-\infty$.

- RN: Round to nearest. The result is the closest representable floating-point number to x. If x is exactly halfway between two floating-point numbers, then RN(x) is the
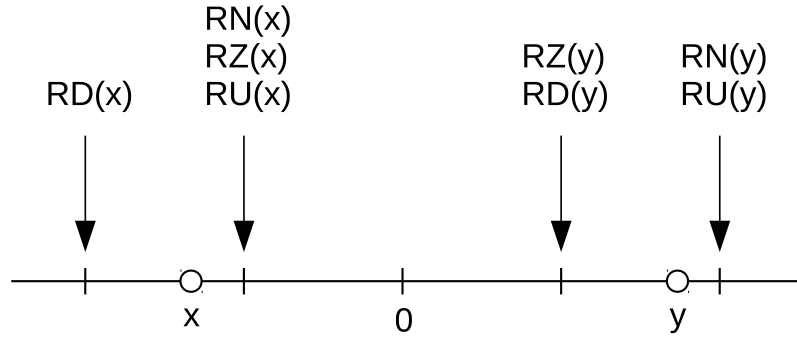
Figure 2.2: Different rounding modes.

number which integral significand is even. This is the so-called round to nearest even strategy.

If the rounding mode is unspecified, the rounded value of x is denoted by $\circ(x)$. In this case, the relative error of rounding:

$$\epsilon(x) = \left| \frac{x - \circ(x)}{x} \right|$$

The relative error of normal numbers in RN mode is less than or equal to $\frac{1}{2}\beta^{1-p}$, and in other rounding modes it is less than β^{1-p}. In subnormal case, the relative error can be close to 1. It is clear that the round-to-nearest strategy can result in less errors than the directed strategies. In the this dissertation we will use the round-to-nearest strategy.

The rounding can create additional numerical problems. The two basic mathematical operations (addition and multiplication) have the following widely known properties:

- Commutativity: $a + b = b + a, a \times b = b \times a, \forall a, b$.

- Associativity: $a + (b + c) = (a + b) + c, a \times (b \times c) = (a \times b) \times c, \forall a, b, c$.

- Distributivity: $a \times (b + c) = a \times b + a \times c, \forall a, b, c$.

The commutativity property holds on to the floating-point arithmetic, but we lose the other two properties. Sometimes $\circ(\circ(a + b) + c)$ drastically different from $\circ(a + \circ(b + c))$, as Figure (2.3) shows by code lines 10 and 11. If there is no overflow or underflow, the multiplications have much lower error rate. Namely:

$$\left(\frac{1 - \frac{1}{2}\beta^{1-p}}{1 + \frac{1}{2}\beta^{1-p}} \right)^2 \leq \frac{\circ(\circ(a \times b) \times c)}{\circ(a \times \circ(b \times c))} \leq \left(\frac{1 + \frac{1}{2}\beta^{1-p}}{1 - \frac{1}{2}\beta^{1-p}} \right)^2$$

In radix-2 systems, if $p = 24$, then the lower and upper bounds are ≈ 0.99999976 and 1.000000238, if $p = 54$, they are 0.999999999999999778 and 1.000000000000000222. Moreover, the distributivity is also problematic as the addition operations, as Figure (2.3) shows. It is clear that if the order of addition operations is not carefully established, the results can be completely wrong.

```cpp
1  #include <iostream>
2
3  int main() {
4      float a = 123322443;
5      float b = -123322453;
6      float c = 10;
7      float d = (a + b) + c;
8      float e = a + (b + c);
9      std::cout.precision(10);
10     std::cout << "d = " << d << std::endl;
11     std::cout << "e = " << e << std::endl;
12     c = 1212442.04553;
13     d = (a * b) * c;
14     e = a * (b * c);
15     std::cout << "d = " << d << std::endl;
16     std::cout << "e = " << e << std::endl;
17     std::cout << "diff: " << (d-e) << std::endl;
18     e = (a + b) * c;
19     d = a * c + b * c;
20     std::cout << "d = " << d << std::endl;
21     std::cout << "e = " << e << std::endl;
22     std::cout << "diff: " << (d - e) << std::endl;
23     return 0;
24 }
```

The output:

```
d = -6
e = -8
d = -1.843933565e+22
e = -1.84393334e+22
diff: -2.251799814e+15
d = -16777216
e = -19399072
diff: 2621856
```

Figure 2.3: A short demo code in C++ to demonstrate the lack of associativity and distributivity. Compiled with gcc version 9.1.0.

Cancellation

It may happen that an arithmetic operation does not cause a new error. But if the operands have an error, we can amplify that. Sterbenz [92] proved that in a radix-β floating-point number system, where there are subnormal numbers, and if

$$\frac{y}{2} \leq x \leq 2y$$

for all x and y floating-point numbers, $x - y$ is exactly representable. Figure (2.4) shows a C++ code example which demonstrates the *cancellation*. In the code, the a2 and b variables are close enough to each other, so because of Sterbenz's lemma, d is the exact difference of a2 and b. This can be seen in the output: The digits of d after the 0.5 part are the same as the fractional digits of a2 (the last digit of a2 is rounded during the printing). However, as saw earlier, we cannot store the value of 0.1 without a rounding error; the error of c is around 1.490116×10^{-8}. However, the value of a2 = RN(a + c) is different from a + c, the relative error of a2 is around 2.44×10^{-9}. This is still not a big error, but if we subtract b from a2 (which is an exact operation due to the Sterbenz lemma), the relative error of d increases: The relative error is 4.073×10^{-5}.

```cpp
#include <iostream>

int main() {
    float a = 1000;
    float b = 999.5;
    float c = 0.1;
    float a2 = a + c;
    float d = a2 - b;
    std::cout.precision(15);
    std::cout << "c = " << c << std::endl;
    std::cout << "a2 = " << a2 << std::endl;
    std::cout << "d = " << d << std::endl;
    std::cout << "relative error = "
              << ((0.6f - d) / 0.6) << std::endl;
    return 0;
}
```

The output:

```
c = 0.100000001490116
a2 = 1000.09997558594
d = 0.5999755859375
relative error = 4.07298405965169e-05
```

Figure 2.4: A short demo code in C++ to demonstrate the cancellation error. Compiled with gcc version 9.1.0.

Name	p	Exponent bits	Max finite	Min normal	Min subnormal
Half	11	5	65504	6.1035×10^{-5}	5.9605×10^{-8}
Single	24	8	$\approx 3.4028 \times 10^{38}$	$\approx 1.1755 \times 10^{-38}$	$\approx 1.4013 \times 10^{-45}$
Double	53	11	$\approx 1.7977 \times 10^{308}$	$\approx 2.225 \times 10^{-308}$	$\approx 4.9407 \times 10^{-324}$
Double extended	64	15	$\approx 1.1897 \times 10^{4932}$	$\approx 3.3621 \times 10^{-4932}$	$\approx 3.6452 \times 10^{-4951}$
Quadruple	113	15	$\approx 1.1897 \times 10^{4932}$	$\approx 3.3621 \times 10^{-4932}$	$\approx 3.6452 \times 10^{-4951}$

Table 2.2: The main IEEE 754-2008 radix-2 floating-point formats.

Let's see how the float type stores 0.6 and the value of d:

- 0.6: $\underbrace{0}_{\text{signal}}$ $\underbrace{01111110}_{\text{exponent}}$ $\underbrace{0011001100110011010}_{\text{significand}}$

- d: $\underbrace{0}_{\text{signal}}$ $\underbrace{01111110}_{\text{exponent}}$ $\underbrace{0011001100100000000000}_{\text{significand}}$

As we can see, the cancellation error "cancelled" the last 9 bits of d. Sometimes this error is also called *catastrophic cancellation*.

2.3 The IEEE 754 Standard

In this section, a small part of the IEEE 754 Standard is introduced. We focus on the number formats used in this dissertation, see Table (2.2). Notice that this is the newest revision of the standard, named IEEE 754-2008. The standard also defines decimal formats, but they are irrelevant for us; there are optional formats (for example single extended) and one that is currently not supported by any hardware (256-bit format). Every format in Table (2.2) except for the double extended type uses the hidden leading bit convention. The 32 bit single and 64 bit double formats are supported by most of the hardware. We use half format later in this dissertation for demonstration purposes.

The 80-bit format is mostly used by the Intel's FPU register stack. The FPU has 8 80-bit wide registers, each register can store 1 floating-point value. Of course, this hardware can store 32 and 64-bit formats also, this is controlled by a special control register. Software developers typically use this feature in order to perform temporary calculations inside the FPU with higher precision.

Special values

The IEEE 754-2008 standard specifies some special values. These are the zero, infinity the and not-a-number symbols. Now let $n = (-1)^s \times m \times \beta^e$ denote the floating-point number that we would like to store. The exponents of the finite numbers are biased: The real exponent e of n is incremented by a fixed value such a way that if the most significand bit can show that $|n| > \beta$ or not. Let E the stored, biased version of e, and M the integer number created by the bits of significand. However, if the E consists of only 1's, n is an infinity or not-a-number symbol. So there are the following cases:

- $E > 0$ and E has at least one 0 bit: n is a normal number.

- $E = 0$: In this case the number can be a zero, or a subnormal number. If $M = 0$, then $|n| = 0$. Notice that, the s bit can be 1 or 0, so we can represent the -0 symbol also. If $M > 0$, then n is a subnormal number, and $e = E + e_{\min}$.

- E contains only 1's: This is not a finite number, the symbol depends on the M:

– $M = 0$: The n is $-/+\infty$, depending on s.

– $M > 0$: The n is a not-a-number (NaN) symbol. This symbol can represent uninitialized numbers, or the result of an invalid operation (for example $\sqrt{-1}$).

Some examples with 16-bit:

- $\underbrace{0}$ $\qquad$ $\underbrace{10010}$ $\qquad\qquad$ $\underbrace{1000100110}$

  sign: positive   exponent: 18 - $\underbrace{15}$ = 3 significand: $\underbrace{1024}$ + 550 = 1574
   bias hidden bit

  $= \frac{1574}{2^7} = 12.296875$

- $\underbrace{0}$ $\qquad\qquad$ $\underbrace{00000}$ $\qquad\qquad\qquad$ $\underbrace{0010101000}$

  sign: positive exponent: 0 - $\underbrace{15}$ = -15, subnormal significand: 0.1640625 (hidden bit = 0)
   bias

  $= 0.1640625 \times 2^{-14} = 1.0013580322265625 \times 10^{-5}$

- $\underbrace{1}$ $\qquad$ $\underbrace{11111}$ $\quad$ $\underbrace{0000000000}$ = $-\infty$

  sign: negative special symbol significand = 0

- $\underbrace{0}$ $\qquad$ $\underbrace{11111}$ $\quad$ $\underbrace{1000000000}$ = NaN

  sign: positive special symbol significand > 0

Double rounding

The CPU rounds every number which is not a p-length floating-point number. For example, let us consider this bit pattern, divided to sections:

$$\underbrace{1.101000}_{\text{Section A}} \quad \underbrace{100}_{\text{Section B}} \quad 0 \quad \underbrace{00100}_{\text{Section C}}$$

The Section A and B are totally 10-bit length, then there is a 0, and a third section, where is at least one 1. Now image what happens, if we would like to round this number to 10 digits. We have to look at the 11th bit, that is 0, so the number is rounded down, the result is:

$$\underbrace{1.101000}_{\text{Section A}} \quad \underbrace{100}_{\text{Section B}}$$

In the next step, round this number to 6 digits: The next, 7th bit is 1, and this bit is followed by only zeros. It means that this number is exactly at the halfway of the two possible results, so the round to nearest even strategy is applied: We have to round to the number that has 0 at the 6th bit, so the result is the following:

$$1.101000$$

Now go back to the original number, and round it to 6 digits. The next, 7th digit is 1, and there is at least additional 1 after this bit, so we round up, the result is:

$$1.101001$$

We have got 2 different results. Although this is an example with short bit patterns, this can happen in real life software also. The traditional 32 bit Intel CPU's used the FPU to perform floating-point calculations. As we mentioned earlier, this FPU has 80 bit wide registers, which can store 32, 64 and 80-bit floating-point numbers. On default settings, on the 32-bit architectures the compiled code uses these registers: The program loads the 64-bit double type variables into the FPU and converts them to an 80-bit format. The calculations are performed with this, more precise format. In this code (see lines 9 and 14) additions and multiplications are executed. The CPU rounds the results of each elementary operations, and stores this rounded 80-bit wide value in the FPU's registers. When the 9th line is executed, or the for loop is finished, the CPU stores the result back to the variable c, but this is a 64-bit variable; so a new rounding is necessary. Of course, both compiled programs are 64-bit binaries, but in the first case, the compiler option -mfpmath=387 forces the compiler to use the FPU.

Let us investigate the code of Figure (2.5). This C++ code is compiled in two ways: In both cases, we build an X86 64-bit Linux executable binary file, so the executables work on 64-bit mode. The first version is compiled with the -mfpmath=387 option, the second is not. As it can be seen, the output of the two versions is different.

Without that option, the compiler does the compiling without the FPU because the 64-bit Intel architectures have the new SIMD registers which are more flexible than the old-fashioned FPU. In these new registers, we can store 32 or 64-bit floating-point numbers only. So if the code uses these registers, the temporary results of the calculation are always rounded to 64-bit, and there is no more additional rounding at the end.

```cpp
1  #include <iostream>
2  #include <cstdlib>
3
4  int main() {
5      srand(0);
6      std::cout.precision(23);
7      double a = 9223372036854775808.0;
8      double b = 1024.25;
9      double c = a + b;
10     std::cout << c << std::endl;
11     for (int i = 0; i < 10000; i++) {
12         a = rand() % 10000000000;
13         b = rand() % 10000000000;
14         c += a * b;
15     }
16     std::cout << c << std::endl;
17     return 0;
18 }
```

The output, with compile option -mfpmath=387:

9223372036854775808
11360200579421323657216

The output, with default options (sse, 64 bit):

9223372036854777856
11360200579421325754368

Figure 2.5: A short demo code in C++ to demonstrate the double rounding. Compiled with gcc version 9.1.0.

Chapter 3

Improvement techniques of pricing

3.1 Updating the phase-1 simplex multiplier

As we saw in the previous chapter, the phase-2 simplex multiplier update formula (1.29) has been known since 1974 [99]. We will show how one can update the phase-1 simplex multiplier below. As we saw in (1.25) this simplex multiplier can be obtained by multiplying the vector $\mathbf{h}$ and the actual $\mathbf{B}^{-1}$ basis inverse:

$$\boldsymbol{\phi}^T = \mathbf{h}^T \mathbf{B}^{-1}$$

Now we move from basis $\mathbf{B}$ to $\bar{\mathbf{B}}$, and the vector $\mathbf{h}$ changes to $\bar{\mathbf{h}}$. Let:

$$\Delta \mathbf{h} = \bar{\mathbf{h}} - \mathbf{h}. \tag{3.1}$$

We suppose that $\Delta \mathbf{h}$ contains much less nonzero components than $\bar{\mathbf{h}}$, so we expect that with $\Delta \mathbf{h}$ we can obtain the new simplex multiplier faster.

Determine the value of $\bar{\boldsymbol{\phi}}^T$ using $\boldsymbol{\phi}^T$, $\Delta \mathbf{h}$ and $\bar{\mathbf{B}}^{-1}$:

$$\bar{\boldsymbol{\phi}}^T = \bar{\mathbf{h}}^T \bar{\mathbf{B}}^{-1} = (\mathbf{h} + \Delta \mathbf{h})^T \bar{\mathbf{B}}^{-1} = \mathbf{h}^T \bar{\mathbf{B}}^{-1} + \Delta \mathbf{h}^T \bar{\mathbf{B}}^{-1} \tag{3.2}$$

We can compute the vector-matrix product $\Delta \mathbf{h}^T \bar{\mathbf{B}}^{-1}$ quickly, because $\Delta \mathbf{h}$ has few nonzero components. However, our expectation is that $\bar{\mathbf{h}}$ does not contain more nonzero components than $\mathbf{h}$, so computing $\mathbf{h}^T \bar{\mathbf{B}}^{-1}$ can be slower than the operation $\bar{\boldsymbol{\phi}}^T = \bar{\mathbf{h}}^T \bar{\mathbf{B}}^{-1}$. Consequently, we need an other way to compute $\mathbf{h}^T \bar{\mathbf{B}}^{-1}$.

If we rearrange (1.25), we obtain that

$$\boldsymbol{\phi}^T \mathbf{B} = \mathbf{h}^T$$

Let be $\hat{\boldsymbol{\phi}}$ the vector, where $\mathbf{h}^T \bar{\mathbf{B}}^{-1} = (\boldsymbol{\phi} + \hat{\boldsymbol{\phi}})^T$ equality holds. Rearrange this formula for

$\mathbf{h}^T$ and obtain $\hat{\phi}$:

$$(\phi + \hat{\phi})^T \bar{\mathbf{B}} = \mathbf{h}^T,$$

$$(\phi + \hat{\phi})^T \bar{\mathbf{B}} = \phi^T \mathbf{B}$$

$$\phi^T \bar{\mathbf{B}} + \hat{\phi}^T \bar{\mathbf{B}} = \phi^T \mathbf{B}$$

$$\hat{\phi}^T \bar{\mathbf{B}} = \phi^T \mathbf{B} - \phi^T \bar{\mathbf{B}}$$

$$\hat{\phi}^T \bar{\mathbf{B}} = \phi^T (\mathbf{B} - \bar{\mathbf{B}})$$

$$\hat{\phi}^T = \phi^T (\mathbf{B} - \bar{\mathbf{B}}) \bar{\mathbf{B}}^{-1} \tag{3.3}$$

Finally, the updated simplex multiplier, if $\mathbf{h}$ changes by $\Delta \mathbf{h}$, based on (3.2), (3.1), and (3.3):

$$\bar{\phi}^T = \phi^T + (\phi^T (\mathbf{B} - \bar{\mathbf{B}}) + \Delta \mathbf{h}^T) \bar{\mathbf{B}}^{-1}. \tag{3.4}$$

The matrices $\mathbf{B}$ and $\bar{\mathbf{B}}$ differ from each other in the pth column, so $\mathbf{B} - \bar{\mathbf{B}}$ is a matrix, where every entry is zero, expect the pth column. Thus in practice, to compute $\phi^T (\mathbf{B} - \bar{\mathbf{B}})$ we need a vector subtraction and a dot product. Similarly to the phase-2 reduced costs, the phase-1 reduced costs can be updated using these observations efficiently. Denote the change in the phase-1 simplex multiplier with $\Delta \phi^T$:

$$\Delta \phi^T = (\phi^T (\mathbf{B} - \bar{\mathbf{B}}) + \Delta \mathbf{h}^T) \bar{\mathbf{B}}^{-1}.$$

In this case the phase-1 reduced costs can be computed as follows: We know that

$$\mathbf{d}_{\mathbf{R}}^T = \phi^T \mathbf{R},$$

therefore

$$\bar{\mathbf{d}}_{\mathbf{R}}^T = \phi^T \mathbf{R} + \Delta \phi^T \mathbf{R} = \mathbf{d}_{\mathbf{R}}^T + \Delta \phi^T \mathbf{R} \tag{3.5}$$

Before computing the simplex multiplier, it has to be verified whether vector $\phi^T (\mathbf{B} - \bar{\mathbf{B}}) + \Delta \mathbf{h}^T$ has less nonzeros than $\bar{\mathbf{h}}$ or not, because if $\bar{\mathbf{h}}$ has more nonzeros, then the usage of (3.5) is preferred.

The computation of $\phi^T (\mathbf{B} - \bar{\mathbf{B}})$ can be implemented by using three successive loops with appropriate vector representations. Since the coefficient matrix contains very few nonzeros, only these values are stored: Vectors are stored as (idx, v) pairs, where idx gives the index of value v ($v \neq 0$) in the vector. This is the so-called sparse vector representation technique. Column and row vectors of $\mathbf{B}$ are stored this way as shown in Figure (3.1). It can slow down the algorithm if a single element of the vector has to be obtained, because this needs a search of complexity $O(n)$. However, the pricing module uses only nonzeros of the necessary vectors, so it is sufficient to go through the list of (idx, v) pairs only. This technique makes the program faster, because it doesn't have to complete operations

$$\mathbf{B} = \begin{bmatrix} 5 & & & 4 \\ & & 2 & \\ & 1 & & 2 \\ & & & \\ 1 & & 3 & \end{bmatrix} \equiv$$

$\mathbf{b}_1$			$\mathbf{b}_2$			$\mathbf{b}_3$			$\mathbf{b}_4$	
$\parallel$			$\parallel$			$\parallel$			$\parallel$	

idx	v
1	5
4	1

idx	v
3	1

idx	v
2	2
4	3

idx	v
1	4
3	2
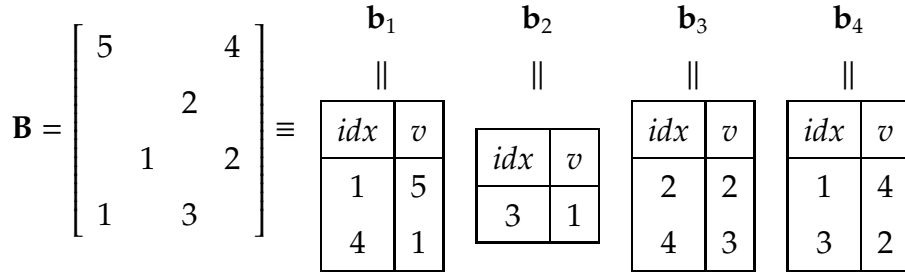
Figure 3.1: Ab example for column-wise matrix representation

where one of the values is zero. But the simplex multiplier ϕ is stored in direct form with its zero elements, since the algorithm reads components of ϕ randomly.

Usually, sparse representation does not ensure the ordering of (idx, v) pairs by idx, because operations on the vectors can modify the order of the elements. Maintaining the right order needs too much computational power. However, the simplex method changes matrix $\mathbf{B}$ by replacing its columns, but the elements in these vectors are unchanged. Since these elements do not vary, *sorting* (idx, v) pairs by idx in the initial phase makes sense. The question is which sorting algorithm is the most suitable? In different situations different methods are preferred. When there are few pairs, using *selection sort* is proposed. This algorithm has $O(n^2)$ complexity, where n is the number of elements. For more elements, a *counting sort* is the obvious choice, because the complexity is $O(n + k)$, where k is the range of elements. The counting sort can be simplified, because each index idx in the pairs is unique. Choosing the appropriate pair causes that the ordering of each vector in the coefficient matrix can be very time consuming, i.e. length of sorting time can be comparable with the total execution time of the simplex. However, choosing the right sorting algorithm based on a simple criterion, running time of sorting is negligible. Let n be the number of nonzeros in a vector, and r the difference between the largest nonzero index and the lowest nonzero index. Then the criterion for choosing sorting algorithm is the following:

$$n^2 \leq 4n + 2r := \begin{cases} \text{true:} & \text{selection sort} \\ \text{false:} & \text{counting sort} \end{cases}$$

Figure (3.2) shows an example for sorting times, where $r = 100$.

We know that $\mathbf{B}$ and $\bar{\mathbf{B}}$ differ from each other in one column vector; denote the vectors with $\mathbf{b}$ and $\bar{\mathbf{b}}$ respectively. This fact can be utilized to compute $\phi^T(\mathbf{B} - \bar{\mathbf{B}})$. Moreover, as it was mentioned above, $\mathbf{b}$ and $\bar{\mathbf{b}}$ are stored in *sparse form*, and the pairs (idx, v) are stored in ascending order. Finally, ϕ is in *direct form*, i.e. each component of ϕ is reachable directly. The algorithm computing $\mathbf{B}$ and $\bar{\mathbf{B}}$ utilizes these facts. It is similar to the merge sort: Each nonzeros of $\mathbf{b}$ and $\bar{\mathbf{b}}$ have to be visited. Let $(idx, v)_{\mathbf{b}}^k$ denote the kth index-value pair in vector $\mathbf{b}$, and similarly, $(idx, v)_{\bar{\mathbf{b}}}^l$ the lth pair in vector $\bar{\mathbf{b}}$. The algorithm has to compute the following formula: $\sum_{i=1}^{n} \phi_i(b_i - \bar{b}_i)$. At the beginning, l and k are 1, and some iterations are executed. The index k refers to the pair (idx_1, v_1), and l refers to the pair (idx_2, v_2). In each iteration the referred indices idx_1 and idx_2 are compared. Three cases can be distinguished:
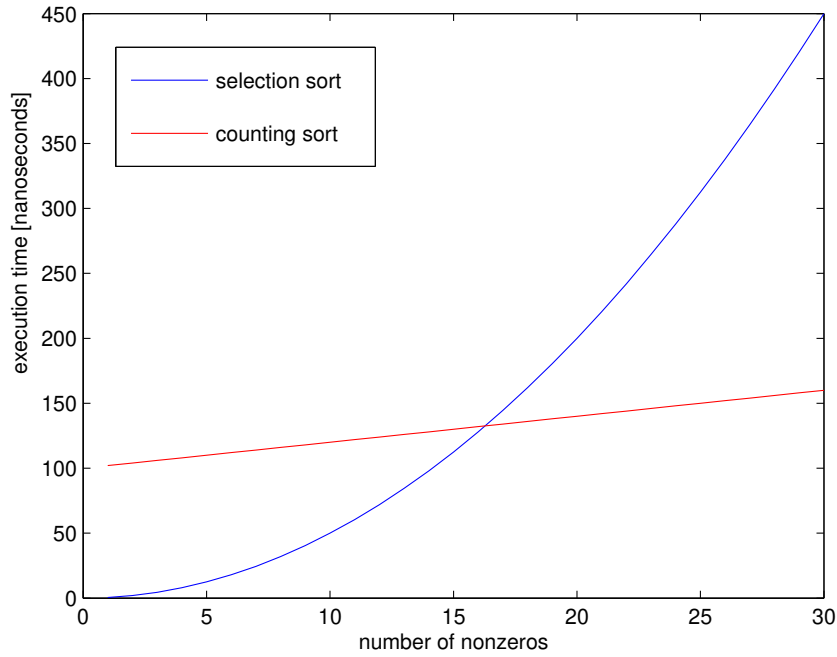
Figure 3.2: Execution times of different sorting algorithms

- $idx_1 = idx_2$ In this case the algorithm refers the $(idx_1, v_1)_{\mathbf{b}}^k$ and $(idx_2, v_2)_{\bar{\mathbf{b}}}^l$ pairs, so then the value $\phi_{idx_1}(v_1 - v_2)$ is computed.

- $idx_1 < idx_2$ In this case the algorithm refers the $(idx_1, v_1)_{\mathbf{b}}^k$ and $(idx_2, v_2)_{\bar{\mathbf{b}}}^l$ pairs. This means that there is no such a $(idx_2, v_2)_{\bar{\mathbf{b}}}^l$, where $idx_1 = idx_2$, so the algorithm computes the value $\phi_{idx_1}(v_1 - 0) = \phi_{idx_1} v_1$.

- $idx_1 > idx_2$ In this case the algorithm refers the $(idx_1, v_1)_{\mathbf{b}}^k$ and $(idx_2, v_2)_{\bar{\mathbf{b}}}^l$ pairs. This means that there is no such a $(idx_1, v_1)_{\mathbf{b}}^k$, where $idx_1 = idx_2$, so the algorithm computes the value $\phi_{idx_2}(0 - v_2) = -\phi_{idx_1} v_2$.

After these simple operations, k and l are increased by 1. Algorithm (3.1) shows the pseudo code of the algorithm, where $c(\mathbf{a})$ denotes the number of nonzeros in vector $\mathbf{a}$, and the operator $nz(\mathbf{a}, i)$ gives the pair $(idx, v)_{\mathbf{a}}^i$ from vector $\mathbf{a}$.

As computing (3.4) needs a BTRAN operation, it is useful to have a specific BTRAN implementation, exploiting *sparsity* of $\phi^T(\mathbf{B} - \bar{\mathbf{B}}) + \Delta \mathbf{h}^T$.

3.2 Column-wise BTRAN algorithm

It is shown in the previous sections that the pricing module heavily depends on the BTRAN operation, so implementing an efficient BTRAN algorithm is a fundamental in a simplex solver. In this section two implementations are introduced, and their efficiency will be investigated.

Algorithm 3.1 Algorithm for computing $\boldsymbol{\phi}^T(\mathbf{B} - \bar{\mathbf{B}})\mathbf{e}_p$

Input: b, $\bar{\mathbf{b}}$ and $\boldsymbol{\phi}$ vectors
Output: s scalar

1: $s := 0$
2: $k := 1$
3: $l := 1$
4: **while** $k < c(\mathbf{b})$ és $l < c(\bar{\mathbf{b}})$ **do**
5:      Let $(idx_1, v_1)_{\mathbf{b}}^k := nz(\mathbf{b}, k)$
6:      Let $(idx_2, v_2)_{\bar{\mathbf{b}}}^l := nz(\bar{\mathbf{b}}, l)$
7:      **if** $idx_1 = idx_2$ **then**
8:          $s := s + \phi_{idx_1}(v_1 - v_2)$
9:          $k := k + 1$
10:          $l := l + 1$
11:      **else if** $idx_1 < idx_2$ **then**
12:          $s := s + \phi_{idx_1} v_1$
13:          $k := k + 1$
14:      **else**
15:          $s := s - \phi_{idx_2} v_2$
16:          $l := l + 1$
17:      **end**
18: **end**
19: **while** $k < c(\mathbf{b})$ **do**
20:      Let $(idx, v)_{\mathbf{b}}^k := nz(\mathbf{b}, k)$
21:      $s := s + \phi_{idx} v$
22:      $k := k + 1$
23: **end**
24: **while** $l < c(\bar{\mathbf{b}})$ **do**
25:      Let $(idx, v)_{\bar{\mathbf{b}}}^l := nz(\bar{\mathbf{b}}, l)$
26:      $s := s - \phi_{idx} v$
27:      $l := l + 1$
28: **end**

Recall the formula of BTRAN (1.24):

$$\boldsymbol{\alpha}^T = \mathbf{a}^T \mathbf{B}^{-1} = \mathbf{a}^T \mathbf{E}_s \mathbf{E}_{s-1} \ldots \mathbf{E}_1$$

Based on the above formula a simple algorithm can be used: To compute $\boldsymbol{\alpha}$ the $\boldsymbol{\eta}$ vectors from the ETMs and the corresponding p indices. Let us use the following notations:

- s: The number of ETMs representing $\mathbf{B}^{-1}$

- $\boldsymbol{\eta}_i$: The $\boldsymbol{\eta}$ vector of the ith ETM

- p_i: The column index of $\boldsymbol{\eta}_i$ in the ith ETM

- $\mathbf{a}$: The vector to be transformed, an m dimensional vector

Let's suppose that vector $\mathbf{a}$ is given in dense form.. The advantage of this storage method is that the complexity of obtaining an element of the vector is $O(1)$. The vector needs more memory in this way, but in this situation this is acceptable. The vectors $\boldsymbol{\eta}_i$ are given in sparse form, i.e. the $(idx, v)_{\eta_i}^k$ pairs are stored, where $1 \leq k \leq c(\boldsymbol{\eta}_i)$.

During BTRAN, dot products of the vectors $\boldsymbol{\eta}_i$ and $\mathbf{a}$ are computed, and the result is substituted into the p_i^{th} position of vector $\mathbf{a}$, as it was shown in Algorithms (3.2) and (3.3).

Algorithm 3.2 BTRAN algorithm using column-wise η representation

---

Input: a
Output: α
 1: **for** $i := s$ to 1
 2:     $d := 0$
 3:         **for each** $\eta_i^j \neq 0$:
 4:             $d := d + a_j \eta_i^j$
 5:         **end**
 6:         $a_{p_i} := d$
 7: **end**
 8: $\alpha := \mathbf{a}$

Analysis of the column-wise BTRAN algorithm

The execution time of algorithm (3.2) depends on the number of η vectors, and the number of nonzeros in these vectors. The disadvantage of this algorithm is that a result of multiplication $a_j \eta_i^j$ (line 4) is often zero. In this case d does not change, thus the algorithm performs too many unnecessary multiplications. The second disadvantage is that the execution time is independent from the number of nonzeros in the input vector $\mathbf{a}$. Figure (3.3) shows an example: There is a step, where the vector remains unchanged. A BTRAN implementation is free from these two disadvantages, so it can improve the efficiency of the pricing algorithm.

$p_4 = 4$ $p_3 = 2$ $p_2 = 1$ $p_1 = 7$

η vectors:

1	5
3	2
5	0.5
6	2

3	4
1	7
8	0.2

6	1

4	2
5	3

input vector **a**$_0$:

0	2	0	0	5	5	0	0
1	2	3	4	5	6	7	8

$i = 4$
$d := 5*0 + 2*0 + 0.5*5 + 5*0 = 2.5$

vector **a**$_1$:

0	2	0	2.5	5	0	0	0
1	2	3	4	5	6	7	8

$i = 3$
$d := 4*0 + 7*0 + 0.2*0 = 0$

vector **a**$_2$:

0	0	0	2.5	5	0	0	0
1	2	3	4	5	6	7	8

$i = 2$
$d := 1*0 = 0$

vector **a**$_3$:

0	0	0	2.5	5	0	0	0
1	2	3	4	5	6	7	8

$i = 1$
$d := 2*2.5 + 3*5 = 20$

output vector **α**:

0	0	0	2.5	5	0	20	0
1	2	3	4	5	6	7	8
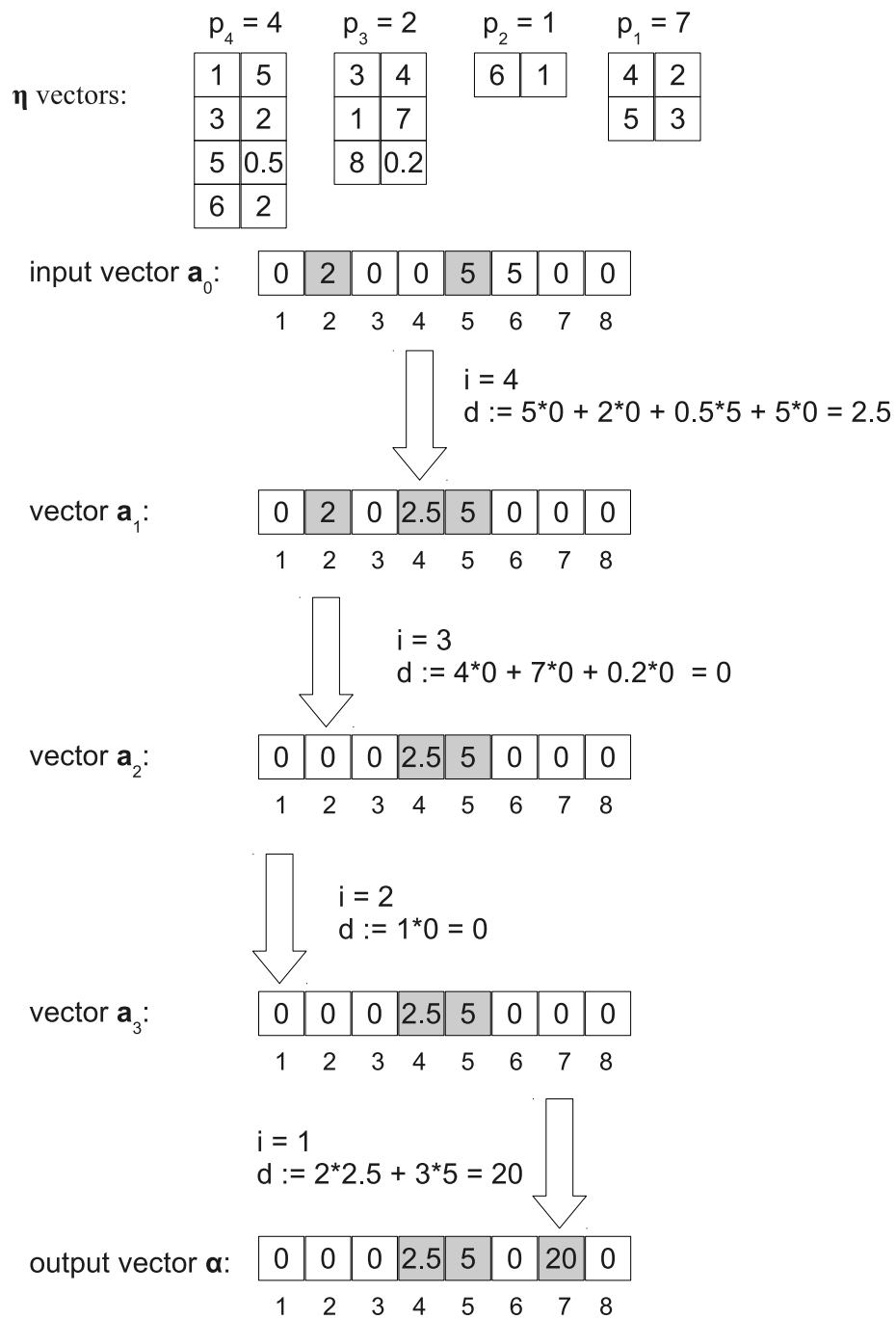
Figure 3.3: Example for a column-wise BTRAN implementation
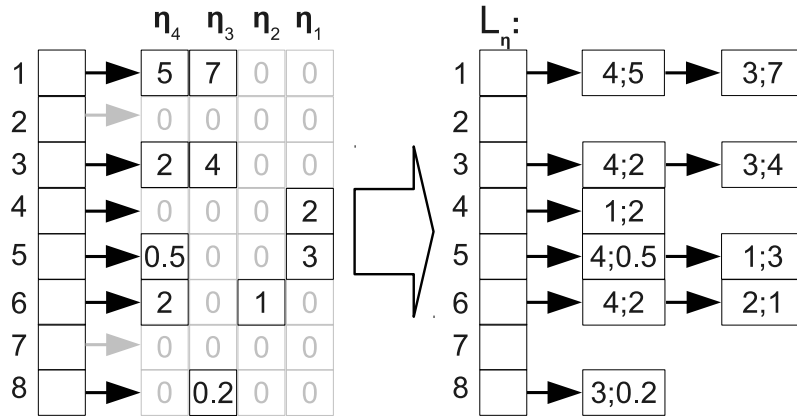
Figure 3.4: The η vectors of example (3.3) using row-wise representation

We investigated several test problems. The number of $a_j \eta_i^j$ products with nonzero results were counted for each dot product in a BTRAN. Also, the distribution of these counters were examined. Figures (3.5) and (3.6) show two examples, it can be seen that the typical number of nonzero results in a dot product of a BTRAN is usually very low.

3.3 Row-wise BTRAN algorithm

A more efficient BTRAN implementation can be achieved using an alternative method for storing the η vectors. Recall that only the nonzeros of these vectors and positions of nonzeros in the vectors are stored, and the number of these vectors is s. The vectors η and $\mathbf{a}$ have m components. Introduce an array called L_η, which contains references to m different lists. Each list contains (idx, v) pairs as follows: The list j contains the η_i^j values from vector η_i, where $\eta_i^j \neq 0$. When these lists are constructed, the descending order of these pairs has to be maintained by indices. Fortunately, it is easy to fulfill this requirement: Creating a new ETM means that a new η vector has to be appended to the list of η vectors, i.e. the nonzero elements of the new vectors are inserted in to the front of the lists. The insertion of a new element has $O(1)$ complexity. Obviously, the nonzeros of the η_s vector are at the beginning of the lists and the values of η_1 are at the end of the lists, as Figure (3.4) shows.

In the first step, the BTRAN that uses this new data structure collects the values and indices of nonzeros from the input vector $\mathbf{a}$. If $a_j \neq 0$, then the algorithm will need the first element of the linked list $L_\eta[j]$. This information was collected by using an other s length array named $L_\mathbf{a}$, which stores linked lists. The linked list $L_\mathbf{a}[j]$ contains such pointers, which refers to the elements of linked lists stored by array L_η: When $a_j \neq 0$, then the algorithm examines, which element of the current η_i does the first element of the list $L_\eta[j]$ belongs to. Namely, the lists of the array $L_\mathbf{a}$ represents that η components that have to be
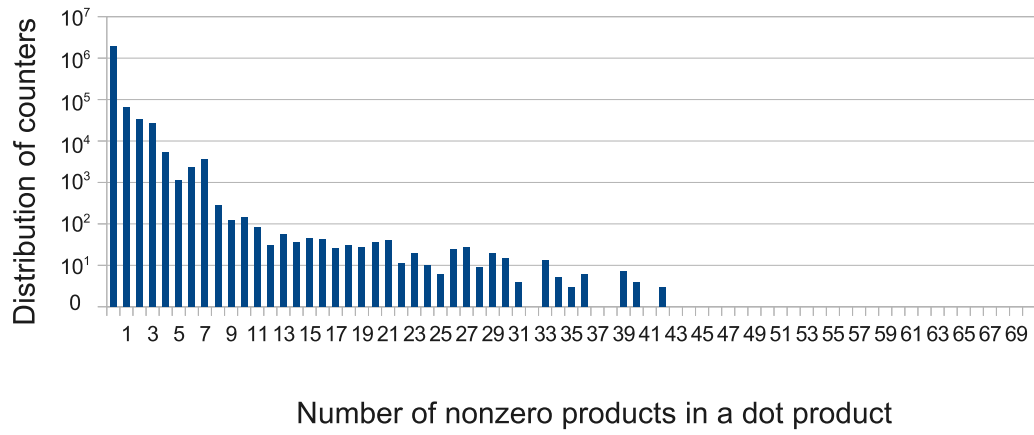
Figure 3.5: Distribution of nonzero products in STOCFOR2. The number of rows is 2156
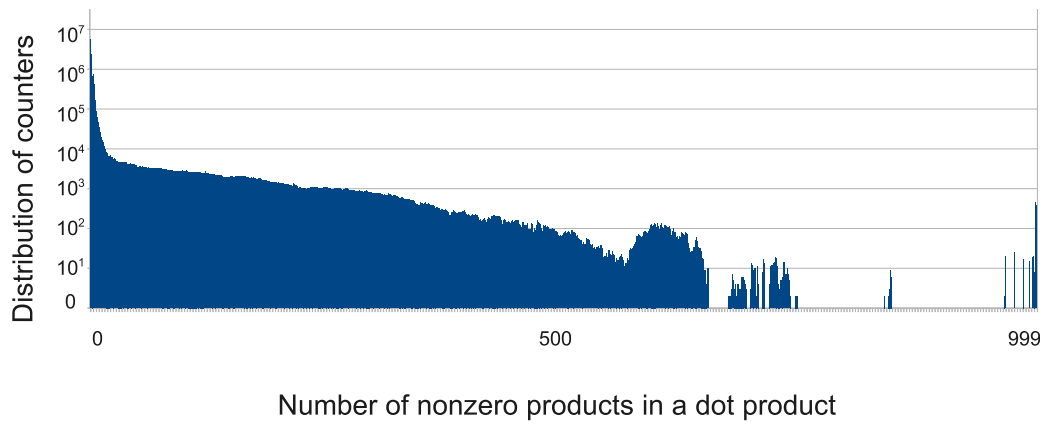
Figure 3.6: Distribution of nonzero products in TRUSS. The number of rows is 999
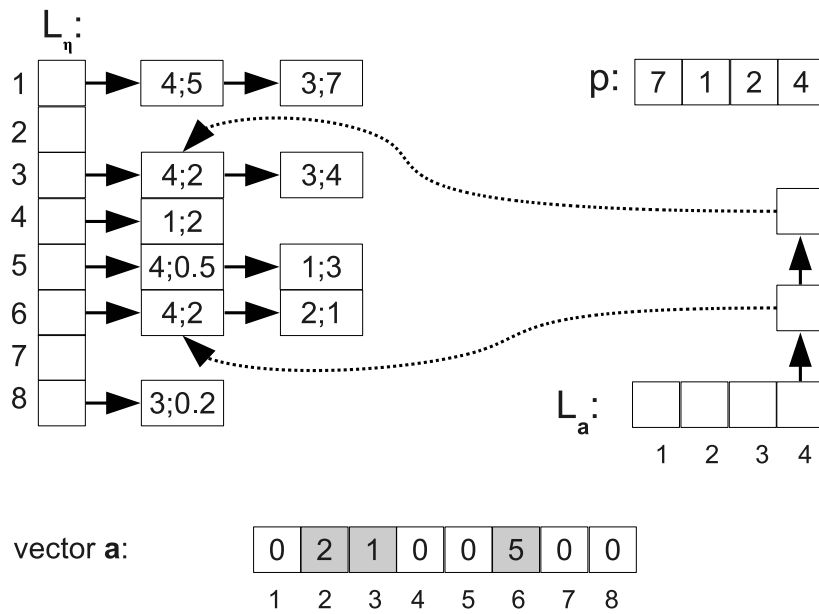
Figure 3.7: An example for the relationship of L_η and L_a arrays

multiplied with the nonzeros of **a**. The algorithm will maintain the processed elements of the L_η lists. In the second step, a pointer is inserted in to the list $L_a[i]$, which refers to the first element of the list $L_\eta[j]$. Figure (3.7) shows an example for this data structure: The figure shows an example input vector **a**, where $a_2 = 2$, $a_3 = 1$ and $a_6 = 5$. There are four η vectors, so at first we have to multiply **a** and η_4. The η_4 has four nonzero elements, but we only have to use the η_4^3 and η_4^6, because $a_3\eta_4^3 \neq 0$, $a_6\eta_4^6 \neq 0$ and any other products are zero. Therefore, we have to insert pointers to η_4^3 and η_4^6 into the list L_a.

From this data structure it can be read with which vector η_i and **a** have to be multiplied: The algorithm chooses the last element of the array L_a, which index denoted by *idx*. Now the list $L_a[idx]$ shows which elements of η_{idx} have to be multiplied with the corresponding elements of the vector **a**. This ensures that the products will be nonzeros, so the algorithm does not perform unnecessary multiplications.

Denote *pivotIndex* the value of p_{idx}. The dot product gives the new value of $a_{pivotIndex}$, and the algorithm decreases *idx* by 1, i.e. it chooses the previous list of L_a. There are three cases when $a_{pivotIndex}$ will be updated:

- The value of $a_{pivotIndex}$ was zero, and it remains zero, or it was nonzero, and remains nonzero also. In this case, extra administration steps are unnecessary.

- If the result is not zero, but the $a_{pivotIndex}$ was zero before, the algorithm have to find the first $\eta_j^{pivotIndex}$ element of $L_\eta[pivotIndex]$, where j is the utmost but less than *idx*, and it inserts its pointer into the $L_a[j]$. Finding this element can be perform in $O(\log_2)$ time, using an appropriate linked list data structure. Figure (3.9) shows this situation: The a_4 increases from zero to 4.5, i.e. gets **a** new nonzero. The current *idx*
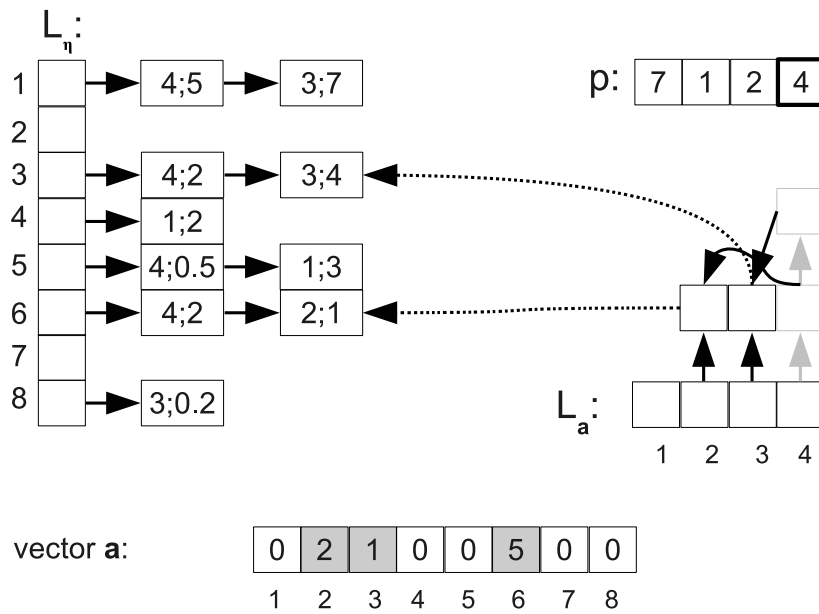
Figure 3.8: Extracting elements from the linked list $L_a[idx]$

is 3. We have to look for an η_j^4 in $L_\eta[4]$, where $j < idx$. The algorithm finds the $\eta_1^4 = 2$ element, so the algorithm inserts the pointer of the η_1^4 to $L_a[1]$.

- The result of a dot product can be zero. In this case the algorithm removes that pointer from L_a, which refers an element of L_η that belongs to the linked list *pivotIndex*. This pointer can be obtained by a search algorithm, which complexity is $O(m)$. However, this problem can be solved using redundancy: The array L_d contains the corresponding pointer from L_a to each nonzero element of **a**. This array ensures that finding above pointer has $O(1)$ complexity.

After each multiplication the elements of $L_a[idx]$ will be removed or moved into an other linked list, as Figure (3.8) shows. If a pointer refers to an η_i^j in L_η which has a next element, then the algorithm adds the pointer of this next element to the corresponding L_a list. If the pointer was the last one in the L_η list, it will be erased.

Aspects of implementation considerations

In this section we introduce a few implementation techniques that improve the performance of the row-wise BTRAN.

The lists of the array L_η are implemented as arrays. Each array has a variable named size that is equal to the number of stored elements, and an other variable named capacity shows the capacity of the array. The size should never exceed the capacity. If the capacity is greater than the size, it means that there is place for further elements in the array. When these variables are equal, then we allocate a greater array. In certain iterations the simplex
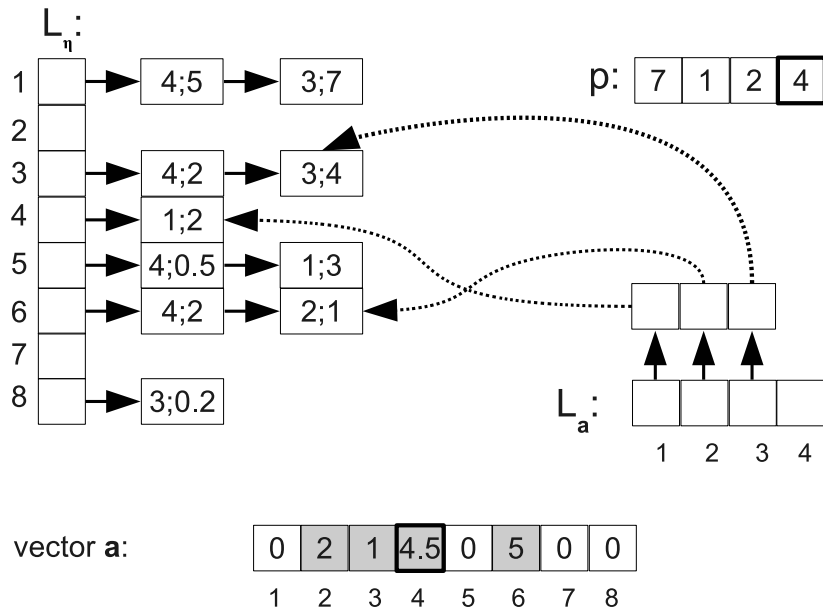
Figure 3.9: A new element was added to vector **a** so $L_\mathbf{a}$ also gets a new element
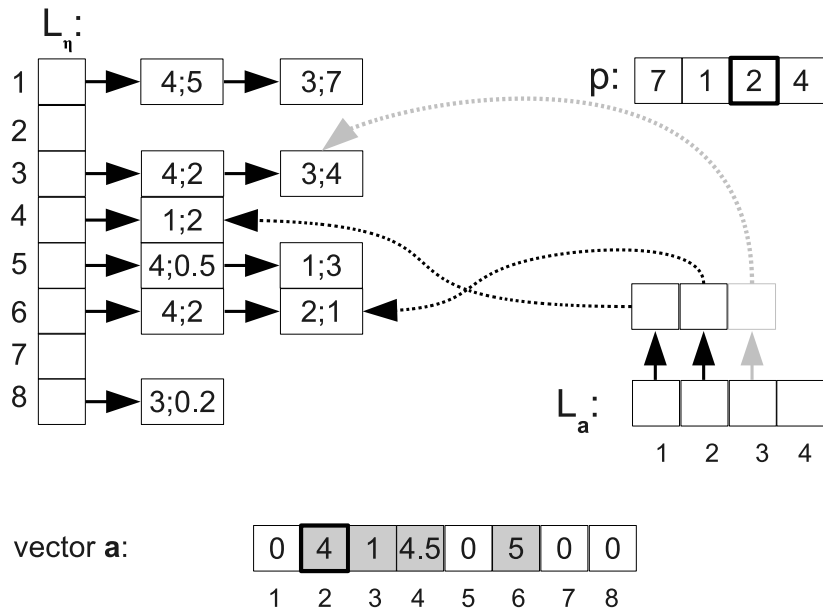
Figure 3.10: The second element of **a** changes, but $L_\mathbf{a}$ remains the same

Algorithm 3.3 Pseudo code of the row-wise BTRAN algorithm

Input: a
Output: α
Remark: s: number of the η vectors
Initialization
1: $idx := s$
2: **for each** $a_j \neq 0$:
3:      **if** $L_\eta[j]$ is not empty **then**
4:          $(i, v) :=$ first element of $L_\eta[j]$
5:          inserting a pointer that refers (i, v) to $L_\mathbf{a}[i]$
6:      **end**
7: **end**
Dot products
8: **while** $idx > 0$
9:      $d := 0$
10:     **for each** (i, v) pointed by $L_\mathbf{a}[idx]$
11:         Let $L_\eta[j]$ the linked list that includes (i, v)
12:         $d := d + va_j$
13:         **if** $next(i, v)$ exits **then**
14:             Let $(i', v') := next(i, v)$
15:             A pointer that refers to (i', v') has been inserted into $L_\mathbf{a}[i']$
16:         **end**
17:         Delete (i, v)
18:     **end**
19:     $pivotIndex := p_{idx}$
20:     **if** $a_{pivotIndex} = 0$ and $d \neq 0$ **then**
21:         **if** (i, v) exits in $L_\eta[pivotIndex]$, where $i < idx$ **then**
22:             insert pointer of (i, v) into $L_\mathbf{a}[i]$
23:         **end**
24:     **end**
25:     **if** $a_{pivotIndex} \neq 0$ and $d = 0$ **then**
26:         Delete from $L_\mathbf{a}$ the pointer that refers such a (i, v),
27:             that is in $L_\eta[pivotIndex]$
28:     **end**
29:     $a_{pivotIndex} := d$
30:     $idx := idx - 1$
31: **end**
32: $\alpha := \mathbf{a}$
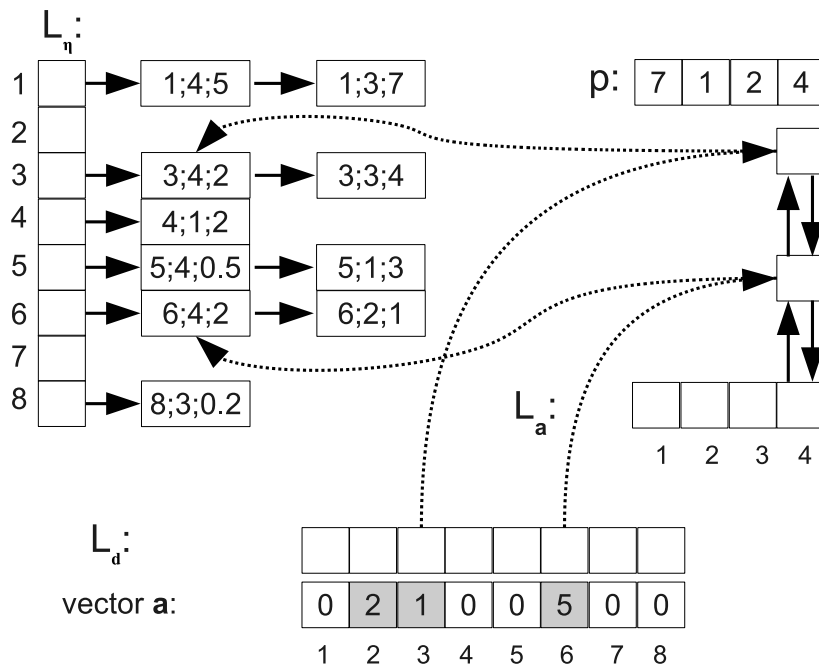
Figure 3.11: Each data structure of the row-wise BTRAN algorithm

algorithm drops the collected ETMs, and substitutes them with less but equivalent ETMs. We call this operation reinversion. The reason of reinversion is the improvement of numerical stability, and the large number of ETMs has a negative effect on the execution time of FTRAN and BTRAN. Thus at reinversion we flush every lists of L_η, before filling them with new data. Using simple arrays for the lists, we can reset the variable size to zero, i.e. we perform a fast logical deletion only, and the new elements will overwrite the old ones. The other advantage of this method is that in the first simplex iterations the capacities of the arrays grow to the necessary size, and later the likelihood of the array reallocation will be extremely low. Moreover, getting the next element of an element in the list needs only one operation. Finally, this approach enables binary search in the lists.

However, we have to use real linked lists in L_a, because sometimes the algorithm removes one element of a linked list. Using array instead of linked list requires moving the elements forward after the removed one, but this solution has $O(n)$ complexity. Removing an element from the linked list has $O(1)$ complexity. In C/C++, we use memory allocation and memory release operations for appending and removing elements in linked lists. However, too many repetitions of these memory operations bring down the efficiency of the algorithm. In our implementation we omit memory releasing when the algorithm deletes an element from a list: We store this element in a storage array. When the algorithm requires memory for a new element, we choose an element from the storage array. When this array is empty, we use the real memory allocation operation. As we know how many list elements will be used during the algorithm, we can utilize this information to make the algorithm more efficient: At most the length of the input vector $\mathbf{a}$, so we can prepare

									capacity / size:
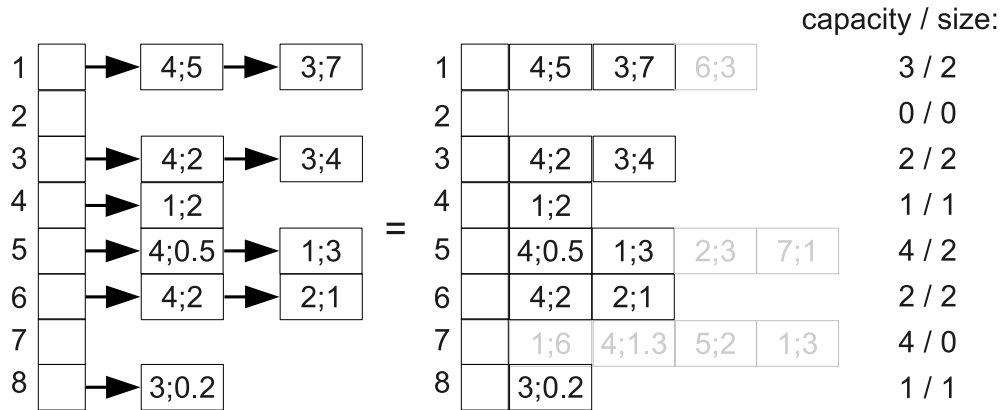
Figure 3.12: Implementation of the L_η lists

the pool of the linked list elements in the initialization phase. This method eliminates the overhead of memory management routines from the algorithm.

When we have a pair (idx, v) from L_η, we have to know, in which list $L_\eta[i]$ contains this pair. An additional redundancy can give this information: We can add a row index to this pair. The final data structure of the efficient implementation is showed by Figure (3.11).

3.4   Test results

The runtime of the row-wise BTRAN and column-wise BTRAN algorithms was tested on different LO problems. The result is shown in Appendix A. Of the 100 test problems, significant acceleration was observed in 64 cases. Our experience shows that the row-wise BTRAN is strong with a higher reinversion number, this value was 500 during the tests. If the reinversion number is 100, it is preferable to use a column-wise BTRAN. This is because the column-wise BTRAN algorithm is simpler, and this simpler program code is faster if there are fewer η vectors in the inverse. In contrast, a row-wise BTRAN handles more η vectors better, as it recognizes when a vector does not need to be taken into account. Furthermore, it can be observed that for smaller problems it is worth using the simpler algorithm as there are few η vectors there.

3.5   Major results and summary of accomplishments

One of the inverse of the basis representations used by the simplex method is the product form of the inverse, an important and extremely time-consuming procedure of its is the BTRAN (Backward Transformation) algorithm. An accelerated row-wise BTRAN algorithm with less addition and multiplication has been developed that significantly

reduces runtime for large problems.

Related conference presentation

- M. I. Smidla József. "Sorfolytonos szorzat alakú bázis inverz reprezentáció a primál szimplex módszerben". In: XXX. Magyar Operációkutatási Konferencia (Balatonőszöd, Magyarország). 2013

Chapter 4

Numerical stability

Today's simplex solvers can effectively solve large, somewhat unstable problems. However, there are LO problems that, although their optimal solution exists, the solver software still produce a false result. For example, both GLPK and COIN-OR open source software gives bad output for certain tasks (for example Rump's matrix and Hilbert matrix problems), because they are unable to recognize that they are working on numerically difficult problems. At the end of this chapter, we present a few examples of LO problems with which we compared the outputs of our own solver and these software. The reason for the difficulty is the numerical instability of these problems. In this chapter we describe the problems of numerical instability and then we present our solution.

In the first section, we present the condition number describing how stable a matrix is. We then show why it is not enough to calculate the condition number for **A** and, based on this information, we decide to use a slower but more stable method of solving. Finally, we describe a method for effectively detecting that the simplex method uses a numerically unstable basis.

It is important to note that this is a heuristic algorithm that we have designed to give a false positive result with very little probability for stable problems, rather a false negative result is typical.

4.1   The condition number

The *condition number* is a value that describes how a small change in the input of a problem affects the solution. The higher this value, the greater the change. There are matrices that have such a high number of condition numbers that even small numerical errors occurring during the computation can result in huge computational errors.

For the condition number, we need the definition of the vector and matrix norm.

Vector and matrix norm

The $\| \cdot \| : \mathbb{R}^n \to \mathbb{R}$ function is called *norm*, if

1. $\|\mathbf{x}\| \geq 0$ for $\forall \mathbf{x} \in \mathbb{R}^n$, and $\|\mathbf{x}\| = 0$ if and only if $\mathbf{x} = \mathbf{0}$

2. $\|\alpha \mathbf{x}\| = |\alpha|\|\mathbf{x}\|$ for $\forall \alpha \in \mathbb{R}, \mathbf{x} \in \mathbb{R}^n$

3. $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$ for $\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

Below we enumerate a few widely used vector norms.

- *p-norn*:

$$\|\mathbf{x}\|_p = \sqrt[p]{\sum_{i=1}^n |x_i|^p}$$

- l_1 *norm* or *Taxicab geometry*:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|$$

- l_∞ *norm* or *Maximum norm*:

$$\|\mathbf{x}\|_\infty = \max(|x_1|, \ldots |x_n|) = \lim_{p \to \infty} \|\mathbf{x}\|_p$$

- *2-norm* or *Euclidean norm*:

$$\|\mathbf{x}\|_p = \sqrt{\sum_{i=1}^n x_i^2}$$

Notice that the p-norm is the generalization of the l_1, l_∞ and 2-norms. The definition of the *induced matrix norm* on a matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$ is the following:

$$\|\mathbf{A}\| \equiv \sup_{\mathbf{x} \neq \mathbf{0}} \frac{\|\mathbf{A}\mathbf{x}\|}{\|\mathbf{x}\|}, \mathbf{x} \in \mathbb{R}^m$$

That is, the matrix norm determines how much a matrix-vector multiplication can increase the vector norm in case of a given matrix. The matrix norm has the following properties:

$$\|\mathbf{A}\| \geq 0, \forall \mathbf{A} \in \mathbb{R}^{n \times m}, \text{ and } \|\mathbf{A}\| = 0 \text{ if and only if } \mathbf{A} = \mathbf{0} \tag{4.1a}$$

$$\|\alpha \mathbf{A}\| = \|\alpha\| \times \|\mathbf{A}\|, \forall \alpha \in \mathbb{R}, \mathbb{A} \in \mathbf{R}^{n \times m} \tag{4.1b}$$

$$\|\mathbf{A} + \mathbf{B}\| \leq \|\mathbf{A}\| + \|\mathbf{B}\|, \forall \mathbb{A}, \mathbb{B} \in \mathbf{R}^{n \times m} \tag{4.1c}$$

$$\|\mathbf{A}\mathbf{B}\| \leq \|\mathbf{A}\| \times \|\mathbf{B}\| \tag{4.1d}$$

$$\|\mathbf{A}\mathbf{x}\| \leq \|\mathbf{A}\| \times \|\mathbf{x}\| \tag{4.1e}$$

The most common matrix norms are the following: Below we enumerate a few widely used matrix norms.

- p-norm:

$$\|\mathbf{A}\|_p = \sqrt[p]{\sum_{j=1}^{m} \sum_{i=1}^{n} |a_j^i|^p}$$

- l_1 norm or Taxicab geometry:

$$\|\mathbf{A}\|_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{ij}| \tag{4.2}$$

- l_∞ norm or Maximum norm:

$$\|\mathbf{A}\|_\infty = \max_{1 \le i \le m} \sum_{i=1}^{n} |a_{ij}| \tag{4.3}$$

It is known that the $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$ matrix norms are equivalent for each arbitrary α and β, that is, for the $(\|\cdot\|_\alpha, \|\cdot\|_\beta)$ pair, there is a pair of positive real numbers l, L, such that:

$$l\|\mathbf{A}\|_\alpha \le \|\mathbf{A}\|_\beta \le L\|\mathbf{A}\|_\alpha, \forall \mathbf{A} \in \mathbf{R}^{n \times m} \tag{4.4}$$

Consider the following equation:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{4.5}$$

Let us call $\delta\mathbf{b}$, and $\delta\mathbf{x}$ the errors of $\mathbf{b}$, and $\mathbf{x}$. If these errors are taken into account, (4.5) is modified as follows:

$$\mathbf{A}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \tag{4.6}$$

Subtract the equation (4.5) from (4.6) and we get:

$$\mathbf{A}\delta\mathbf{x} = \delta\mathbf{b} \tag{4.7}$$

It follows from the equations (4.7) and (4.1e) that:

$$\|\delta\mathbf{x}\| = \|\mathbf{A}^{-1}\delta\mathbf{b}\| \le \|\mathbf{A}^{-1}\|\|\delta\mathbf{b}\|$$

Moreover, we know that

$$0 < \|\mathbf{b}\| = \|\mathbf{A}\mathbf{x}\| \le \|\mathbf{A}\|\|\mathbf{x}\|$$

Consequently:

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \le \frac{\|\mathbf{A}^{-1}\|\|\delta\mathbf{b}\|}{\|\mathbf{x}\|} \le \frac{\|\mathbf{A}^{-1}\|\|\delta\mathbf{b}\|}{\|\mathbf{b}\|/\|\mathbf{A}\|} = \|\mathbf{A}^{-1}\|\|\mathbf{A}\|\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}$$

So the relative error of the **x** solution depends on the relative error of the right side of (4.5). How sensitive the vector **x** is to an error in **b** is determined by the condition number:

$$\kappa(\mathbf{M}) = \|\mathbf{M}^{-1}\|\|\mathbf{M}\| \tag{4.8}$$

For a complete row-ranked, non-square matrix, the condition number:

$$\kappa(\mathbf{M}) = \|\mathbf{M}^\dagger\|\|\mathbf{M}\|, \tag{4.9}$$

where $\mathbf{M}^\dagger$ is the so-called *pseudo inverse* of complete sequence matrices:

$$\mathbf{M}^\dagger = \mathbf{M}^T(\mathbf{M}\mathbf{M}^T)^{-1} \tag{4.10}$$

If we want to prepare our solver software for handling large condition number matrices, it is not enough to determine the condition number of the input matrix and decide which number representation to use. This is proven by the following example.

Let be $\mathbf{M} = [\mathbf{I}|\mathbf{O} + \epsilon\mathbf{I}]$, where $0 < \epsilon$, $O_{i,j} = 1$, and $\mathbf{O}, \mathbf{I} \in \mathbb{R}^{n \times n}$, and $\mathbf{M} \in \mathbb{R}^{n \times 2n}$, $n > 1$. By this $\lim\limits_{\epsilon \to 0} \dfrac{\kappa(\mathbf{M})}{\kappa(\mathbf{O} + \epsilon\mathbf{I})} = 0$.

Proof for norms $\| \cdot \|_1$:

If a matrix is composed of two types of values, one in the main diagonal and the other in the other, the two norms mentioned above are the same. In the following, the norm refers to these two norms.

At first, compute the value of $\kappa(\mathbf{O} + \epsilon\mathbf{I})$:

$$\kappa(\mathbf{O} + \epsilon\mathbf{I}) = \|\mathbf{O} + \epsilon\mathbf{I}\|\|(\mathbf{O} + \epsilon\mathbf{I})^{-1}\|$$

Taking into account the equations (4.2) and (4.3):

$$\|\mathbf{O} + \epsilon\mathbf{I}\| = n + \epsilon.$$

The elements of the matrix $(\mathbf{O} + \epsilon\mathbf{I})^{-1}$ can take a total of two values a, b, where a denotes the elements of the principal diagonal and b denotes the other items. We know that the product of $\mathbf{O} + \epsilon\mathbf{I}$ and the inverse matrix gives a unit matrix. So the values of a and b can be determined using a simple equation system:

$$1 = a(1 + \epsilon) + (n - 1)b$$

$$0 = b(1 + \epsilon) + a + (n - 2)b$$

Solving this equation system:

$$a = \frac{n + \epsilon - 1}{\epsilon(n + \epsilon)}$$

$$b = -\frac{1}{\epsilon(n + \epsilon)}$$

Based on these $\|(\mathbf{O} + \epsilon\mathbf{I})^{-1}\| = a + (n - 1)|b| = \dfrac{n + \epsilon - 1}{\epsilon(n + \epsilon)} + \dfrac{n - 1}{\epsilon(n + \epsilon)} = \dfrac{2n + \epsilon - 2}{\epsilon(n + \epsilon)}$. By this

$$\lim_{\epsilon \to 0} \|\mathbf{O} + \epsilon\mathbf{I}\|\|(\mathbf{O} + \epsilon\mathbf{I})^{-1}\| = \lim_{\epsilon \to 0}\left[(n + \epsilon)(\frac{2n + \epsilon - 2}{\epsilon(n + \epsilon)})\right] = \lim_{\epsilon \to 0}\frac{2n^2 + n\epsilon - 2n + 2n\epsilon + \epsilon^2 - 2\epsilon}{\epsilon(n + \epsilon)} =$$

$$\lim_{\epsilon \to 0}\frac{2n^2 - 2n + \epsilon(3n + \epsilon - 2)}{\epsilon(n + \epsilon)} = \lim_{\epsilon \to 0}\frac{2n^2 - 2n}{\epsilon(n + \epsilon)} + \lim_{\epsilon \to 0}\frac{\epsilon(3n + \epsilon - 2)}{\epsilon(n + \epsilon)} =$$

$$\lim_{\epsilon \to 0}\frac{2n^2 - 2n}{\epsilon(n + \epsilon)} + \lim_{\epsilon \to 0}\frac{3n + \epsilon - 2}{n + \epsilon} = \frac{2n^2 - 2n}{\lim_{\epsilon \to 0}\epsilon(n + \epsilon)} + \frac{\lim_{\epsilon \to 0} 3n + \epsilon - 2}{\lim_{\epsilon \to 0} n + \epsilon} = \infty + \frac{3n - 2}{n} = \infty$$

Thus, it appears that the matrix $\mathbf{O} + \epsilon\mathbf{I}$ is a poorly conditioned matrix, even if *epsilon* is small enough.

Now determine the value of $\kappa(\mathbf{M}) = \|[\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I}]\|\|[(\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I})^\dagger\|$. To do this, determine the inverse of $(\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I})^\dagger$.

$$(\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I})^\dagger = [\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I}]^T([\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I}][\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I}]^T)^{-1}$$

Based on previous findings $[\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I}][\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I}]^T = \mathbf{I} + (\mathbf{O} + \epsilon\mathbf{I})(\mathbf{O} + \epsilon\mathbf{I})^T = \mathbf{I} + \mathbf{O}\mathbf{O}^T + \epsilon\mathbf{O}\mathbf{I} + \epsilon\mathbf{I}\mathbf{O} + \epsilon^2\mathbf{I} = \mathbf{I} + n\mathbf{O} + 2\epsilon\mathbf{O} + \epsilon^2\mathbf{I} = \mathbf{I}(1 + \epsilon^2) + (n + 2\epsilon)\mathbf{O}$

So the matrix $\mathbf{I}(1 + \epsilon^2) + (n + 2\epsilon)\mathbf{O}$ consists of elements where the elements of the main diagonal are $1 + \epsilon^2 + n + 2\epsilon$ and the other elements are $n + 2\epsilon$. The inverse of this matrix can be obtained by a system of equations. For the sake of simplicity, let $\hat{\epsilon} = 1 + \epsilon^2$, $c = n + 2\epsilon$, and $d = c + \hat{\epsilon}$:

$$1 = a(c + \hat{\epsilon}) + c(n - 1)b$$

$$0 = b(c + \hat{\epsilon}) + ac + c(n - 2)b$$

Solving this for a and b results in the following:

$$a = \frac{c(n - 1) + \hat{\epsilon}}{\hat{\epsilon}(cn + \hat{\epsilon})} = \frac{c(n - 1) + (1 + \epsilon^2)}{(1 + \epsilon^2)(cn + (1 + \epsilon^2))}$$

$$b = -\frac{c}{\hat{\epsilon}(cn + \hat{\epsilon})} = -\frac{c}{(1 + \epsilon^2)(cn + (1 + \epsilon^2))}$$

We already know the elements of matrix $((\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I})^T(\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I}))^{-1}$, so we can write the norm of $(\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I})^\dagger$:

$$\left\| \begin{bmatrix} \mathbf{I} \\ \mathbf{O} + \epsilon \mathbf{I} \end{bmatrix} ((\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I}])^T(\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I}]))^{-1} \right\| =$$

$$\left\| \begin{bmatrix} ((\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I}])^T(\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I}]))^{-1} \\ (\mathbf{O} + \epsilon \mathbf{I})((\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I}])^T(\mathbf{I}|\mathbf{O} + \epsilon * \mathbf{I}]))^{-1} \| \end{bmatrix} \right\| =$$

$$\left\| \begin{bmatrix} \begin{bmatrix} a & b & \dots & b \\ b & a & \dots & b \\ \vdots & & \ddots & \vdots \\ b & b & \dots & a \end{bmatrix} \\ \begin{bmatrix} 1+\epsilon & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1+\epsilon \end{bmatrix} \times \begin{bmatrix} a & b & \dots & b \\ b & a & \dots & b \\ \vdots & & \ddots & \vdots \\ b & b & \dots & a \end{bmatrix} \end{bmatrix} \right\|$$

First determine the norm $\| \cdot \|_1$. The latter matrix consists of $2n$ rows and n columns. Each column contains the same elements, only in a different order, so the norm is the same as the sum of the absolute values of the elements in any column. Take the first column, here the absolute sum of the first n elements is equal to $a + (n-1)|b|$. The absolute sum of the lower n elements is $|a(1 + \epsilon) + (n-1)b + (n-1)(a + (1+\epsilon)b + (n-2)b)|$. Since a and b always have ϵ in some subtotal, and $n, a > 0$, a $\lim_{\epsilon \to 0} \|(\mathbf{I}|\mathbf{O} + \epsilon\mathbf{I})^\dagger\|_1$ is easy to define:

$$\lim_{\epsilon \to 0} \|(\mathbf{I}|\mathbf{O} + \epsilon\mathbf{I})^\dagger\|_1 =$$

$$\lim_{\epsilon \to 0} [a + (n-1)|b| + |a(1 + \epsilon) + (n-1)b + (n-1)(a + (1+\epsilon)b + (n-2)b)|] =$$

$$\lim_{\epsilon \to 0} [a + (n-1)|b| + |a + (n-1)b + (n-1)(a + b + (n-2)b)|] =$$

$$\lim_{\epsilon \to 0} [|n(a - b + b * n| + a + (n-1)|b|] =$$

$$\lim_{\epsilon \to 0} \left[\left| n\left(\frac{c(n-1) + (1+\epsilon^2) + c - nc}{(1+\epsilon^2)(cn + (1+\epsilon^2))} \right) \right| + \frac{c(n-1) + (1+\epsilon^2) + (n-1)c}{(1+\epsilon^2)(cn + (1+\epsilon^2))} \right] =$$

We know that $c = n + 2\epsilon$, so $\lim_{\epsilon \to 0} c = n$ so we can continue the above formula like this:

$$\lim_{\epsilon \to 0} \left[\left| n\left(\frac{n(n-1) + (1+\epsilon^2) + n - n^2}{(1+\epsilon^2)(n^2 + (1+\epsilon^2))} \right) \right| + \frac{n(n-1) + (1+\epsilon^2) + (n-1)n}{(1+\epsilon^2)(n^2 + (1+\epsilon^2))} \right] =$$

$$\left| n\left(\frac{n(n-1) + 1 + n - n^2}{(n^2 + 1)} \right) \right| + \frac{n(n-1) + 1 + (n-1)n}{(n^2 + 1)} =$$

$$\frac{n}{n^2 + 1} + \frac{2(n-1)n + 1}{n^2 + 1}$$

That is in the case of the norm $\|\cdot\|_1$,

$$\lim_{\epsilon \to 0} \|\mathbf{M}^\dagger\|_1 = \frac{n}{n^2 + 1} + \frac{2(n-1)n + 1}{n^2 + 1},$$

and

$$\lim_{\epsilon \to 0} \|\mathbf{M}\|_1 = n.$$

Based on these

$$\lim_{\epsilon \to 0} \kappa(\mathbf{M}) = \left(\frac{n}{n^2 + 1} + \frac{2(n-1)n + 1}{n^2 + 1}\right) n = \frac{2n^3 - n^2 + n}{n^2 + 1}$$

We know that $\lim_{\epsilon \to 0} \kappa(\mathbf{O} + \epsilon \mathbf{I}) = \infty$, therefore, for the norm $\|\cdot\|_1$, the original statement is that $\lim_{\epsilon \to 0} \dfrac{\kappa(\mathbf{M})}{\kappa(\mathbf{O} + \epsilon \mathbf{I})} = 0$ is true.

Generalization to other norms

The theorem's (4.4) result is that

$$\lim_{\epsilon \to 0} \|\mathbf{A}\|_\alpha = \infty \Leftrightarrow \lim_{\epsilon \to 0} \|\mathbf{A}\|_\beta = \infty, \forall \mathbf{A} \in \mathbf{R}^{n \times m} \tag{4.11}$$

and

$$\lim_{\epsilon \to 0} \|\mathbf{A}\|_\alpha = L_1 \Leftrightarrow \lim_{\epsilon \to 0} \|\mathbf{A}\|_\beta = L_2, \forall \mathbf{A} \in \mathbf{R}^{n \times m}, L_1, L_2 \in \mathbf{R} \tag{4.12}$$

We saw that for the norm $\|\cdot\|_1$:

$$\lim_{\epsilon \to 0} \|\mathbf{O} + \epsilon \mathbf{I}\|_1 = L, 0 < L,$$

that is, the limit approaches a finite value L, and

$$\lim_{\epsilon \to 0} \|(\mathbf{O} + \epsilon \mathbf{I})^{-1}\|_1 = \infty$$

So, if we calculate the condition number with the general norm $\|\cdot\|$, then based on (4.11) and (4.12)

$$\kappa(\mathbf{O} + \epsilon \mathbf{I}) = \|\mathbf{O} + \epsilon \mathbf{I}\|\|(\mathbf{O} + \epsilon \mathbf{I})^{-1}\| = \infty$$

is also true. As stated earlier, and based on (4.12), we know that for the general norm $\|\cdot\|$ it is also true that

$$\lim_{\epsilon \to 0} \|\mathbf{M}^\dagger\|_\infty = L_3, 0 < L_3$$

and

$$\lim_{\epsilon \to 0} \|\mathbf{M}\|_\infty = L_4, 0 < L_4,$$

so

$$\lim_{\epsilon \to 0} \|\mathbf{M}^\dagger\| \|\mathbf{M}\| = L_5, 0 < L_5,$$

that is, the condition number $\kappa(\mathbf{M})$ is finite even for a general norm. Consequently, even when using a general norm, it is true that $\lim_{\epsilon \to 0} \dfrac{\kappa(\mathbf{M})}{\kappa(\mathbf{O} + \epsilon \mathbf{I})} = 0$.

$\square$

4.2 The Rounded Hilbert matrix

Hilbert introduced the **H** *Hilbert matrix* in his work in 1894 [34]. This is an n size square matrix, where the entries are the following fractions:

$$\mathbf{H}_{ij} = \frac{1}{i + j - 1}, \forall 1 \le i, j \le n$$

For instance, the 4×4 Hilbert matrix:

$$H_4 = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{bmatrix}$$

The inverse of the Hilbert matrix consists of integer numbers, where the elements can be computed with this formula:

$$(H_n)^{-1}_{ij} = (-1)^{i+j}(i + j - 1)\binom{n + i - 1}{n - j}\binom{n + j - 1}{n - i}\binom{i + j - 2}{i - 1}^2$$

This is the inverse of the matrix $\mathbf{H}_4$:

$$\mathbf{H}_4^{-1} = \begin{bmatrix} 16 & -120 & 240 & -140 \\ -120 & 1200 & -2700 & 1680 \\ 240 & -2700 & 6480 & -4200 \\ -140 & 1680 & -4200 & 2800 \end{bmatrix}$$

It is known that the Hilbert matrix is ill-conditioned; Table (4.1) illustrates the maximum absolute values in the matrices and their condition numbers. It can be seen that this

matrix is very ill-conditioned; for example, the Gaussian-elimination implementations used 64-bit floating-point numbers cannot compute the inverse matrix.

Size	$\max(\|H_{ij}^{-1}\|)$	$\kappa(\mathbf{H}_n)$
2	12	19.28147
4	6480	15513.739
5	179200	476607.25
6	4410000	14951059
10	3.481×10^{12}	1.526×10^{10}
12	3.659×10^{15}	1.422×10^{16}
15	1.147×10^{20}	9.329×10^{19}
20	3.614×10^{27}	2.881×10^{25}
50	1.290×10^{73}	3.573×10^{53}

Table 4.1: Condition numbers

As it is shown in Chapter (2), not every number is representable in floating-point number systems; these problematic numbers are rounded. Consequently, if we write a software that stores a Hilbert-matrix in the memory, we store only the rounded elements of the matrix. We will refer to this modified matrix as the *rounded Hilbert matrix*, and we denote it with the following notation:

$$\text{RN}(\mathbf{H}_n) = \begin{bmatrix} 1 & \cdots & \text{RN}(\frac{1}{1+j-1}) & \cdots & \text{RN}(\frac{1}{1+n-1}) \\ \vdots & \ddots & & & \vdots \\ \text{RN}(\frac{1}{i+1-1}) & \cdots & \text{RN}(\frac{1}{i+j-1}) & \cdots & \text{RN}(\frac{1}{i+n-1}) \\ \vdots & & \ddots & & \vdots \\ \text{RN}(\frac{1}{n+1-1}) & \cdots & \text{RN}(\frac{1}{n+j-1}) & \cdots & \text{RN}(\frac{1}{2n-1}) \end{bmatrix} \tag{4.13}$$

4.2.1 Rounded Hilbert matrix example problem

From the Rounded Hilbert matrix we can define an extremely unstable LO problem:

$$\min z = \quad 0 \tag{4.14}$$
$$s.t. \quad \text{RN}(\mathbf{H})\mathbf{x} = \mathbf{b} \tag{4.15}$$
$$\mathbf{0} \le \mathbf{x} \tag{4.16}$$

where the right-hand side is the sum of the corresponding row of the matrix:

$$b_i = 2.025 \sum_{j=1}^{n} RN\left(\frac{1}{i+j-1}\right) \tag{4.17}$$

Obviously, for exact (non-rounded) values there is only one possible solution, where all $x_j = 2.025$, for all $1 \le j \le n$. The simplex algorithm has to invert the Hilbert matrix to obtain this solution. Because of the equality type of constraints, the logical variables are fixed with zero. If the starting basis is a logical basis, the starting solution is infeasible, because the logical variables are the basic variables, and their value is nonzero. In this case, the classic dual simplex algorithm is preferred. Below we demonstrate the solution process of a simple Rounded Hilbert matrix problem with rounded matrix elements, but using exact arithmetic. After that, we solve the same problem using 16-bit floating-point numbers to demonstrate the weak points of this number system if we would like to solve this problem. For the sake of a simpler demonstration, let $n = 4$. The rounded matrix of this matrix is the following:

$$RN(\mathbf{H}_4) = \begin{bmatrix} 1 & 0.5 & 0.333252 & 0.25 \\ 0.5 & 0.333252 & 0.25 & 0.1999512 \\ 0.3332520 & 0.25 & 0.1999512 & 0.166626 \\ 0.25 & 0.1999512 & 0.166626 & 0.1428223 \end{bmatrix}$$

This is the correct inverse of this matrix, where the fractional parts are rounded to 4 digits:

$$RN(\mathbf{H}_4)^{-1} = \begin{bmatrix} 22.7985 & -190.5524 & 403.0549 & -243.3644 \\ -190.5524 & 1929.8555 & -4384.0736 & 2746.5030 \\ 403.0549 & -4384.0736 & 10362.6972 & -6657.6286 \\ -243.3644 & 2746.5030 & -6657.6286 & 4355.1226 \end{bmatrix}$$

It is clear that the difference of $RN(\mathbf{H}_4)$ and $\mathbf{H}_4$ is small:

$$\|RN(\mathbf{H}_4) - \mathbf{H}_4\|_F = 0.0001775$$

While the difference between $RN(\mathbf{H}_4)^{-1}$ and $\mathbf{H}_4^{-1}$ is much larger:

$$\|RN(\mathbf{H}_4)^{-1} - \mathbf{H}_4^{-1}\|_F = 6175.5905$$

The above matrix is the author's own example.

A note on computing of b

In both cases the matrix and **b** elements are computed with 16-bit floating-point numbers. As Figure (4.1) shows, there are numbers whose value is represented by their approximation, for example, $RN(\frac{1}{3}) = \frac{1365}{4096}$. Moreover, the computation method of **b** can be problematic. The obvious algorithm is the Algorithm (4.1): The algorithm adds items in a row as they come after each other. It is known that the results can be more accurate if we add the elements in a way that we add the two smallest numbers at first, and we treat this sum as the other items in the row, see Algorithm (4.2) [67]. But there are more sophisticated algorithms as well, we used Kahan's algorithm [40] here. This belongs to the group of *compensated sum* algorithms. We can mention some other, more accurate algorithms also, like Priest's double compensated summation algorithm [74, 75], Pichat and Neumaier's algorithm [73, 69], the cascaded summation of Rump, Ogita, and Oishi [78]. These algorithms make fewer errors in the sum, but Kahan's solution is sufficient for our demonstration purposes. The Table (4.2) contains the different b_i values depending on the summation algorithm. The last column shows the exact values, and underlines values that are the best approximations, it is clear that Kahan's algorithm produces an acceptable **b** vector.

Algorithm 4.1 Computing the vector **b**, naive version

Input: A
Output: b
1: **for** $i := 1$ to n
2:      $b_i := 0$
3:      **for** $j := 1$ to n
4:          $b_i := RN(b_i + A_i^j)$
5:      **end**
6: **end**

Algorithm 4.2 Computing the vector **b**, adding in increasing order

---

Input: A
Output: b
1: **for** $i := 1$ to n
2:      $\mathcal{S} := \{A_i^0, A_i^1, \ldots A_i^n\}$
3:      **while** $|\mathcal{S}| > 1$
4:          $a := \min(\mathcal{S})$
5:          $\mathcal{S} := \mathcal{S} \setminus \{a\}$
6:          $b := \min(\mathcal{S})$
7:          $\mathcal{S} := \mathcal{S} \setminus \{b\}$
8:          $\mathcal{S} := \mathcal{S} \cup \{RN(a + b)\}$
9:      **end**
10:      $b_i :=$ the only one element of $\mathcal{S}$
11: **end**

Algorithm 4.3 Computing the vector **b**, Kahan's algorithm

Input: A
Output: b

1: **for** $i := 1$ to n
2:      $s := A_i^0$
3:      $c := 0$
4:      **for** $j := 2$ to n
5:              $y := \text{RN}(A_i^j - c)$
6:              $t := \text{RN}(s + y)$
7:              $c := \text{RN}(\text{RN}(t - s) - y)$
8:              $s := t$
9:      **end**
10:     $b_i := s$
11: **end**

	Naive add	Increasing add	Kahan	Exact
b_1	4.21875	4.21484375	4.22265625	$\frac{38909580815606846191}{9223372036854775808} \approx 4.2185852$
b_2	2.599609375	2.59765625	2.599609375	$\frac{2995850767122195939}{1152921504606846976} \approx 2.5984863$
b_3	1.923828125	1.922851562	1.923828125	$\frac{35480540059326950687}{18446744073709551616} \approx 1.9234039$
b_4	1.5390625	1.538085938	1.538085938	$\frac{28367104447895252567}{18446744073709551616} \approx 1.5377838$
x_1	$\frac{12159235}{7261219} \approx 1.6745$	$\frac{3737895}{2074634} \approx 1.8017$	$\frac{29063199}{14522438} \approx 2.0013$	2.025
x_2	$-\frac{50460544}{7261219} \approx -6.9493$	$-\frac{3712912}{1037317} \approx -3.5793$	$\frac{8181232}{7261219} \approx 1.1267$	2.025
x_3	$\frac{6030062}{1037317} \approx 5.8131$	$\frac{4551145}{1037317} \approx 4.3874$	$\frac{2475719}{1037317} \approx 2.3867$	2.025
x_4	$\frac{56739562}{7261219} \approx 7.814$	$\frac{5859777}{1037317} \approx 5.649$	$\frac{18954417}{7261219} \approx 2.6104$	2.025

Table 4.2: Different **b** input vectors and solution vectors of a Rounded Hilbert matrix problem, after 4 iterations

Our final demo LO model is the following:

$$\min z: 0$$

$$y_1 \qquad\qquad + \quad x_1 \quad + \quad 0.5x_2 \quad + \quad 0.333252x_3 \quad + \quad 0.25x_4 \quad = \quad 4.22265625$$

$$y_2 \qquad + \quad 0.5x_1 \quad + \quad 0.333252x_2 \quad + \quad 0.25x_3 \quad + \quad 0.1999512x_4 \quad = \quad 2.599609375$$

$$y_3 \quad + \quad 0.333252x_1 \quad + \quad 0.25x_2 \quad + \quad 0.1999512x_3 \quad + \quad 0.166626x_4 \quad = \quad 1.923828125$$

$$y_4 \quad + \quad 0.25x_1 \quad + \quad 0.1999512x_2 \quad + \quad 0.166626x_3 \quad + \quad 0.1428223x_4 \quad = \quad 1.538085938$$

$$x_1, x_2, x_3, x_4 \geq 0, y_1, y_2, y_3, y_4 = 0$$

4.2.2 Analyzing the solutions

As Table (4.2) shows, if the vector **b** is computed by Kahan's algorithm, the simplex method finds a solution after 4 iterations. Still other cases, where **b** is a little bit different, i.e. it is computed by Algorithm (4.1) and (4.2), the solution is still infeasible.

	x_1	x_2	x_3	x_4	y_1	y_2	y_3	y_4	**x**
y_1	1	0.5	0.333252	0.25	1				4.22266
y_2	0.5	0.333252	0.25	0.199951		1			2.59961
y_3	0.333252	0.25	0.199951	0.166626			1		1.92383
y_4	0.25	0.199951	0.166626	0.142822				1	1.53809
d_j	0	0	0	0	0	0	0	0	0

Figure 4.1: The starting tableau of the Rounded Hilbert matrix problem. The most infeasible variable is y_1, and x_1 is the incoming variable.

Figures (4.1)-(4.5) show the process of the dual simplex method. The starting basis is the logical basis, so the variables $y_1 \ldots y_4$ are the basic variables. These variables are infeasible because they are fixed type variables; their feasible value is 0. We only need 4 iterations to obtain the solution; the algorithm moves every x variable into the basis.

The cells in Figures (4.2)-(4.5) have 3 parts: The first lines are the results of the 16-bit floating-point arithmetic. The values in the second line are the real values, they are computed by rational arithmetic. The last lines show the relative errors of the first lines (if the error is greater than zero). The pivot values are bordered.

It can be seen that the relative errors grow very quickly, the greatest error after the last iteration is 118%. But even the size of the error is not the main problem. Figure (4.5) shows, the computed value of $x_3 = -0.203125$, but the correct value is 1.1267. As we know, x_3 is a plus type variable, namely its feasible range is $x_3 \geq 0$. For the simplex

	x_1	x_2	x_3	x_4	y_1	y_2	y_3	y_4	$\mathbf{x}$
	1	0.5	0.333252	0.25	1				4.22266
x_1	1	0.5	0.333252	0.25	1				4.22266
		0.083252	0.083374	0.0749512	−0.5	1			0.488281
y_2		0.083252	0.083374	0.0749512	−0.5	1			0.488281
		0.083374	0.0888672	0.083313	−0.333252		1		0.516602
y_3		0.083374	0.0888943	0.083313	−0.333252		1		0.51662
			0.0305023 %						0.00350761 %
		0.0749512	0.083313	0.0803223	−0.25			1	0.482422
y_4		0.0749512	0.083313	0.0803223	−0.25			1	0.482422
d_j	0	0	0	0	0	0	0	0	0

Figure 4.2: The second tableau of the Rounded Hilbert matrix problem. The variable x_1 has a feasible variable, so the next outgoing variable is y_3, and x_2 moves into the basis. The value of y_3 has a small relative error.

algorithm, this means that the current basis is still infeasible, thus the algorithm will not terminate; it identifies x_3 as an outgoing variable. In fact, the simplex starts cycling in our example. Moreover, we cannot be sure that every variable that seems feasible (for example x_1, x_2 and x_4) is actually feasible.

A variable may be in a given range, but the solver software thinks this variable is outside of this range. However, the displacement is so small that we can treat this little difference as a numerical error. The simplex solvers use small tolerances for the checking of basic variable feasibilities (ϵ_f) or optimality (ϵ_o) conditions. For example, if the feasibility range of the plus-type basic variable x_j is $0 \le x_j$, its feasibility is checked by the following:

$$-\epsilon_f \le x_j (\text{instead of } 0 \le x_j)$$

Let $E(v)$ denote the exact value of a variable v, where v can be an element of $\mathbf{x}$ of $\mathbf{d}$ also, and $C(v)$ is its computed version. For example in the Figure (4.5) $E(x_3) = 1.1267$ and $C(x_3) = -0.203125$. If we can trust the computed variables, and the feasibility and optimality conditions met (by considering the tolerances), we can treat the calculated values as an acceptable optimal solution:

$$-\epsilon_f \le C(x_j) \quad \text{and} \quad -\epsilon_f \le E(x_j)$$

	x_1	x_2	x_3	x_4	y_1	y_2	y_3	y_4	$\mathbf{x}$
	1		−0.199463	−0.249756	3		−5.99609		1.125
x_1	1		−0.199854	−0.249634	2.99854		−5.99707		1.12445
			0.195435 %	0.0488281 %	0.0488281 %		0.0163116 %		0.0488281 %
			−0.00537109	−0.00823975	−0.167236	1	−0.998535		−0.0273438
y_2			−0.00539013	−0.00823984	−0.167236	1	−0.998536		−0.027582
			0.353271 %	0.00108433 %	0.000213742 %		7.15852e − 05 %		0.86377 %
		1	1.06543	0.999512	−3.99805		11.9922		6.19531
x_2		1	1.06621	0.999268	−3.99707		11.9941		6.19641
			0.0733032 %	0.0243988 %	0.0243988 %		0.0163116 %		0.0177155 %
			0.00341797	0.00543213	0.0495605		−0.898926	1	0.0180664
y_4			0.00339922	0.00542596	0.0495852		−0.898975	1	0.0179936
			0.551758 %	0.113647 %	0.0497437 %		0.00548553 %		0.404297 %
d_j	0	0	0	0	0	0	0	0	0

Figure 4.3: The third tableau of the Rounded Hilbert matrix problem. The outgoing variable is y_2, and x_3 moves into the basis. The computation errors are more significant, for example, y_4 has a 1.25% of error.

$$-\epsilon_o \le C(d_i) \quad \text{and} \quad -\epsilon_o \le E(d_i)$$

Let's define the following sets:

- $O_C = \{d_i :\ d_i$ is in the optimal range according to $C(d_i)\ \}$

- $O_E = \{d_i :\ d_i$ is in the optimal range according to $E(d_i)\ \}$

- $\mathcal{F}_C = \{x_j :\ x_j$ is in the feasible range according to $C(x_j)\ \}$

- $\mathcal{F}_E = \{x_j :\ x_j$ is in the feasible range according to $E(x_j)\ \}$

The current basis is numerically acceptable if and only if $O_C = O_E$ and $\mathcal{F}_C = \mathcal{F}_E$. If the basis numerically not acceptable, it means that we cannot trust the computed values, so we have to use more precise number representations.

However, if we would like to determine O_E and $\mathcal{F}_E$, we have to compute the current basis inverse using higher precision. The traditional simplex implementations use the Double format of IEEE 754-2008 standard. The modern CPUs have native hardware support for this format, so the calculations can be quite fast. We can use the quadruple format to approximate the $E(v)$. Unfortunately, there are only a few CPU architectures which supports this format natively, one of them is the IBM Power ISA 3.0 [93]. On X86, there is no hardware support, so we have to use the slower software implementations.

	x_1	x_2	x_3	x_4	y_1	y_2	y_3	y_4	**x**
	1			0.0561523	9.21094	−37.125	31.0938		2.14062
x_1	1			0.05588	9.19926	−37.0777	31.0263		2.14713
				0.487305 %	0.126953 %	0.127563 %	0.217285 %		0.302979 %
			1	1.53418	31.1406	−186.125	185.875		5.08984
x_3			1	1.52869	31.0263	−185.524	185.253		5.11714
				0.359131 %	0.368408 %	0.32373 %	0.335938 %		0.533203 %
		1		−0.635254	−37.1875	198.375	−186		0.773438
x_2		1		−0.630637	−37.0777	197.808	−185.524		0.740464
				0.731934 %	0.296143 %	0.286621 %	0.256348 %		4.45312 %
				0.00019455	−0.0568848	0.63623	−1.53418	1	0.000671387
y_4				0.000229615	−0.05588	0.630637	−1.52869	1	0.000599378
				15.2734 %	1.79785 %	0.886719 %	0.359131 %		12.0156 %
d_j	0	0	0	0	0	0	0	0	0

Figure 4.4: The fourth tableau of the Rounded Hilbert matrix problem. The last outgoing variable is y_4, and x_4 moves into the basis. The computation errors are unacceptable, for example, the error of y_4 is 65.5%!

Fortunately, the GNU Compiler can generate a code which uses this format. Table (4.3) shows the total basis reinversion times using 64 and 128-bit formats for different LO problems. Moreover, the table demonstrates the slowness of some advanced number formats; the mpf_class is the arbitrary precision data type of the GNU Multiple Precision Library, and mpq_class is the rational number data type. We used the PFI basis inverse representation, and the test environment was the following:

- CPU: Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz

- Memory: 16 GB

- Operating system: Debian 10, 64 bit

- Window manager: IceWM

- Compiler: gcc 8.3.0

As Table (4.3) shows, contrary to expectations for the larger problems, the 128-bit formats can speed-up the reinversion. The PFI format uses an advanced algorithm to produce the η vectors; it rearranges the rows and columns of the matrix in order to obtain sparser vectors to increase the speed of the current reinversion, the FTRAN, and BTRAN operations. If we use a more precise number representation, the reinverse algorithm

	x_1	x_2	x_3	x_4	y_1	y_2	y_3	y_4	**x**
	1				25.625	−220.75	473.75	−288.75	1.94727
x_1	1				22.7985	−190.552	403.055	−243.364	2.00126
					12.3984 %	15.8438 %	17.5469 %	18.6562 %	2.69727 %
			1		480	−5204	12280	−7884	−0.203125
x_3			1		403.055	−4384.07	10362.7	−6657.63	1.1267
					19.0938 %	18.7031 %	18.5 %	18.4219 %	118 %
		1			−223	2274	−5192	3266	2.96484
x_2		1			−190.552	1929.86	−4384.07	2746.5	2.38666
					17.0312 %	17.8281 %	18.4219 %	18.9219 %	24.2188 %
				1	−292.5	3270	−7884	5140	3.45117
x_4				1	−243.364	2746.5	−6657.63	4355.12	2.61036
					20.1875 %	19.0625 %	18.4219 %	18.0156 %	32.2188 %
d_j	0	0	0	0	0	0	0	0	0

Figure 4.5: The last tableau of the Rounded Hilbert matrix problem.
The accumulated relative errors make our solution unusable.

obtains more precise results, and it can produce fewer false-nonzero numbers, therefore the produced vectors will be a little bit sparser. The sparser vectors can compensate for the slower floating operations. However, the mpf_class and mpq_class types are more accurate, but they are much slower.

If we would like to implement a simplex solver which can detect that the current basis is numerically not acceptable with minimal overhead, we cannot compute basis inverse twice. We need a much faster method we can perform a primary test with. However, this test is only a heuristic, and we cannot competely trust it; if this primary test says that the basis is *maybe* numerically not acceptable, we will perform the much slower final test introduced above.

4.2.3   Primary test

The *primary test* performs a quick heuristic check on the current basis. Moreover, if the basis is stable, it is unnecessary to execute the primary test in every iteration. In our implementation, it is executed in every iteration with a probability of 0.05. Random behavior can help to avoid the effects of the structural properties of the current LO problem.

In every iteration, it is necessary to use an FTRAN and a BTRAN operation. After each step, we can perform a modified FTRAN and BTRAN (*FTRAN-CHECK* and *BTRAN-*

Name	64-bit [sec]	128-bit [sec]	mpf_class [sec]	mpq_class [sec]
MAROS-R7	26.1996	36.495	3202.25	> 3600
STOCFOR2	0.025799	0.025726	0.335758	0.152447
STOCFOR3	2.88615	2.9099	15.1871	10.6659
OSA-60	1.91193	1.89189	3.11958	2.94903
CRE-D	1.33815	1.41701	22.9924	> 3600
KEN-13	5.5606	5.41003	22.3411	69.0658
KEN-18	116.673	114.768	303.817	> 3600
PDS-10	1.29678	1.37789	7.64831	> 3600
PDS-20	8.37138	8.3885	69.3707	> 3600

Table 4.3: Reinversion times using different number representations.

CHECK) such a way that we use 128-bit internal numbers. The input and the η vectors store 64-bit numbers, and we convert them to 128-bit in order to use this type of precision during the algorithm. In the end, the result is converted back to 64-bit precision. If the basis is numerically not acceptable, the difference between the results of FTRAN (BTRAN) and FTRAN-CHECK (BTRAN-CHECK) is higher than a given μ threshold. Let denite α and α_{CHECK} the output of FTRAN and FTRAN-CHECK. This difference can be expressed in several ways, so we introduce the following *indicators*:

$$I_\alpha = \frac{\max(\|\alpha\|, \|\alpha_{\text{CHECK}}\|)}{\min(\|\alpha\|, \|\alpha_{\text{CHECK}}\|)} - 1 \tag{4.18}$$

$$I_\beta = \|\alpha - \alpha_{\text{CHECK}}\|_1 \tag{4.19}$$

$$max_1 = \epsilon_r \|\alpha\|_\infty \tag{4.20}$$

$$max_2 = \epsilon_r \|\alpha_{\text{CHECK}}\|_\infty \tag{4.21}$$

$$I_\gamma = \max \left\{ \frac{\|\alpha^i - \alpha^i_{CHECK}\|}{\max\{|\alpha^i|, |\alpha^i_{CHECK}|\}} : \text{if } \alpha^i > max_1 \text{ or } \alpha^i_{\text{CHECK}} > max2 \right\} \tag{4.22}$$

Here, I_α measures the summarized relative difference of the norm of the vectors. I_β is the absolute difference, and I_γ measures the largest relative difference of the elements of the two vectors. The values of max_1 and max_2 (where ϵ_r is a relative tolerance) can help us to exclude values that are too small so they are treated as zeroes. We can define a threshold for each indicator, namely μ_α, μ_β, and μ_γ, or if we do not want to distinguish them, just simply μ. If for any $\mu_x < I_x$ (where $x \in \{\alpha, \beta, \gamma\}$), then the primary test triggers

the final test:

$$trigger_x = \begin{cases} 1, & \text{if } \mu_x < I_x \\ 0, & \text{otherwise} \end{cases} \tag{4.23}$$

The question is what should be the value of μ?

Adaptivity

The value of μ depends on the numerical properties of the current LO problem. The adaptivity method originally introduced here is used in the area of digital signal processing [86]. As the final test is slow, it is not worth applying frequently. Let ζ denote the final test frequency, it can be in the range of $0.02\ldots0.05$. We follow this strategy: Set the threshold μ in a way that the rate of triggering the final test is around ζ. This strategy fine-tunes μ so that it follows the changes of the indicator variables. The triggering rate is computed with exponential averaging, using the forgetting factor α:

$$\overline{rate} := \alpha \times rate + (1 - \alpha)trigger \tag{4.24}$$

If the triggering rate is higher than ζ, our mechanism increases μ, otherwise it decreases it:

$$\overline{\mu} := \mu \times (1 + \text{signum}(rate - \zeta)\epsilon) \tag{4.25}$$

We notice that (4.25) changes the threshold here by a ratio because the numerical errors can change in a wide range, but in some other applications we can use finer steps, where the step is $\epsilon \times \text{signum}(rate - \zeta)$, and it is added to the threshold.

We emphasize that as we have more indicator variables, each indicator has its own set of adaptivity variables, like $rate, \epsilon, \zeta$, and α. This method can be more flexible if there is feedback from the final test. If the final test says that the triggering was a false alarm, it can "punish" the primary tester; the corresponding μ is increased, and the adaptivity variables change in a way that the primary tester will be less sensitive.

4.3   Test results

In the following section different LO problems are tested. The problems have the simple $\mathbf{Ax} = \mathbf{b}$ form, where the elements of $\mathbf{b}$ are the row sums of the actual $\mathbf{A}$ matrix, and the objective function is 0. In an ideal case, when there is no floating-point rounding and numerical limitations, then every element of $\mathbf{x}$ is 1. The last tests of Netlib and Kennington's problems are different, they are well-known stable real-life problems. The proposed numerical error detector algorithms are implemented in our software named Pannon Optimizer (PanOpt).

During the tests, if a software finds an optimal solution, the relative error of the

solution is calculated by the following formula:

$$\mathbf{A}\overline{\mathbf{x}} = \overline{\mathbf{b}}$$

$$error = \max_{i=1}^{m} \left\{ \frac{|b_i - \overline{b}_i|}{\max(|b_i|, |\overline{b}_i|)} \right\},$$

where $\overline{\mathbf{x}}$ is the computed solution vector.

Rounded Hilbert matrix

The traditional simplex implementations use double-precision floating-point numbers to represent the LO model. Moreover, these models are stored in text file formats, like the MPS. In these files, the numbers are written in a scientific format with a finite number of digits. Consequently, not every rational number can be stored accurately, for example, $\frac{1}{3}$. If we want to solve the classic, non-rounded Hilbert matrix, we have to change the inner structure of the LO input model representation too, but this is not the topic of this dissertation. This is why we investigated the Rounded Hilbert matrix problem introduced in (4.2.1). Table (4.4) shows the output of 5 different sizes of problems. As can be seen, each software found an optimal solution, which is, of course, not the solution of the non-rounded Hilbert matrix. For example, the nonzero basis variables of the Pannon Optimizer's result for the largest test problem are the following:

```
x1 = 0.99955     x2 = 1.02222     x3 = 0.755118    x4 = 1.9287

x7 = 2.876       x9 = 3.63773     x17 = 11.5868    x30 = 12.5929

x39 = 6.9013     x54 = 26.0923    x84 = 16.7383    x89 = 14.8661
```

Still, if we substitute these values into the input problem, and compute the $\overline{\mathbf{b}} = \mathbf{A}\mathbf{x}$ vector, and the largest relative error between the original $\mathbf{b}$ and $\overline{\mathbf{b}}$ is obtained (see last column of Table (4.4), it is clear that these solutions are really acceptable.

Rounded Pascal matrix

The P_n Pascal matrix contains the binomial coefficients. It is an n-size square matrix, where the elements are the following:

$$p^i_j = \binom{i + j}{i} = \frac{(i + j)!}{i!j!}$$

In other words,

$$p^i_j = \begin{cases} p^i_{j-1} + p^{i-1}_j, & \text{if } i > 1 \text{ and } j > 1 \\ 1, & \text{otherwise} \end{cases}$$

Test [n, software]	Iterations	Output	Time [sec]	Error
5, PanOpt	6	Optimal	0.0009	2.51033e-16
5, Glpk	5	Optimal	0.000233	1.8612e-17
5, Clp	5	Optimal	0.000462	7.44482e-17
10, PanOpt	8	Optimal	0.0025	4.36744e-09
10, Glpk	9	Optimal	0.000327	9.11289e-08
10, Clp	6	Optimal	0.000997	5.86112e-08
20, PanOpt	17	Optimal	0.0057	2.15281e-09
20, Glpk	10	Optimal	0.000659	4.23211e-08
20, Clp	8	Optimal	0.000775	4.79649e-09
50, PanOpt	23	Optimal	0.0201	2.08407e-09
50, Glpk	14	Optimal	0.004156	9.03789e-08
50, Clp	13	Optimal	0.00241	1.33875e-08
100, PanOpt	28	Optimal	0.0448	8.31182e-09
100, Glpk	19	Optimal	0.014449	2.06594e-08
100, Clp	20	Optimal	0.007494	6.90211e-09

Table 4.4: Comparison of different size of Rounded Hilbert matrix problems with 3 softwares.

For example, P_4 is the following:

$$\mathbf{P}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 3 & 6 & 10 \\ 1 & 4 & 10 & 20 \end{bmatrix}$$

It is well-known that the Pascal matrix is very ill-conditioned [1], so it can be a good test matrix for our numerical error detector. However, as n grows, the magnitude of the matrix elements grow exponentially, so after a given size, the floating-point number system that was used stores only rounded values. Formally, the elements of the Rounded Pascal matrix are the following:

$$p^i_j = \begin{cases} RN(p^i_{j-1} + p^{i-1}_j), & \text{if } i > 1 \text{ and } j > 1 \\ 1, & \text{otherwise} \end{cases}$$

We tested large ($n \geq 50$) Rounded Pascal matrices in order to check that they are not singular. The test software inverted the matrices with Gaussian elimination, using rational arithmetic. The software executed the inversions perfectly, so the test matrices showed in Table (4.5) are not singular. As can be seen, our software (Pannon Optimizer) can detect that we need advanced number representation, while the answers of the other software (singular matrix, infeasible problem) are wrong.

Rump's ill-conditioned matrix

As saw in the previous tests, the matrices were rounded, and they had very large elements. However, the later property is not reasonable in real life examples. There are ill-conditioned matrices with much smaller elements. Rump proposed a method that can produce ill-conditioned matrices, which can have smaller elements [77]. Later, he proposed a more general method in [94]. We constructed the following matrix based on the later paper:

$$\mathbf{A} = \begin{bmatrix} -1 & 99 & 99 & 99 & 101 \\ 1 & -100 & 0 & 0 & 0 \\ 0 & 1 & -100 & 0 & 0 \\ 0 & 0 & 1 & -100 & 0 \\ 0 & 0 & 0 & 1 & -100 \end{bmatrix}$$

The condition number of $\mathbf{A}$ is large: $\kappa(\mathbf{A}) = 49830891433.49631$ (computed by GNU Octave). The test software gave the following results:

Test [n, software]	Iterations	Output	Time [sec]	Error
10, PanOpt	13	<u>Optimal</u>	0.0018	2.58379e-16
10, Glpk	10	<u>Optimal</u>	0.000286	9.68922e-17
10, Clp	13	<u>Optimal</u>	0.000647	1.89848e-15
50, PanOpt	18	<u>Double precision is insufficient</u>	0.0487	
50, Glpk	18	Infeasible	0.003956	
50, Clp	20	Infeasible	0.002454	
100, PanOpt	18	<u>Double precision is insufficient</u>	0.011	
100, Glpk	18	Infeasible	0.003634	
100, Clp	20	Infeasible	0.002351	
150, PanOpt	18	<u>Double precision is insufficient</u>	0.012	
150, Glpk	0	Singular matrix	0.032122	
150, Clp	85	Infeasible	0.027324	
200, PanOpt	18	<u>Double precision is insufficient</u>	0.016	
200, Glpk	0	Singular matrix	0.043781	
200, Clp	135	Infeasible	0.05328	

Table 4.5: Comparison of different size of Rounded Pascal matrix problems with 3 softwares. The right outputs are underlined.

Configuration	Netlib time	Netlib iterations	Kennington time	Kennington iterations
A	224.48 sec	252614	2086.75 sec	502802
B	234.71 sec	245763	2321.67 sec	538561
C	231.4 sec	245877	2245.03 sec	523667
D	228.69 sec	252707	2119.63 sec	507557

Table 4.6: Summarized execution times and iteration numbers in different test configurations.

- Pannon Optimizer: A numerical error was detected by the final test after 4 iterations.

- Clp, Glpk: The produced solution vector is the following: $x_1 = 0, x_2 \approx 0.99, x_3 \approx 0.9999, x_4 \approx 0.999999, x_5 \approx 1$. The values are similar, but there is a little difference between the two software's output, so the relative error of the Clp is ≈ 1.33, and the error of Glpk is ≈ 1.04.

That is, the Clp and Glpk produced a significantly wrong solution, while the PanOpt detected that we need advanced number representation.

Netlib and Kennington problems

The Netlib and Kennington LO problems are tested with the proposed detector algorithm. Obviously, they are numerically stable problems, so the expected output is that they can be solved using simple double floating-point numbers. Our tests were successful, we have obtained the optimal solution for every problem. It is clear that the new test steps can slow down the software, so the running times were tested. There are four test configurations:

- A: The detector algorithm is switched off.

- B: The primary test is executed with a probability of 0.05.

- C: The primary test is executed with a probability of 0.05, and if the final test detects a false positive triggering, this probability is multiplied by 0.7.

- D: The primary test is executed with a probability of 0.01, and if the final test detects a false positive triggering, this probability is multiplied by 0.7.

We summarized the total execution times and iterations per configurations, the results can be seen in Table (4.6).

The Appendix (B) contains the detailed results for the test problems. It can be seen that there are a few problems, where the checking algorithm can *decrease* the iteration number (and the execution time, too): If the primary test triggers a reinversion, we obtain a more accurate basis inverse representation, so the software can navigate back to the correct direction. For example, QAP8 is solved in under 26.85 seconds in Configuration-B, instead of 32.91 seconds. These test results are highlighted with a green background.

4.4   Major results and summary of accomplishments

We have developed an adaptive heuristic algorithm embedded in the simplex method that monitors the numerical stability of the basis.

- In Section 4.1 we proved for a general matrix norm that it is not enough to calculate the condition number of the coefficient matrix during preprocessing, as it can be low, while the matrix can have a submatrix with extremely high condition number.

- A two-phase testing algorithm has been incorporated into the simplex method: The first phase is a quick test, and if its output exceeds a threshold, a slower but reliable test is run. The method works adaptively, tests run less frequently for stable problems and more often for unstable ones.

Related publication

- J. Smidla and G. Simon. "Accelerometer-based event detector for low-power applications". In: *Sensors* 13.10 (2013), pp. 13978–13997

Related conference presentations

- J. Smidla, P. Tar, and I. Maros. "Adaptive Stable Additive Methods for Linear Algebraic Calculations". In: 20th Conference of the International Federation of Operational Research Societies (Barcelona, Spain). 2014

- M. I. Smidla József. "Numerikusan szélsőségesen instabil feladatok megoldása a szimplex módszerrel". In: XXXI. Magyar Operációkutatási Konferencia (Cegléd, Magyarország). 2015

- J. Smidla, P. Tar, and I. Maros. "A numerically adaptive implementation of the simplex method". In: VOCAL (Veszprém, Hungary). 2014

Chapter 5

Low-level optimizations

5.1   Intel's SIMD architecture

The *SIMD* (Single Instruction, Multiple Data) architectures provide a powerful tool to perform the identical low-level operations on multiple data in parallel [31]. It was successfully used in the simplex method [98], and in other numerical algorithms [95]. The older Intel CPUs (more precisely, their FPU, i.e. Floating-point Unit) used a stack to store 32, 64 or 80-bit floating-point numbers. This architecture can perform one operation only on a single data. In 1999 Intel introduced the *SSE* (Streaming SIMD Extensions) instruction set in the Pentium III processor. It contained 70 new instructions, which can operate on multiple (namely four) single-precision numbers. The processor had 8 brand new, 128 bit wide registers, their names were XMM0, XMM1, ..., XMM7. One XMM register can store 4 single-precision (32-bit) numbers. The arguments of the operations are the registers (or memory), and the result will be stored in the other register (or memory). For instance, the

```
ADDPS xmm0, xmm1
```

instruction adds the content of register XMM1 to XMM0, that is, the CPU adds the first number in XMM0 to the first number of XMM1, and so on, as Figure (5.1) shows.
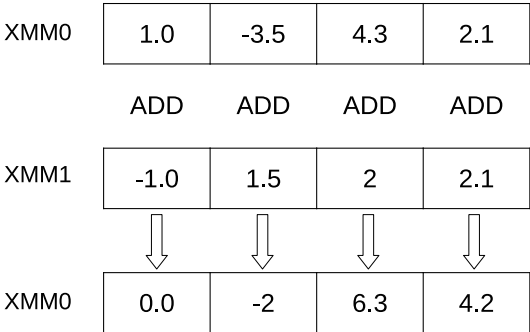
Figure 5.1: Addition using XMM registers

The main area of use of SSE is digital signal processing and graphics processing because the single-precision format is enough. However, because of the fact that SSE does not support 64-bit floating-point operations, it was not very useful for engineering and scientific purposes. Intel's answer to this problem was the *SSE2* which supports 64-bit numbers as well; it was introduced by the Pentium 4. The XMM registers remained 128 bit wide, so one register can contain two double-precision floating-point numbers (and of course, four single-precision numbers). The later *SSE3* and *SSE4* instruction sets added a few expansions to the earlier instruction sets, like dot product and integer number support, but they are irrelevant in this dissertation.

In 2011, Intel extended the SIMD with the *AVX* (Advanced Vector Extensions) instruction set. This extension doubles the size of the XMM registers to 256-bit wide. The new, wider registers are called YMM. The lower 128-bit range of these registers are the old XMM registers. Now one register can store four 64 bit floating-point numbers, or eight 32-bit numbers. Moreover, AVX's have 16 YMM registers. The AVX2 instruction set provides some integer operations with the new registers, and *FMA* (Fused Multiply Add) operations on floating-point numbers.

Intel introduced AVX-512 in July 2013, which extended the SIMD registers to 512-bit length. The new registers were the ZMM registers, from ZMM0 to ZMM31. The AVX-512 included several instruction sets, and different CPUs support different sets. Here are some examples (including but not limited to):

- AVX-512F

- AVX-512DQ

The modern Intel CPUs have one more useful feature; they have two memory ports. It means that, while the CPU calculates, they can load other data from the memory in parallel.

Cache

In this section, a brief summary of the CPU caching will be given because it has some non-intuitive properties: The inappropriate use of the cache cannot achieve the highest performance. The communication between the main memory and the CPU is much slower than the CPU's speed, i.e., while the CPU is waiting for the memory, it can execute many (up to a hundred) instructions. To keep the CPU working, engineers injected a *cache* memory between the CPU and the main memory. The cache uses faster electric-circuit elements, but it is more expensive, so the cache size is limited relative to the main memory. Typical cache sizes are 3-10 MBytes today, while the main memory can be 32 GBytes on desktop computers. Moreover, modern CPUs have more cache levels, where the lower levels are smaller, but they are faster.

The memory is divided into so-called *cache lines*, which are equal lengths of memory partitions. 32 or 64 bytes are typical lengths. If the CPU reads certain bytes from a memory address, the total cache lines that contain the required data moves to the cache. Notice that if the required data overlap two cache lines, then those cache lines will be

loaded. If later instructions need an adjacent memory address its content will already be in the faster cache, thus the reading time is reduced. However, as the cache size is limited, the CPU has to make room for a new cache line if the required data item is not in the cache. In this case, a formerly used cache line is dropped out and its content is written back to the main memory if it is necessary. When the CPU writes to an address, the corresponding cache line will be loaded into the cache, and the instruction will be written there. In this case, the content of that memory address has a copy in the cache, which is different; this cache line is considered to be *dirty*, but if we write this content to the memory, this flag is cleared.

There is little intelligence in the cache controller. If the CPU senses that the software accesses adjacent memory addresses, then it automatically loads certain next cache lines, if it is possible. So, if we read or write a memory region from its beginning to its end, the data that is currently needed will already be in the cache with a high probability.

The SSE2 and AVX instructions support bypassing the cache for memory writing; the cache line of the current memory address is not loaded and the CPU writes into the main memory directly. This is the so-called *non-temporal writing*. Obviously, this mode is much slower if we need a small amount of data. However, we can keep the more important (i.e. the frequently used) data in the cache. What happens if we add two large vectors (larger than the cache), and the result is stored in a third vector? Without non-temporal writing, the CPU reads the next two items of the sum and it writes the result to the memory. In the first step, the result is placed into the cache. But, since the vectors are too large, and their contents fill the cache, the CPU has to drop out an older cache line to the memory. If the cache line of the result is not prepared for the cache, the CPU has to load that cache line and, obviously, drops out an older line too. The bypassing prevents the CPU from loading the cache-line of the destination, so it drops out older cache lines if and only if there is no more room for the input data. Finally, the performance of this algorithm is improved.

5.2   Vector addition

Vector addition is one of the most frequently used operations in computational linear algebra as many optimizations algorithms rely on it.

Let $\mathbf{a}$ and $\mathbf{b}$ be two n dimensional vectors, $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$. We propose our implementations of the following vector addition operation:

$$\mathbf{a} := \mathbf{a} + \lambda\mathbf{b},$$

where $\lambda \in \mathbb{R}$. Its naive pseudo-code:

If a numerical error occurs, the approach shown in Algorithm (5.1) can generate fake non zeros. This error can be handled with an absolute tolerance ϵ_a. If the absolute value of the sum is smaller than the ϵ_a, we set the result to zero as in Algorithm (5.2).

The absolute tolerance cannot take into consideration the magnitudes of the input values. The solution can be the relative tolerance ϵ_r. If the sum is much smaller compared

---

Algorithm 5.1 Naive vector addition

Input: a, b, λ
Output: a
  1: **for** $i := 1$ to n
  2:      $a_i := a_i + \lambda b_i$
  3: **end**

Algorithm 5.2 Vector addition using absolute tolerance

Input: a, b, λ
Output: a
  1: **for** $i := 1$ to n
  2:      $a_i := a_i + \lambda b_i$
  3:      **if** $|a_i| < \epsilon_a$ **then**
  4:          $a_i := 0$
  5:      **end**
  6: **end**

---

to the largest absolute value of the input numbers the result is set to zero: In 1968 William Orchard-Hays [72] suggested the following method using this tolerance, see Algorithm (5.3).

Algorithm 5.3 Vector addition using relative tolerance, Orchard-Hays's method

---

Input: a, b, λ
Output: a
  1: **for** $i := 1$ to n
  2:      $c := a_i + \lambda b_i$
  3:      **if** $\max\{|a_i|, |\lambda b_i|\}\epsilon_r \geq |c|$ **then**
  4:          $c := 0$
  5:      **end**
  6:      $a_i := c$
  7: **end**

Determining the maximum of two numbers uses a conditional branching instruction. We propose a simplified method that uses fewer operations and branching. It is sufficient to multiply the absolute value of one of the input numbers by the relative tolerance. By doing this we can save an absolute value and a conditional branching step. The result is close to zero if the operands have the same order of magnitude and their signs are the inverses.

The implementation shown in Algorithm (5.2) requires one comparison and a conditional jump instruction. The simplified method with relative tolerance uses one addition, two multiplications, two assignments, two absolute values, one comparison, and one conditional jump. Orchard-Hays's implementation needs one more absolute value and conditional branching. The additional operations cause overhead in time, and the

---

Algorithm 5.4 Vector addition using simplified relative tolerance

---

Input: a, b, λ
Output: a
1: **for** $i := 1$ to n
2:     $c := a_i + \lambda b_i$
3:       **if** $|a_i|\epsilon_r \geq |c|$ **then**
4:             $c := 0$
5:       **end**
6:       $a_i := c$
7: **end**

---

more conditional branching breaks the pipeline mechanism, so these implementations are slower than the naive one.

The simplified and Orchard-Hays's method give the same output if $|a_i| = |\lambda b_i|$, or $\max\{|a_i|, |\lambda b_i|\} = |a_i|$. There is difference if $\max\{|a_i|, |\lambda b_i|\} = |\lambda b_i|$, and $a_i(\lambda b_i) < 0$.

Figure (5.2) shows four examples. Subfigures (5.2a) and (5.2b) represent two examples for the functioning of Orchard-Hays's method. In these cases, the magnitude of a is greater than the magnitude of b. In this case $|a|\epsilon$ is chosen by both methods in the comparison. However, in subfigures (5.2c) and (5.2d) $|b|$ is greater than $|a|$, so we can observe the difference between the two methods: There is a small range between $|a|\epsilon$ and $|b|\epsilon$, where the simplified method will not move the result to zero unlike the Orchard-Hays's method. Namely, the output of the two methods is different if

$$|a|\epsilon < |ab| \leq |b|\epsilon.$$

Note that not only can the two results differ if a and b are close to zero.
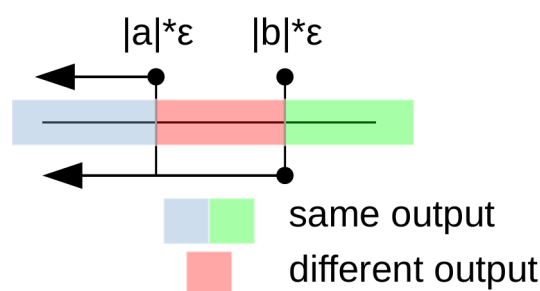
Figure 5.3: The red area is the small range where the two methods give a different result. The Orchard-Hays's method goes to zero in the blue and red ranges, while the simplified method moves to zero only in the blue area.

(a) Orchard-Hays's method if $b < 0$ and $a > 0$.

(b) Orchard-Hays's method if $b > 0$ and $a > 0$.

(c) The simplified method if $a < 0$, $b > 0$, and $|a| < b$.

(d) The simplified method if $a < 0$, $b > 0$, and $|a| < b$.
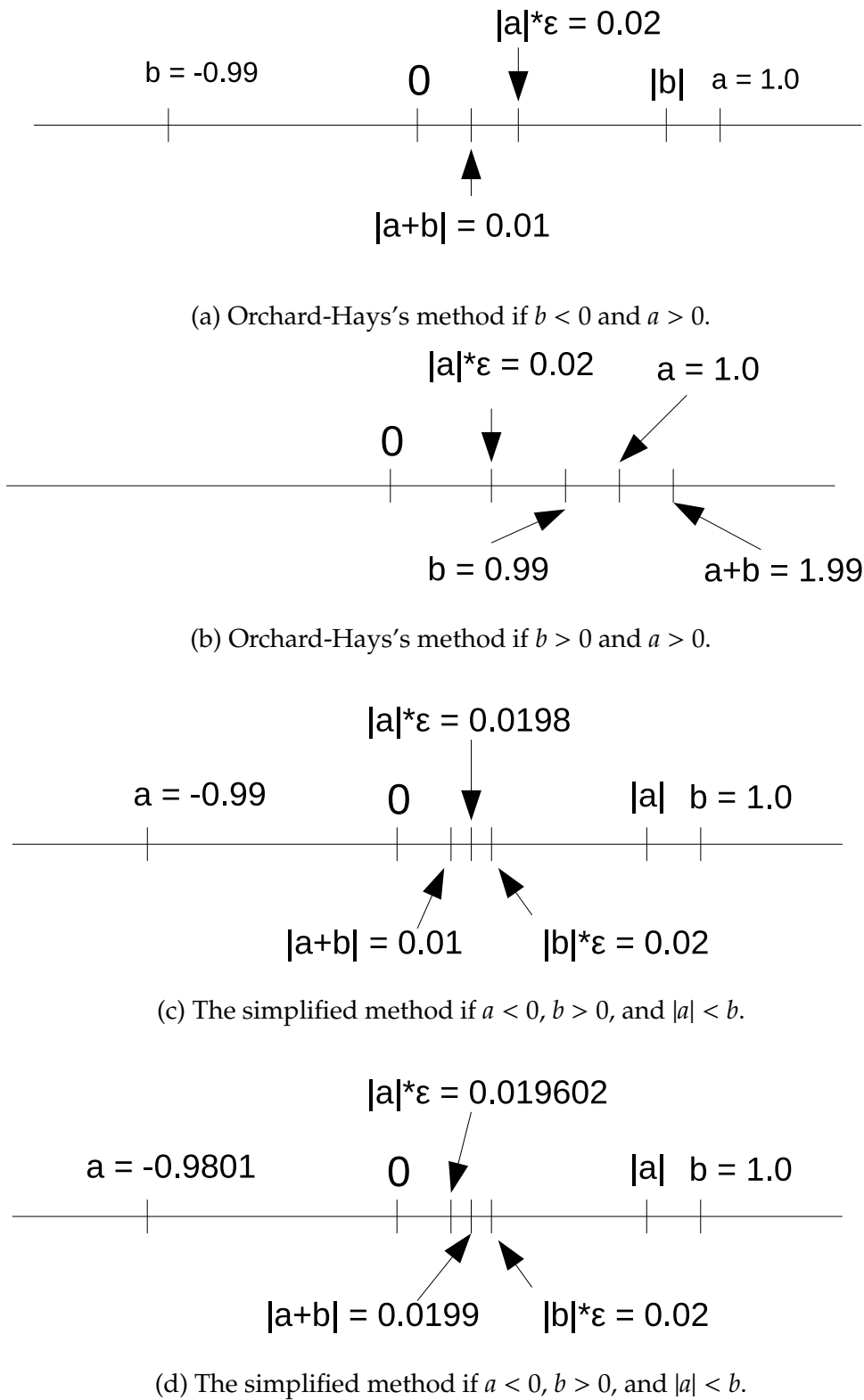
Figure 5.2: Comparison of Orchard-Hays's and the simplified addition algorithms. For the sake of simplicity, $\lambda = 1$ and $\epsilon = 0.02$.

5.2.1 SIMD vector addition

As conditional jumping breaks the pipeline mechanism, it slows down the execution of the program. It is useful to implement the algorithms in a way that is free from conditional jumping. Intel's SIMD architecture contains several instructions that help us design such an implementation. We will use the following atomic instructions:

- *Move:* Moves the content of a register to another register.

- *Multiply:* Multiplies the number pairs of two registers.

- *Add:* Adds the number pairs of two registers.

- *And:* Performs a bitwise AND between two registers.

- *Compare:* Compares the number pairs of two registers. If they are identical, the destination register will contain a bit pattern filled by 1's (the result is NaN), otherwise 0.

- *Max:* Chooses the larger of two numbers stored in two registers. It is used for the implementation of Orchard-Hays's addition method.

The detailed description of these instructions can be found in Intel's instruction set documentation [37]. The key point of the conditional jump-free implementations (called accelerated stable addition in this paper) is the compare instruction. It compares the numbers and stores the results in a register. If the register contains two double pairs then the comparator places two bit patterns to the destination area. One pattern can be filled by 1 if the result of the comparison is true, otherwise 0 as it is shown in Figure (5.4). These bit patterns can be used for masking.
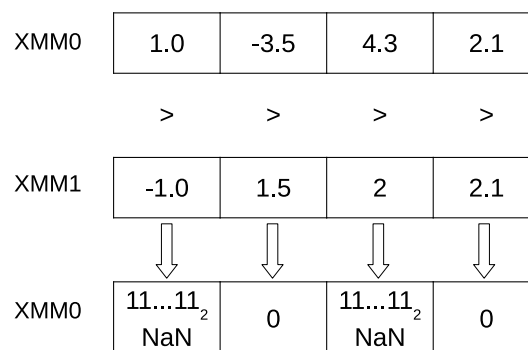
Figure 5.4: The compare instruction of the SSE2 instruction set

In this section, we will refer to the XMM registers only for the easier discussion. But we notice that the AVX and AVX-512 implementations use YMM and ZMM registers respectively. Moreover, at first, we describe the algorithm as if we would implement it in assembly language. However, as we will see in (5.4) it is not the most clever way, it is much better if we let a modern C++ compiler to generate the assembly level code, and this code likely will use the registers in the other way.

Figures (5.5) and (5.6) show the flowchart of the SSE2 versions of our stable add operations with relative and absolute tolerances. The algorithms add two number pairs loaded to registers XMM0 and XMM1. The final result goes into XMM2.

The implementations have two major phases: initialization, and processing. We prepare certain registers to store the constant value of λ (XMM7), ϵ_r (XMM4), ϵ_a (XMM6) and the absolute value mask (XMM5). In the process part we perform the stable add operations for the successive number pairs, without modifying registers XMM4-XMM7. Figures (5.5) and (5.6) show only one iteration in the processing phase. One iteration of the absolute tolerance stable adder performs these 6 steps:

1. Multiply XMM1 and XMM7, store the result in XMM1, XMM1 will store λb_i.

2. Add XMM1 to XMM0, so XMM0 stores $c = a_i + \lambda b_i$.

3. Move XMM0 to XMM2. We have to store the original value of c, in order to use its absolute value in later steps.

4. Bitwise AND between XMM2 and XMM5, store the result in XMM2. Therefore XMM2 stores $|c|$.

5. Now we have to compare $|c|$ and ϵ_a. If $|c| < \epsilon_a$, then the CPU sets the bits of the corresponding floating point number in XMM2, otherwise clears them.

6. Bitwise AND between XMM2 and XMM0. After this step, if $|c| < \epsilon_a$ then XMM2 stores zero, because of the cleared bit mask in XMM0, otherwise XMM2 stores c.

The stable add operation that uses relative tolerance performs the following 9 steps in one iteration:

1. Multiply XMM1 and XMM7, store the result in XMM1, XMM1 will store λb_i.

2. Move XMM0 to XMM2. We have to store the original value of a_i and λb_i, in order to use their absolute value in the later steps.

3. Add XMM1 to XMM2, so XMM1 stores $c = a_i + \lambda b_i$.

4. Move XMM2 to XMM3, because we will use the absolute value of c in the next steps, but we will need the original value of c as well.

5. Bitwise AND between XMM3 and XMM5, store the result in XMM3. Therefore XMM3 stores $|c|$.

6. Bitwise AND between XMM0 and XMM5, XMM0 stores $|a_i|$.

7. Multiply XMM0 and XMM4, and store the result in XMM0, so XMM0 stores $|a_i|\epsilon_r$.

8. Now we have to compare $|a_i|\epsilon_r$ and $|c|$. If $|a_i|\epsilon_r < |c|$, then the CPU sets the bits of the corresponding floating point number in XMM0, otherwise clears them.

9. Bitwise AND between XMM2 and XMM0. After this step, if $|a_i|\epsilon_r \geq |c|$ then XMM2 stores zero, because of the cleared bit mask in XMM0.

The operations above belong to exactly one SSE2 or AVX instruction, so the reader can easily reproduce our results. These implementations use several additional operations on top of the one addition and multiplication, so they have an overhead compared to the naive implementation. They use a few additional bit masking steps because Intel's SIMD instruction sets (except the AVX-512) have no absolute value operations. However, we can obtain the absolute value of a floating-point number by clearing the sign bit. Therefore, we have to apply the bit masking technique to obtain the absolute values, as in the steps 5-7, in relative tolerance adder, and step 4 in absolute tolerance adder.

However, SSE2 performs every instruction between two number pairs in parallel, so this overhead is not significant. Moreover, AVX can execute the instructions between 4 number pairs (and AVX-512 doubles this amount). Consequently, the overhead will be even lower. In order to improve the speed of the algorithms, our implementations utilize the two memory ports mentioned in Section (5.1): While one number pair is being processed, the next pair is loaded to other unused registers, so the delay of memory access is decreased. We use this technique in our dot-product implementations.

We modified the above described relative tolerance adder procedure to implement Orchard-Hays's method. Two additional steps are inserted after step 6:

1. Bitwise AND between XMM1 and XMM5, XMM1 stores $|\lambda b_i|$.

2. Use MAX operation between XMM0 and XMM1, XMM0 stores $\max\{|a_i|, |\lambda b_i|\}$.

5.3   Vector dot-product

The dot-product between two n dimensional vectors $\mathbf{a}$ and $\mathbf{b}$ is defined as:

$$\mathbf{a}^T \mathbf{b} = \sum_{i=1}^{n} a_i b_i.$$

Algorithm (5.5) shows the pseudo-code of its naive implementation. The problem is that the add operation in line 3 can cause a cancellation error if the operands have different signs.

Algorithm 5.5 Naive dot-product

Input: $\mathbf{a}, \mathbf{b}$
Output: dp
  1: $dp := 0$
  2: **for** $i := 1$ to n
  3:     $dp := dp + a_i b_i$
  4: **end**
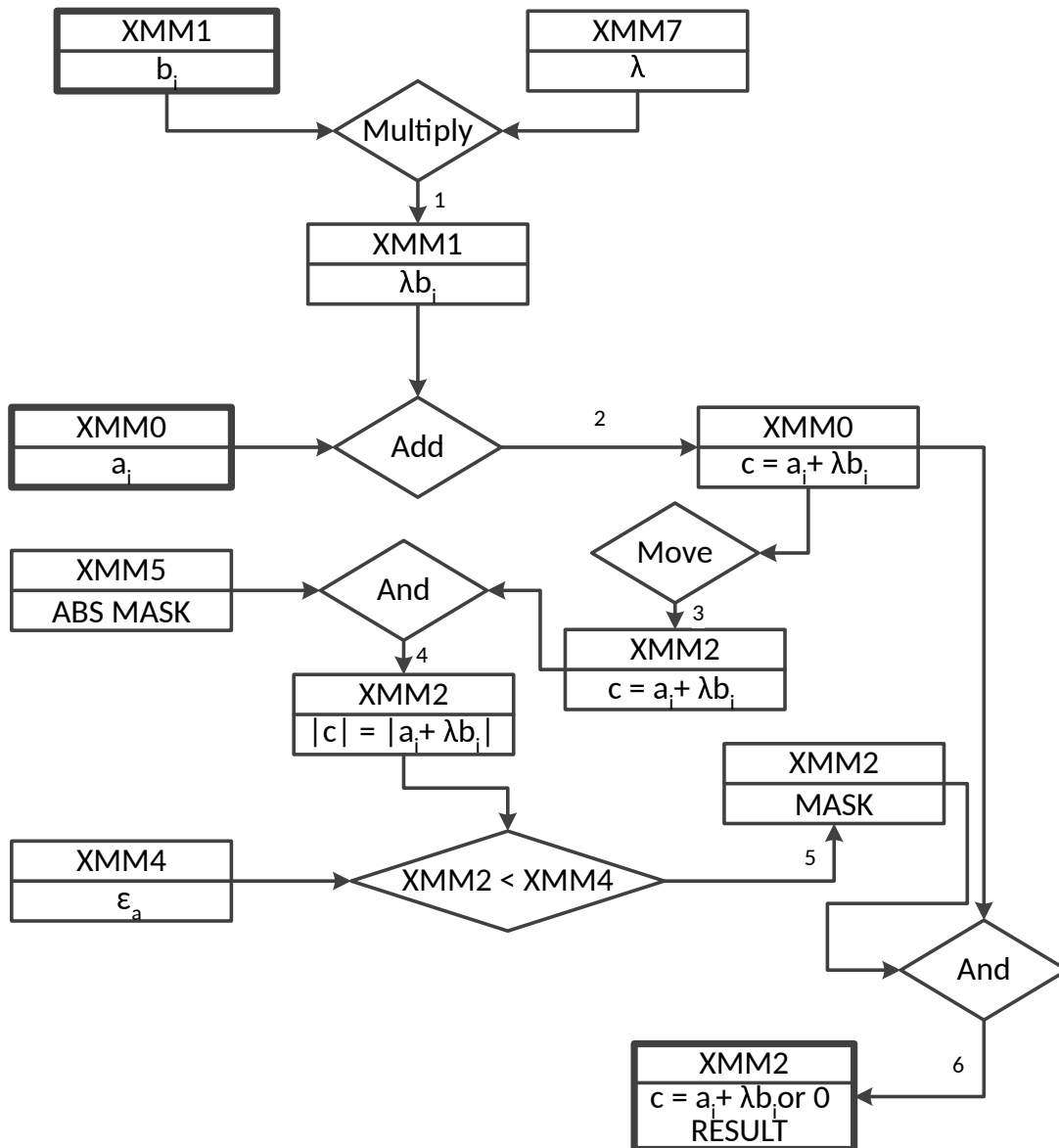
Figure 5.5: Flow chart of the stable add implementation, using absolute tolerance. Arrow numbers show the order of the operations.

This error can be greatly reduced by using a *pos* and a *neg* auxiliary variables as introduced by Maros and Mészáros in 1995 [62]. Positive (negative) products accumulate in variable *pos* (*neg*). Finally, the result is the sum of *pos* and *neg* as Algorithm (5.6) shows. The final add is a stable add operation introduced in Section (5.2). Notice that Kulisch published his idea of long accumulator hardware in [52] and [53], but CPU vendors had not implemented it.

The conditional jump of line 5 breaks the CPU pipelining and the execution slows down accordingly. We developed a solution for C/C++ programs, where we avoid the conditional jump and substitute it with pointer arithmetic. This method can be used if the later introduced SIMD based masking methods are not available, for example, the AVX is disabled by the operating system, or the target platform is a cheaper microcontroller. The
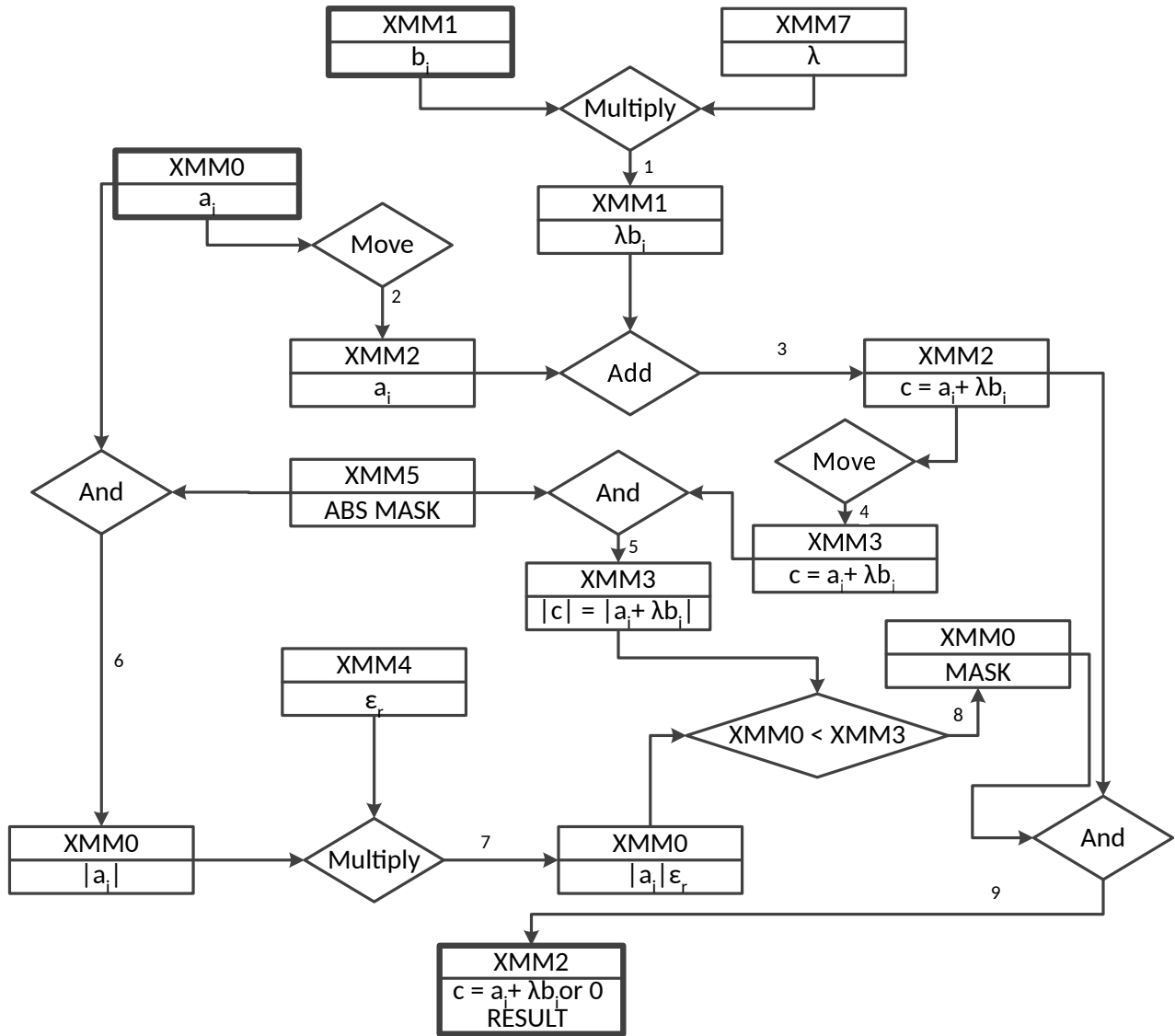
Figure 5.6: Flow chart of the stable add implementation, using relative tolerance. Arrow numbers show the order of the operations.

array elements are stored on continuous memory addresses. If a pointer is increased by 1 in C/C++, it will point to the next object. The most significant bit in the bit pattern of a double type variable is the sign bit. If this bit is 1, the number is negative, otherwise, it is positive. The conditional jump free implementation uses a double array, where the first element stores the positive, the second one stores the negative sums. The current product is added to one of these elements. The address of the current sum variable is selected by a C/C++ expression: The address of the array is shifted by the product's sign bit, as Figure (5.7) shows.

SIMD dot-product

The SIMD dot product uses similar techniques introduced in Section (5.2.1). This implementation also includes two phases, initialization and processing. We use XMM1 for the

---

Algorithm 5.6 Stable dot-product, where *StableAdd* is an implementation of the addition, which can use tolerances

---

Input: a, b
Output: *dp*
 1: $dp := 0$
 2: $pos := 0$
 3: $neg := 0$
 4: **for** $i := 1$ to n
 5:       **if** $a_i b_i < 0$ **then**
 6:              $neg := neg + a_i b_i$
 7:       **else**
 8:              $pos := pos + a_i b_i$
 9:       **end**
10: **end**
11: $dp := StableAdd(pos, neg)$

---

negative products, while XMM2 stores the positive products, and XMM4 contains zero for the comparison.

In the first step, the product is moved into XMM0, the multiplication is supported by SSE2. The separation of positive and negative products can be implemented in 7 steps:

1. In order to keep the value of the product $a_i b_i$, save the content of XMM0 to XMM5.

2. Move the content of XMM0 to XMM3, in order to perform the comparison between zero and the product.

3. Compare XMM3 with XMM4, if $a_i b_i < 0$, then the CPU sets the bits of the corresponding floating point number in XMM3, otherwise clears them.

4. Bitwise AND between XMM5 and XMM3. If $a_i b_i < 0$, then XMM5 stores $a_i b_i$, otherwise zero.

5. Add XMM5 to XMM1, i.e if $a_i b_i < 0$, then we add this negative value to XMM1, otherwise we add zero.

6. Bitwise AND between the inverse of XMM3 and XMM0. If $a_i b_i \geq 0$, then XMM3 stores $a_i b_i$, otherwise zero.

7. Add the content of XMM3 to XMM2, that is, we update the positive sum.

Similarly to the SIMD accelerated vector addition, this dot product algorithm can be improved using AVX and AVX-512. The stable dot product uses fewer instructions than the stable add, so the performance of this implementation is better, as we will see in Section (5.5).
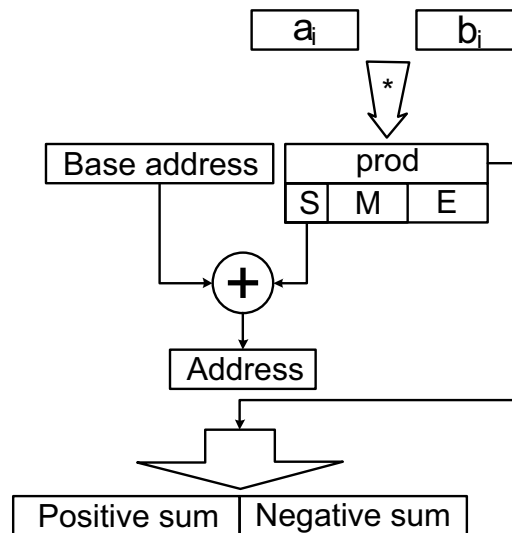
```
 1  union Number {
 2      double num;
 3      unsigned long long int bits;
 4  } number;
 5
 6  double negpos[2] = {0.0, 0.0};
 7  [...]
 8  const double prod = a * b;
 9  number.num = prod;
10  *(negpos + (number.bits >> 63)) += prod;
```

Figure 5.7: Handling positive and negative sums with pointer arithmetic without branching, where S is the sign bit, M is the significand and E is the exponent

5.4   Implementation details

The obvious way to implement these algorithms is the use of an assembler. The elementary SIMD operations introduced in this chapter are equivalent to specific machine code instructions. However, this method has numerous disadvantages:

- Performance: Although until the 90s, writing software in assembly was a good idea to optimize the speed and size, it is not a state of the art programming today. The modern compilers produce much better machine code than most programmers can write in assembly. Today's CPUs are much more complex than decades ago. They use caching, branch predictors, instruction pipelines, and so on. Some instructions can help or pull back these features. The programmers have to have very strong and specific knowledge in order to write better code than a compiler can produce.

- Portability: There are major differences between Intel's 32-bit and 64-bit architectures. The operations can have different arguments, and moreover, there are new
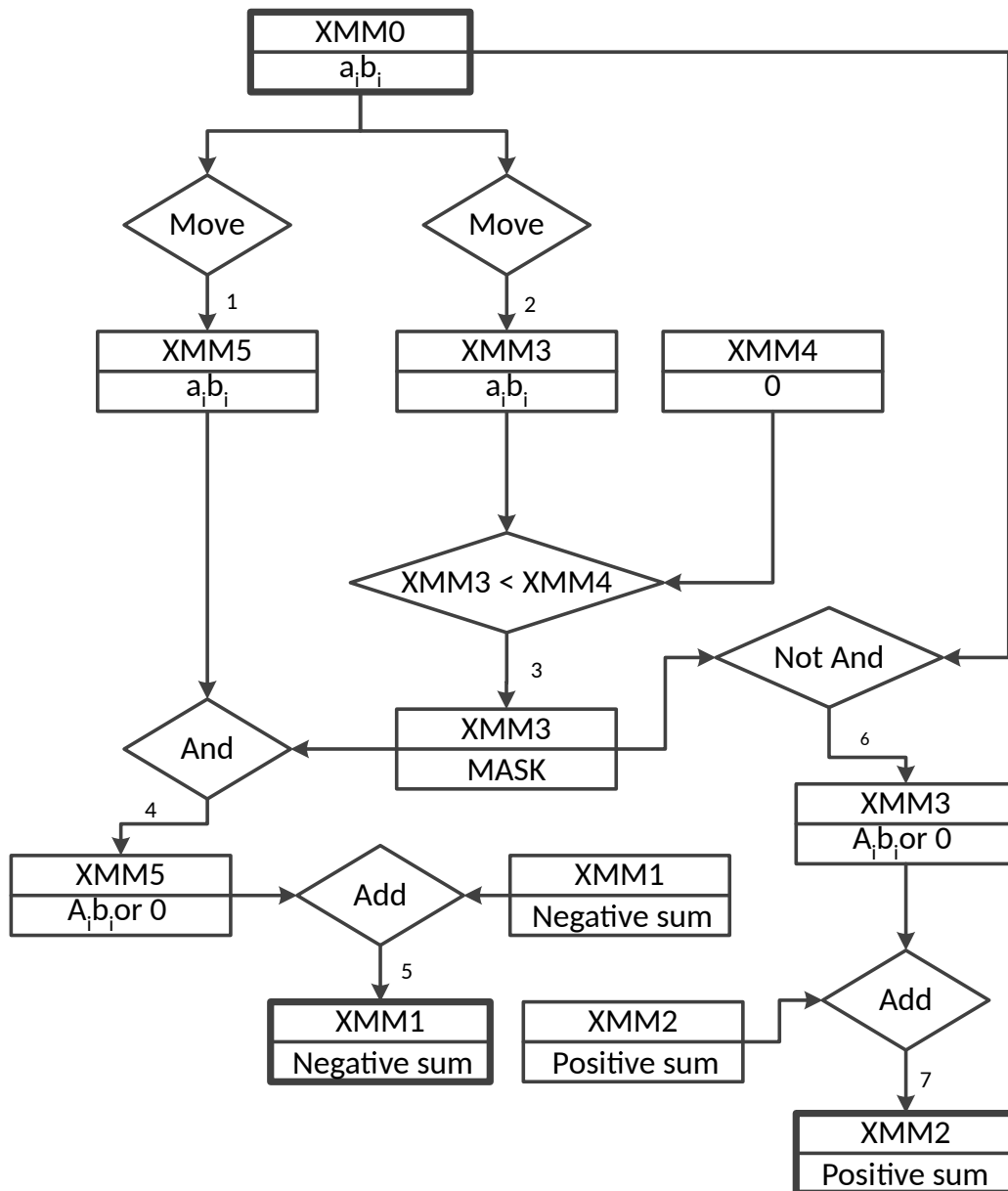
Figure 5.8: Flow chart of the stable dot product implementation. Arrow numbers show the order of the operations.

instructions in 64-bit mode. For example, the X86 CPUs do not support integer operations with 64-bit integers in 32-bit mode (this feature is realized by the compilers or compiler libraries). If we would like to implement a 32-bit assembly code to 64 bit, we have to rewrite it with different instruction codes. However, the 64-bit world is more complicated: There are different function conventions between Unix/Unix-like and Windows systems. For example, under Windows, the first four integer arguments are passed in registers RCX, RDX, R8, and R9. System V AMD64 ABI is used by some Unix and Unix-like systems: Linux, Solaris, FreeBSD, and macOS. Here, the first six integers are passed in these registers: RDI, RSI, RDX, RCX, R8, R9. It is clear that if we would like to implement our software to Windows and Linux,

we need to implement 64-bit codes twice.

- Maintainability: Assembly codes are very hard to read, implement and modify. The higher-level languages, like C++ support clean code conventions [63], so it is much easier to write a code that is easy to read, and it can be the documentation of itself. Moreover, the probability of writing flawless code in C++ is easier than with assembly.

The compilers can utilize the SIMD instructions, but sometimes we have to give a bit of help , especially, when our algorithm is complex. Intel's solution to this problem is the intrinsic instructions. These special compiler instructions fit the higher-level source code and suggest the compiler how it should generate the machine code. Figure (5.9) shows a simple sample code. The special data type `__m128d` stores 2 double type variables, and it can mean that `dataA1`, `dataB1`, and other similar variables have to move into the SIMD registers. The `_mm_load_pd` loads data from the memory to these variables, and this time, to the registers. Obviously, `_mm_add_pd` and `_mm_mul_pd` adds and multiplies numbers between these registers. There are other instructions also, we can use bit operations, comparisons, and more.

There is one more implementation problem: Our SIMD implementations have two versions, one uses the cache to store the result, one not. The only difference between the two versions is one instruction: The non-cache versions use movnt instructions instead of the simple data moving operations. It is clear that in the clean code principles writing of these functions twice (with one small difference) is contraindicated. The C++ language gives certain tools to solve this problem. Somehow, we have to pass the storing method to our function, without loss of speed, so the branching is inappropriate. The traditional way is using a function pointer or function object. However, our measurements show performance loss with these methods. The problem is that the compiler has to generate a machine code that somehow takes into account the current storing method in one single function. The template system of C++ can help us: If the storing method is the template argument, then the compiler has to generate different versions of our function, and one function can be simpler (and faster). In our solution, we used special local defined structs with inline static functions as template parameters. For example, in Figure (5.10) the `denseToDenseAddSSE2_cache` function creates the `CACHE_WRITER` struct with the write function inside. This function implements the simple store method using cache. This struct is passed as template argument to the `denseToDenseAddSSE_temp` template function, and this function calls the write method in code lines 15, 21 and 30 of Figure (5.9). Additional sample codes can be found in Appendix E.

For performance reasons, in Figures (5.9) and (5.10) the array pointers of the function arguments have the `__restrict__` modifier. This modifier tells the compiler that the two arrays do not overlap each other, so the compiler can generate a machine code that is a little bit faster.

The SSE2, AVX, and AVX-512 implementations have their special hardware requirements. If the current CPU can support only SSE2 instructions, the software will not use the AVX functions. However, these C++ instructions needs some GCC compiler options: -msse2, -mavx, -mavx512f -mavx512dq. But if we apply these options on every function,

```
1  template <class WRITER>
2  void denseToDenseAddSSE2_temp(const double * __restrict__ a,
3                                const double * __restrict__ b,
4                                double * c,
5                                size_t count, double lambda) {
6      const size_t rem1 = count % 4;
7      size_t i;
8      __m128d mul = _mm_set_pd1(lambda);
9      for (i = 0; i < count - rem1; i += 4) {
10         // c = a + b * lambda
11         __m128d dataA1 = _mm_load_pd(&a[i]);
12         __m128d dataB1 = _mm_load_pd(&b[i]);
13         __m128d dataC1 = _mm_add_pd(dataA1,
14                 _mm_mul_pd(dataB1, mul));
15         WRITER::write(c + i, dataC1);
16
17         __m128d dataA2 = _mm_load_pd(&a[i + 2]);
18         __m128d dataB2 = _mm_load_pd(&b[i + 2]);
19         __m128d dataC2 = _mm_add_pd(dataA2,
20                 _mm_mul_pd(dataB2, mul));
21         WRITER::write(c + i + 2, dataC2);
22
23     }
24     const size_t secondCount = rem1 / 2;
25     if (secondCount) {
26         __m128d dataA = _mm_load_pd(&a[i]);
27         __m128d dataB = _mm_load_pd(&b[i]);
28         __m128d dataC = _mm_add_pd(dataA,
29                 _mm_mul_pd(dataB, mul));
30         WRITER::write(c + i, dataC);
31         i += 2;
32     }
33     for (; i < count; i++) {
34         c[i] = a[i] + b[i] * lambda;
35     }
36 }
```

Figure 5.9: The source code of the naive SSE2 version vector-vector add template function.

```
1  extern "C" void denseToDenseAddSSE2_cache(
2      const double * __restrict__ a,
3      const double * __restrict__ b,
4      double * c,
5      size_t count, double lambda) {
6      struct CACHE_WRITER {
7          inline static void write(double * address,
8                                   __m128d & value) {
9              _mm_store_pd(address, value);
10         }
11     };
12     denseToDenseAddSSE2_temp<CACHE_WRITER>(a, b, c,
13                                          count, lambda);
14 }
15
16 extern "C" void denseToDenseAddSSE2_nocache(
17     const double * __restrict__ a,
18     const double * __restrict__ b,
19     double * c,
20     size_t count, double lambda) {
21     struct NOCACHE_WRITER {
22         inline static void write(double * address,
23                                  __m128d & value) {
24             _mm_stream_pd(address, value);
25         }
26     };
27     denseToDenseAddSSE2_temp<NOCACHE_WRITER>(a, b, c,
28                                          count, lambda);
29 }
```

Figure 5.10: The source code of the naive SSE2 version vector-vector
add functions, with caching and non-caching versions.

the compiler will use AVX512 instructions in our SSE2 and AVX implementations, because it will be much faster. In order to avoid this, we have to put the SSE2, AVX and AVX-512 codes into three different libraries, and we have to build these libraries with different compiler options.

5.5 Computational experiments

In this section a few benchmarking results are presented. At first the tests were performed on a computer with the following parameters:

- CPU: Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz

- Level 1 cache: 32 KByte

- Level 2 cache: 256 KByte

- Level 3 cache: 4096 KByte

- Memory: 16 GByte

- Operating system: Debian 10, 64 bit

- Window manager: IceWM

The i7-2640M CPU has three cache levels. Intel processors have an inclusive cache architecture: the higher level caches contain the lower levels, so the test CPU has a 4 MByte cache in total. Moreover, this CPU has two cores, where the L1 and L2 caches are unique in each core. However, the cores share the L3 cache, so it can happen that more than one process uses the L3 cache [38].

In order to test the AVX-512 implementations, an Intel Xeon-based server computer was used:

- CPU: Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz

- Level 1 cache: 32 KByte

- Level 2 cache: 1024 KByte

- Level 3 cache: 33792 KByte

- Memory: 350 GByte

- Operating system: CentOS 7.6.1810 (Core)

We have performed 120 measurements with different sizes of test vectors. In the sequel, s_i ($0 \leq i < 120$) denotes the size of one vector in the ith test. In the first test, the size of a vector is 1000 elements ($s_0 = 1000$). The vector sizes grew exponentially: $s_i = \lfloor 1000 * 1.1^i \rfloor$, thus the largest vector size is 84280971 elements. Since one element is 8 bytes long, the smallest vector needs 8000 bytes, while the largest is 643 MB long.

5.5.1 Vector addition

Each test was repeated 5000 times, and the execution time was measured. The performance was calculated in FLOPS (Floating-point Operations Per Second) based on the vector lengths and the execution time. We count only the effective floating-point operations, i.e. the multiplication by λ and the addition. The number of effective floating-point operations determines how long input vectors can be processed by the current implementation under a fixed time range. If an implementation uses additional auxiliary floating-point operations (like multiplying by a ratio), those operations do not count.

The input vectors were randomly generated. If we add two numbers, then we have two cases: (1) The result is stable, so we keep it, (2) or the result violates a tolerance, so it is set to zero. Hence, we have generated the input vectors in a way that the likelihood for setting the result to zero is $\frac{1}{2}$. This method does not support the efficiency of the CPU's branch-prediction mechanism. Moreover, if it is required to set for zero half of the results, it ensures that the non-vectorized implementations have to execute all of their branches.

We distinguish two cases of the vector addition operation:

1. $\mathbf{c} = \mathbf{a} + \lambda\mathbf{b}$, three vectors case

2. $\mathbf{a} := \mathbf{a} + \lambda\mathbf{b}$, two vectors case

where the memory areas of the vectors $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ are different. Since these cases use the memory in different ways we tested them for every implementation.

Results for three vectors

If three different memory areas are used with cache, the cache is divided into 3 partitions, so the performance is decreased. However, if non-temporal memory writing is used, then larger vectors can be placed in the cache. Moreover, if the larger cache is still tight, the non-temporal writing saves unnecessary memory operations. Therefore, this writing mode is recommended for large vectors. Figure (C.1) shows the results for the unstable implementations. It can be seen that the AVX is the best alternative, because it can perform four floating point operations per CPU cycle. The performance decreases if the vectors grow out of the available cache sizes. Since the L3 cache is shared among the cores, our process cannot use the whole cache, so the efficiency decreases sooner as the total vector sizes exceed the size of larger caches.

If the vectors are too large, the non-temporal SSE2 and AVX implementations have the same performance because they execute quick calculating operations, but the speed of memory operations is much slower than a floating-point operation. This holds for the cache writing implementations too, but their performance is half of that of the non-temporal versions, because they use slower memory operations.

Figure (C.2) shows the results for the stable add implementations, where relative tolerance is used. Since more operations are used for one stable add step, the performance is lower than in the unstable case. If the vectors are larger than the non-temporal writing version with AVX is a little faster than the non-temporal SSE2 because the AVX instructions

have to wait fewer times to read data from memory. While the 9 steps of the stable add are executed the CPU can read the next data into the cache.

As Figure (C.3) shows, the performance of the absolute tolerance versions has a similar behavior to the unstable implementations but, of course, in this case the performance is lower. However, the AVX versions have almost the same performances.

Zigzag curves can be observed in Figures (C.14), (C.15), and (C.16), because the Xeon CPU responds poorly to cache-free implementations with a small number of elements.

Results for two vectors

If two vectors are used and one of them is the result, the cache line of the current result memory area is in the cache. This entails that there is no additional communication between the cache and the memory, so the performance increases. Obviously, bypassing the cache is not unprofitable in this case, as Figures (C.4), (C.5), and (C.6) show. If the cache is not bypassed, the overall performance is better than in the three vectors case.

Overhead analysis

We examined how much slower stable adder implementations are than non-stable, naive versions. To do this, we compared the measurement results of each stable adder with naive (using cache) and calculated for each test vector size how the performance of the stable adder is proportional to naive. The results for the i7 processor are shown in Figures (C.11) and (C.12). It can be seen that in 3 and 2 vector cases, the performance of AVX implementations is closer to that of naive than that of SSE2. Figures (C.24) and (C.25) show the results of Xeon Platinum, based on which the AVX-512 strongly approximates the performance of a naive implementation.

Orchard-Hays's relative tolerance method

Since SSE2, AVX and AVX-512 have a MAX operation which selects the maximum of two numbers, Orchard-Hays's relative tolerance test can be implemented on Intel's SIMD architecture. As mentioned in subsection (5.2.1), two additional operations are inserted into the assembly code; the max selector and an absolute value operation. The modified implementation uses 11 instructions, where the max operation requires significant amount of execution time, as Figures (C.7)-(C.10) show. In most cases our algorithm is the fastest, the highest speedup ratio was 1.281 (1.379 on the Xeon CPU) in the 3 vector SSE2 test, using cache on the i7 CPU. The worst ratio was 0.97 in the 3 vector, SSE2, and cache-free case. We mention that this is a very extreme case, in most cases, if our approach is worse, the ratio moves around 0.99. However, as we saw, a simple policy can be constructed: depending on the vector's size, and numbers (2 or 3 vectors), we can choose between the cache and cache-free implementations. We can avoid most of the situations, when our method's performance is lower than the original.

Currently, the Pannon Optimizer uses the C++ version of the introduced algorithms, so we tested the simplified and Orchard-Hays's method. We emphasize that during the

test we only compared the old and the new method, and for this we used the Pannon Optimizer developed for this purpose, we did not perform any tests in other software. The test results can be seen in Appendix (D). For most of the test problems, the software became significantly faster using our simplified solution. There are three problems where the total execution time was more, but the average time of an iteration decreased; the origin of longer execution time was the more iteration number. The CYCLE problem suffered the most drastic slowdown (-20.52 %).

5.5.2 Dot-product

The dot product requires only two vectors and the result is a scalar value. Since, in general, the input vectors have much more than one element, the writing time of the result to the memory is irrelevant. The stable AVX implementation uses only 7 instructions in addition to the loading, multiplying, and add operations, so its performance is better than the stable add. As Figure (C.13) shows, the performance of stable AVX dot product is close to the unstable AVX version on the i7 CPU. The stable SSE2 requires more cycles, so the performance is considerably lower than the unstable SSE2 version. Meanwhile on the Xeon Platinum (see Figure (C.26), the most efficient stable implementation is the AVX-512 version, but for small vectors, it is much slower than the naive version, but its handicap is almost gone at larger inputs. The figures shows that if there is no SIMD support, the branching-free techniques can be very useful if the input vectors are sufficiently large. Of course, in dot-product tests, it can be observed that the lower the level of caches used by the program (i.e., the fewer data to be expected), the more different implementations differ in terms of performance. Later, when the vectors are too high, and the system cannot take advantage of the cache, the performances will be nearly the same.

5.6 Conclusions

As the performance tests prove, our simplified stable add method is faster than Orchard-Hays's method. The applicability of our method is also tested by our simplex method implementation; the test problems of NETLIB were successfully solved. It is clear that our pointer arithmetic based stable dot-product implementation is much more efficient than the conditional branching version if the input vectors are sufficiently large. Moreover, the tests show that using Intel's SIMD instruction sets provides strong tools in order to implement the stable algorithms in an efficient way.

Modern Intel CPUs have at least two memory ports. So, while the next data set is loading from the memory, the CPU can execute complex computations on the previous set. This is why the AVX and AVX-512 are so efficient in high performance stable computations.

5.7   Major results and summary of accomplishments

In this chapter, we have introduced various low-level acceleration techniques.

- One common numerically stable adder is the Orchard-Hays algorithm. We proposed an accelerated alternative in Section 5.2, and the speed increase was detected during testing, and the solutions remained unchanged.

- Various SIMD (Single Instruction, Multiple Data) instruction sets are widespread in today's processors, of which we have developed accelerated, conditional branching free, parallel stable addition and dot-product operations based on Intel's SIMD architecture.

Related publication

- J. Smidla and I. Maros. "Stable vector operation implementations, using Intels SIMD architecture". In: *Acta Polytechnica Hungarica* 15.1 (2018)

Related conference presentations

- J. Smidla, P. Tar, and I. Maros. "Adaptive Stable Additive Methods for Linear Algebraic Calculations". In: 20th Conference of the International Federation of Operational Research Societies (Barcelona, Spain). 2014

Summary

The new results described in the dissertation are briefly summarized below. In the first two chapters we presented the results, concepts and connections important for the dissertation in the field of linear optimization, simplex method and floating-point number representation. In the third chapter, we described an accelerated BTRAN algorithm that significantly accelerated the computational speed in most test problems. The measurement results are shown in the Appendix A. In the next chapter we described the most important results of the dissertation.

As we know, today's LP solver software gives a wrong answer for many LO problems, for example, the problem can be solved, but the software's answer is that there is no solution or it is unbounded. It is important for the user of the software to be informed that they want to solve a problem that cannot be handled with traditional double precision floating-point number representation, so they should switch to a slower but more accurate number representation, or even the software can do it automatically. To do this, we present a heuristic detection algorithm that slightly increases runtime, but notices if the task is numerically unmanageable and informs the user about it. The runtime changing was tested for different settings, the results are placed in the Appendix B.

Finally, in the Chapter 5, we described the acceleration possibilities of numerically stable addition operations. We presented an alternative stable addition algorithm that simplifies Orchard-Hays's procedure and uses fewer instructions, which slightly increases the speed based on our measurements, but the solving software still gives correct results. The measurement results can be found in the Appendix D. Also, since the traditional implementation of these methods includes conditional jumping instructions, we presented implementations based on SIMD technology that are free of these, and due to the data-level parallelization, a significant increase in the speed of these operations can be achieved on a larger data set. This was also measured on two different target computers, a traditional desktop and a server built on scientific calculations, the results are included in the Appendix C.

Appendix A

Column-wise and row-wise BTRAN test results

Name	Column-wise		Row-wise		Speedup	Iteration speedup
	Iterations	Time	Iterations	Time		
cre-a.mps	10760	0.554	10760	0.5614	0.9869	0.9869
cre-b.mps	145214	36.14	150888	82.93	0.4358	0.4528
cre-c.mps	11085	0.6013	11085	0.6639	0.9058	
cre-d.mps	121078	26.46	121078	57.78	0.4579	0.4579
ken-07.mps	3220	0.08376	3220	0.04454	1.881	
ken-11.mps	22692	6.56	22692	3.429	1.913	
ken-13.mps	99873	72.16	101580	172.9	0.4172	0.4243
pds-20.mps	84984	98.17	85773	107.5	0.913	0.9215
25FV47.SIF	10425	2.307	10781	1.413	1.633	1.689
80BAU3B.SIF	4222	0.151	4236	0.09611	1.571	1.576
ADLITTLE.SIF	155	0.0004702	155	0.0007409	0.6347	
AFIRO.SIF	20	5.572e-06	20	1.79e-05	0.3113	
AGG2.SIF	164	0.001509	164	0.0003373	4.473	
AGG3.SIF	173	0.001826	173	0.0004179	4.368	
BANDM.SIF	814	0.04261	814	0.03768	1.131	

Name	Column-wise		Row-wise		Speedup	Iteration speedup
	Iterations	Time	Iterations	Time		
BEACONFD.SIF	164	0.0008677	164	0.0005094	1.703	1.703
BLEND.SIF	93	0.0002236	93	0.0002765	0.8084	
BNL1.SIF	3477	0.2269	3480	0.1819	1.248	1.249
BNL2.SIF	6500	0.8703	6488	0.3796	2.292	2.288
BOEING1.SIF	552	0.01063	552	0.007764	1.369	1.369
BOEING2.SIF	180	0.0008274	180	0.0009156	0.9036	
BORE3D.SIF	210	0.001392	210	0.001192	1.168	1.168
BRANDY.SIF	490	0.01416	490	0.0208	0.6807	
CAPRI.SIF	288	0.00211	288	0.001046	2.017	
CZPROB.SIF	2387	0.03879	2387	0.03687	1.052	
D2Q06C.SIF	28132	20.16	27766	10.91	1.848	1.824
D6CUBE.SIF	1156	0.1308	1156	0.1455	0.8992	
DEGEN2.SIF	1226	0.1177	1233	0.1157	1.017	1.023
DEGEN3.SIF	8343	4.33	7720	3.373	1.284	1.188
E226.SIF	714	0.02053	714	0.02291	0.8963	
ETAMACRO.SIF	941	0.02051	941	0.01322	1.551	1.551
FFFFF800.SIF	1320	0.04689	1320	0.0416	1.127	
FINNIS.SIF	377	0.002788	377	0.001822	1.53	
FIT1D.SIF	96	0.0002713	96	0.0005968	0.4546	0.4546
FIT1P.SIF	1629	0.4564	1629	0.06499	7.022	
FIT2D.SIF	202	0.001329	202	0.00317	0.4193	
FIT2P.SIF	8191	13.64	8191	0.8225	16.58	
FORPLAN.SIF	391	0.00709	391	0.008465	0.8376	
GANGES.SIF	1357	0.02828	1357	0.01094	2.585	
GFRD-PNC.SIF	538	0.006851	538	0.006085	1.126	
GREENBEA.SIF	9285	1.923	8869	1.19	1.616	1.544
GREENBEB.SIF	13830	3.059	13471	2.083	1.469	1.431
GROW15.SIF	5150	0.4462	5150	0.3762	1.186	
GROW22.SIF	9577	1.19	9575	0.8228	1.447	1.446
GROW7.SIF	1465	0.07342	1465	0.087	0.844	

	Column-wise		Row-wise			
Name	Iterations	Time	Iterations	Time	Speedup	Iteration speedup
ISRAEL.SIF	483	0.02077	481	0.01822	1.139	1.135
KB2.SIF	84	0.0002088	84	0.0003652	0.5717	
LOTFI.SIF	319	0.003087	319	0.003407	0.906	0.906
MAROS-R7.SIF	4486	5.79	4486	5.215	1.11	1.11
MAROS.SIF	7156	0.9679	7239	0.5903	1.64	1.659
MODSZK1.SIF	747	0.01367	747	0.01327	1.03	1.03
NESM.SIF	2710	0.08968	2704	0.0595	1.507	1.504
PEROLD.SIF	3850	0.8053	3850	0.5202	1.548	1.548
PILOT-JA.SIF	5364	1.691	5347	0.8225	2.056	2.049
PILOT-WE.SIF	6765	1.668	6768	0.9471	1.761	1.762
PILOT.SIF	9285	8.506	9285	6.107	1.393	
PILOT4.SIF	1274	0.1068	1274	0.08167	1.308	1.308
PILOTNOV.SIF	1759	0.3356	1759	0.1623	2.067	2.067
QAP8.SIF	22948	15.81	16205	7.979	1.981	1.399
RECIPELP.SIF	48	1.96e-05	48	5.017e-05	0.3906	
SC105.SIF	122	0.000378	123	0.0005826	0.6488	0.6542
SC205.SIF	217	0.001487	217	0.001905	0.7802	0.7802
SC50A.SIF	50	4.375e-05	50	0.0001037	0.4218	0.4218
SC50B.SIF	49	3.736e-05	49	8.951e-05	0.4174	0.4174
SCAGR25.SIF	579	0.01068	579	0.006835	1.562	1.562
SCAGR7.SIF	208	0.001184	208	0.001466	0.8075	
SCFXM1.SIF	550	0.01092	550	0.005898	1.852	
SCFXM2.SIF	1135	0.03693	1135	0.01555	2.375	2.375
SCFXM3.SIF	1748	0.0869	1748	0.03016	2.881	2.881
SCORPION.SIF	360	0.002453	360	0.001432	1.713	1.713
SCRS8.SIF	742	0.01254	742	0.008207	1.528	
SCSD1.SIF	133	0.0004349	133	0.0008199	0.5304	
SCSD6.SIF	452	0.009113	452	0.01387	0.6569	
SCSD8.SIF	1913	0.1259	1913	0.1028	1.225	
SCTAP1.SIF	481	0.006822	481	0.005496	1.241	

Name	Column-wise		Row-wise		Speedup	Iteration speedup
	Iterations	Time	Iterations	Time		
SCTAP2.SIF	944	0.01251	944	0.006374	1.963	1.963
SCTAP3.SIF	1206	0.01757	1206	0.00848	2.072	
SEBA.SIF	439	0.001856	439	0.001109	1.674	
SHARE1B.SIF	394	0.009144	397	0.01131	0.8088	0.8149
SHARE2B.SIF	195	0.001327	195	0.001127	1.178	1.178
SHELL.SIF	599	0.005039	599	0.006929	0.7272	0.7272
SHIP04L.SIF	380	0.001894	380	0.0008154	2.323	2.323
SHIP04S.SIF	381	0.001805	381	0.0007275	2.482	
SHIP08L.SIF	690	0.005376	690	0.00357	1.506	
SHIP08S.SIF	646	0.005111	646	0.002804	1.823	1.823
SHIP12L.SIF	1157	0.01282	1157	0.007758	1.652	
SHIP12S.SIF	1090	0.01152	1090	0.006345	1.816	
SIERRA.SIF	626	0.006186	626	0.004052	1.527	
STAIR.SIF	456	0.03242	456	0.03615	0.897	0.897
STANDATA.SIF	161	0.0002842	161	0.000381	0.746	
STANDGUB.SIF	132	0.0001635	132	0.0002212	0.7394	
STANDMPS.SIF	286	0.001004	286	0.0008803	1.141	1.141
STOCFOR1.SIF	99	0.0001685	99	0.0002292	0.7352	0.7352
STOCFOR2.SIF	2369	0.2126	2369	0.0506	4.201	
STOCFOR3.SIF	18294	12.2	18294	2.319	5.262	
TRUSS.SIF	7018	2.523	7018	1.776	1.421	
TUFF.SIF	604	0.01571	604	0.01754	0.8958	
VTP-BASE.SIF	351	0.007725	351	0.006073	1.272	
WOOD1P.SIF	625	0.03345	625	0.0765	0.4372	0.4372
WOODW.SIF	2633	0.195	2955	0.314	0.6211	0.697

Appendix B

Numerical error detector test results

Name	Configuration-A		Configuration-B		Slow-down
	Iterations	Time	Iterations	Time	
cre-a.mps	9904	2.542	9944	2.61	0.974
cre-b.mps	119589	431.1	126991	492.6	0.8752
cre-c.mps	11559	2.971	11350	2.959	1.004
cre-d.mps	100722	353.1	126487	470.2	0.7508
ken-07.mps	3219	0.6639	3219	0.6691	0.9922
ken-11.mps	22502	49.1	22405	47.26	1.039
ken-13.mps	101266	574.5	103000	591.4	0.9715
osa-07.mps	889	0.543	889	0.5515	0.9847
osa-14.mps	2115	4.142	2115	4.267	0.9707
osa-30.mps	3967	18.49	3967	18.45	1.002
osa-60.mps	9118	104.7	9118	104.7	1.001
pds-02.mps	2805	0.6413	2805	0.6469	0.9913
pds-06.mps	11660	14.13	11708	14.47	0.9761
pds-10.mps	22290	48.95	21644	47.89	1.022
pds-20.mps	81197	481.2	82919	523	0.92
25FV47.SIF	10675	3.054	10281	3.11	0.982
80BAU3B.SIF	4086	1.18	4086	1.186	0.9949
ADLITTLE.SIF	155	0.003016	155	0.003333	0.9049
AFIRO.SIF	20	0.000433	20	0.000468	0.9252

Name	Configuration-A		Configuration-B		Slow-down
	Iterations	Time	Iterations	Time	
AGG2.SIF	164	0.006125	164	0.006539	0.9367
AGG3.SIF	173	0.006594	173	0.007105	0.9281
BANDM.SIF	746	0.0632	746	0.06842	0.9237
BEACONFD.SIF	164	0.004298	164	0.004522	0.9505
BLEND.SIF	93	0.002123	93	0.002339	0.9077
BNL1.SIF	3446	0.3759	3446	0.3923	0.9582
BNL2.SIF	6398	1.523	6398	1.547	0.9847
BOEING1.SIF	555	0.0291	555	0.03093	0.9409
BOEING2.SIF	183	0.004476	183	0.004865	0.92
BORE3D.SIF	208	0.006709	208	0.007499	0.8947
BRANDY.SIF	466	0.02269	466	0.02543	0.8922
CAPRI.SIF	288	0.009512	288	0.0106	0.8973
CYCLE.SIF	14034	5.124	12846	4.867	1.053
CZPROB.SIF	2336	0.2933	2336	0.2988	0.9814
D2Q06C.SIF	26402	25.64	25343	26.12	0.9817
D6CUBE.SIF	1100	0.5642	1100	0.5882	0.9591
DEGEN2.SIF	1124	0.147	1124	0.1615	0.9102
DEGEN3.SIF	9201	6.557	9016	6.844	0.9582
E226.SIF	708	0.03801	708	0.04122	0.9221
ETAMACRO.SIF	980	0.05054	980	0.05237	0.9651
FFFFF800.SIF	1350	0.09098	1350	0.09239	0.9847
FINNIS.SIF	368	0.01382	368	0.01451	0.9529
FIT1D.SIF	96	0.009465	96	0.00982	0.9638
FIT1P.SIF	1629	0.3351	1629	0.3563	0.9407
FIT2D.SIF	202	0.1787	202	0.1802	0.9916
FIT2P.SIF	8191	14.37	8113	14.7	0.9773
FORPLAN.SIF	395	0.01781	395	0.01921	0.9269
GANGES.SIF	1356	0.1246	1356	0.1249	0.998
GFRD-PNC.SIF	544	0.02977	544	0.03104	0.9592
GREENBEA.SIF	8757	3.649	8892	3.853	0.947

Name	Configuration-A		Configuration-B		Slow-down
	Iterations	Time	Iterations	Time	
GREENBEB.SIF	14329	6.584	13457	6.524	1.009
GROW15.SIF	4854	0.7732	4854	0.8185	0.9447
GROW22.SIF	9835	2.496	9835	2.634	0.9475
GROW7.SIF	1465	0.1207	1465	0.1281	0.9427
ISRAEL.SIF	493	0.0233	493	0.02557	0.9109
KB2.SIF	84	0.001675	84	0.001856	0.9025
LOTFI.SIF	319	0.00911	319	0.009591	0.9498
MAROS-R7.SIF	4486	23.97	4486	25.67	0.9339
MAROS.SIF	7231	1.499	7119	1.543	0.972
MODSZK1.SIF	731	0.0555	731	0.058	0.9569
NESM.SIF	2339	0.2846	2339	0.5383	0.5288
PEROLD.SIF	4167	1.444	4159	1.531	0.9427
PILOT-JA.SIF	5509	3.478	5962	4.617	0.7534
PILOT-WE.SIF	7061	2.299	7065	2.462	0.9337
PILOT.SIF	8682	34.57	9284	43.72	0.7907
PILOT4.SIF	1286	0.288	1286	0.315	0.9145
PILOTNOV.SIF	1626	0.5988	1626	0.6305	0.9497
QAP8.SIF	19217	32.91	15302	26.86	1.225
RECIPELP.SIF	48	0.000879	48	0.000975	0.9015
SC105.SIF	122	0.003517	122	0.004085	0.861
SC205.SIF	215	0.009347	215	0.01056	0.885
SC50A.SIF	50	0.00111	50	0.001395	0.7957
SC50B.SIF	49	0.001084	49	0.0014	0.7743
SCAGR25.SIF	631	0.02983	631	0.03353	0.8896
SCAGR7.SIF	208	0.004601	208	0.004898	0.9394
SCFXM1.SIF	551	0.02317	551	0.0243	0.9536
SCFXM2.SIF	1133	0.08375	1133	0.08609	0.9729
SCFXM3.SIF	1759	0.1833	1759	0.1854	0.9886
SCORPION.SIF	362	0.0122	362	0.0129	0.946
SCRS8.SIF	742	0.03975	742	0.04035	0.985

Name	Configuration-A		Configuration-B		Slow-down
	Iterations	Time	Iterations	Time	
SCSD1.SIF	136	0.005368	136	0.005651	0.9499
SCSD6.SIF	400	0.02348	400	0.02523	0.9305
SCSD8.SIF	3113	0.4608	3113	0.4785	0.963
SCTAP1.SIF	449	0.01468	449	0.0153	0.9593
SCTAP2.SIF	907	0.06747	907	0.0685	0.9849
SCTAP3.SIF	1234	0.1212	1234	0.1225	0.9896
SEBA.SIF	439	0.01703	439	0.2546	0.06689
SHARE1B.SIF	391	0.0143	391	0.01517	0.9428
SHARE2B.SIF	195	0.004689	195	0.005102	0.9191
SHELL.SIF	599	0.03724	599	0.03814	0.9766
SHIP04L.SIF	378	0.01816	378	0.01862	0.9753
SHIP04S.SIF	381	0.01581	381	0.01598	0.9898
SHIP08L.SIF	685	0.06147	685	0.06274	0.9797
SHIP08S.SIF	647	0.04315	647	0.04455	0.9685
SHIP12L.SIF	1148	0.136	1148	0.1377	0.9879
SHIP12S.SIF	1089	0.0927	1089	0.09545	0.9712
SIERRA.SIF	626	0.04839	626	0.05031	0.9619
STAIR.SIF	462	0.1165	462	0.1375	0.8471
STANDATA.SIF	165	0.005971	165	0.006181	0.966
STANDGUB.SIF	132	0.004857	132	0.005039	0.9639
STANDMPS.SIF	295	0.01167	295	0.01221	0.9557
STOCFOR1.SIF	99	0.00215	99	0.002368	0.9079
STOCFOR2.SIF	2249	0.4259	2249	0.4309	0.9885
STOCFOR3.SIF	17929	41.45	17711	43.19	0.9595
TRUSS.SIF	7739	4.658	7723	4.893	0.952
TUFF.SIF	626	0.03487	626	0.03708	0.9403
VTP-BASE.SIF	346	0.01095	346	0.01153	0.9501
WOOD1P.SIF	625	0.2632	625	0.2728	0.9648
WOODW.SIF	2953	0.99	2953	1.004	0.9865

Name	Configuration-A		Configuration-C		Slow-down
	Iterations	Time	Iterations	Time	
cre-a.mps	9904	2.542	9976	2.827	0.8992
cre-b.mps	119589	431.1	126475	492.4	0.8756
cre-c.mps	11559	2.971	11649	3.078	0.9653
cre-d.mps	100722	353.1	112219	407.9	0.8655
ken-07.mps	3219	0.6639	3219	0.671	0.9894
ken-11.mps	22502	49.1	22503	49.34	0.9952
ken-13.mps	101266	574.5	102366	584.8	0.9824
osa-07.mps	889	0.543	889	0.5487	0.9898
osa-14.mps	2115	4.142	2115	4.139	1.001
osa-30.mps	3967	18.49	3967	18.43	1.003
osa-60.mps	9118	104.7	9118	105.3	0.9941
pds-02.mps	2805	0.6413	2805	0.7249	0.8846
pds-06.mps	11660	14.13	11599	14.36	0.9838
pds-10.mps	22290	48.95	22849	51.78	0.9454
pds-20.mps	81197	481.2	81918	508.7	0.946
25FV47.SIF	10675	3.054	10280	3.127	0.9767
80BAU3B.SIF	4086	1.18	4086	1.194	0.988
ADLITTLE.SIF	155	0.003016	155	0.003644	0.8277
AFIRO.SIF	20	0.000433	20	0.000513	0.8441

	Configuration-A		Configuration-C		
Name	Iterations	Time	Iterations	Time	Slow-down
AGG2.SIF	164	0.006125	164	0.006906	0.8869
AGG3.SIF	173	0.006594	173	0.007733	0.8527
BANDM.SIF	746	0.0632	746	0.07039	0.8979
BEACONFD.SIF	164	0.004298	164	0.004623	0.9297
BLEND.SIF	93	0.002123	93	0.002371	0.8954
BNL1.SIF	3446	0.3759	3445	0.3928	0.957
BNL2.SIF	6398	1.523	6398	1.557	0.9779
BOEING1.SIF	555	0.0291	555	0.03281	0.8868
BOEING2.SIF	183	0.004476	183	0.005216	0.8581
BORE3D.SIF	208	0.006709	208	0.007695	0.8719
BRANDY.SIF	466	0.02269	466	0.02827	0.8027
CAPRI.SIF	288	0.009512	288	0.01074	0.886
CYCLE.SIF	14034	5.124	-1	-1	-5.124
CZPROB.SIF	2336	0.2933	2336	0.3012	0.9738
D2Q06C.SIF	26402	25.64	26173	27.3	0.9392
D6CUBE.SIF	1100	0.5642	1100	0.5883	0.9589
DEGEN2.SIF	1124	0.147	1124	0.1622	0.9062
DEGEN3.SIF	9201	6.557	9108	6.808	0.9632
E226.SIF	708	0.03801	708	0.04204	0.9039
ETAMACRO.SIF	980	0.05054	980	0.05697	0.8871
FFFFF800.SIF	1350	0.09098	1350	0.09487	0.959
FINNIS.SIF	368	0.01382	368	0.01471	0.9398
FIT1D.SIF	96	0.009465	96	0.01074	0.8811
FIT1P.SIF	1629	0.3351	1629	0.3591	0.9334
FIT2D.SIF	202	0.1787	202	0.1812	0.9862
FIT2P.SIF	8191	14.37	8134	14.64	0.9816
FORPLAN.SIF	395	0.01781	395	0.02106	0.8457
GANGES.SIF	1356	0.1246	1356	0.128	0.9734
GFRD-PNC.SIF	544	0.02977	544	0.03235	0.9204
GREENBEA.SIF	8757	3.649	8895	3.838	0.9507

Name	Configuration-A		Configuration-C		Slow-down
	Iterations	Time	Iterations	Time	
GREENBEB.SIF	14329	6.584	12176	5.698	1.156
GROW15.SIF	4854	0.7732	4854	0.8282	0.9336
GROW22.SIF	9835	2.496	9834	2.595	0.9617
GROW7.SIF	1465	0.1207	1465	0.1325	0.9112
ISRAEL.SIF	493	0.0233	493	0.02784	0.8368
KB2.SIF	84	0.001675	84	0.001983	0.8447
LOTFI.SIF	319	0.00911	319	0.009472	0.9618
MAROS-R7.SIF	4486	23.97	4486	24.46	0.98
MAROS.SIF	7231	1.499	7233	1.559	0.9618
MODSZK1.SIF	731	0.0555	731	0.05859	0.9472
NESM.SIF	2339	0.2846	2339	0.2949	0.9651
PEROLD.SIF	4167	1.444	4167	1.531	0.9433
PILOT-JA.SIF	5509	3.478	5503	3.745	0.9288
PILOT-WE.SIF	7061	2.299	7061	2.399	0.9583
PILOT.SIF	8682	34.57	9271	43.31	0.7983
PILOT4.SIF	1286	0.288	1286	0.3131	0.9201
PILOTNOV.SIF	1626	0.5988	1626	0.6369	0.9402
QAP8.SIF	19217	32.91	15124	27.02	1.218
RECIPELP.SIF	48	0.000879	48	0.000984	0.8933
SC105.SIF	122	0.003517	122	0.003952	0.8899
SC205.SIF	215	0.009347	215	0.01053	0.8874
SC50A.SIF	50	0.00111	50	0.001353	0.8204
SC50B.SIF	49	0.001084	49	0.001368	0.7924
SCAGR25.SIF	631	0.02983	631	0.03222	0.9257
SCAGR7.SIF	208	0.004601	208	0.004898	0.9394
SCFXM1.SIF	551	0.02317	551	0.02537	0.9134
SCFXM2.SIF	1133	0.08375	1133	0.08909	0.9401
SCFXM3.SIF	1759	0.1833	1759	0.1913	0.9581
SCORPION.SIF	362	0.0122	362	0.0132	0.9246
SCRS8.SIF	742	0.03975	742	0.04365	0.9106

Name	Configuration-A		Configuration-C		Slow-down
	Iterations	Time	Iterations	Time	
SCSD1.SIF	136	0.005368	136	0.005617	0.9557
SCSD6.SIF	400	0.02348	400	0.02613	0.8986
SCSD8.SIF	3113	0.4608	3113	0.4834	0.9531
SCTAP1.SIF	449	0.01468	449	0.01565	0.9378
SCTAP2.SIF	907	0.06747	907	0.06991	0.965
SCTAP3.SIF	1234	0.1212	1234	0.125	0.9695
SEBA.SIF	439	0.01703	439	0.02097	0.8121
SHARE1B.SIF	391	0.0143	391	0.01516	0.9432
SHARE2B.SIF	195	0.004689	195	0.005208	0.9003
SHELL.SIF	599	0.03724	599	0.03988	0.9339
SHIP04L.SIF	378	0.01816	378	0.01986	0.9142
SHIP04S.SIF	381	0.01581	381	0.01621	0.9757
SHIP08L.SIF	685	0.06147	685	0.06404	0.9598
SHIP08S.SIF	647	0.04315	647	0.04596	0.9389
SHIP12L.SIF	1148	0.136	1148	0.1414	0.9616
SHIP12S.SIF	1089	0.0927	1089	0.09845	0.9416
SIERRA.SIF	626	0.04839	626	0.05125	0.9442
STAIR.SIF	462	0.1165	462	0.1387	0.8398
STANDATA.SIF	165	0.005971	165	0.006926	0.8621
STANDGUB.SIF	132	0.004857	132	0.005338	0.9099
STANDMPS.SIF	295	0.01167	295	0.01274	0.9159
STOCFOR1.SIF	99	0.00215	99	0.002408	0.8929
STOCFOR2.SIF	2249	0.4259	2249	0.4314	0.9874
STOCFOR3.SIF	17929	41.45	17729	42.35	0.9786
TRUSS.SIF	7739	4.658	7501	4.692	0.9926
TUFF.SIF	626	0.03487	626	0.03749	0.9302
VTP-BASE.SIF	346	0.01095	346	0.01229	0.8913
WOOD1P.SIF	625	0.2632	625	0.2737	0.9615
WOODW.SIF	2953	0.99	2953	0.9992	0.9908

Name	Configuration-A		Configuration-D		Slow-down
	Iterations	Time	Iterations	Time	
cre-a.mps	9904	2.542	9904	2.575	0.9873
cre-b.mps	119589	431.1	117783	442.4	0.9744
cre-c.mps	11559	2.971	11559	3.038	0.9781
cre-d.mps	100722	353.1	109615	407.4	0.8666
ken-07.mps	3219	0.6639	3219	0.6744	0.9843
ken-11.mps	22502	49.1	22502	49.35	0.995
ken-13.mps	101266	574.5	98998	559.7	1.026
osa-07.mps	889	0.543	889	0.5449	0.9966
osa-14.mps	2115	4.142	2115	4.173	0.9927
osa-30.mps	3967	18.49	3967	17.95	1.03
osa-60.mps	9118	104.7	9118	101.8	1.029
pds-02.mps	2805	0.6413	2805	0.6502	0.9862
pds-06.mps	11660	14.13	11660	13.86	1.019
pds-10.mps	22290	48.95	22290	50.32	0.9727
pds-20.mps	81197	481.2	81133	465.1	1.034
25FV47.SIF	10675	3.054	10675	3.076	0.9927
80BAU3B.SIF	4086	1.18	4086	1.184	0.9962
ADLITTLE.SIF	155	0.003016	155	0.003442	0.8762
AFIRO.SIF	20	0.000433	20	0.000461	0.9393

Name	Configuration-A		Configuration-D		Slow-down
	Iterations	Time	Iterations	Time	
AGG2.SIF	164	0.006125	164	0.006764	0.9055
AGG3.SIF	173	0.006594	173	0.007155	0.9216
BANDM.SIF	746	0.0632	746	0.06863	0.9209
BEACONFD.SIF	164	0.004298	164	0.004518	0.9513
BLEND.SIF	93	0.002123	93	0.00234	0.9073
BNL1.SIF	3446	0.3759	3443	0.3868	0.9718
BNL2.SIF	6398	1.523	6398	1.536	0.9912
BOEING1.SIF	555	0.0291	555	0.03202	0.9087
BOEING2.SIF	183	0.004476	183	0.004747	0.9429
BORE3D.SIF	208	0.006709	208	0.007298	0.9193
BRANDY.SIF	466	0.02269	466	0.02545	0.8916
CAPRI.SIF	288	0.009512	288	0.01092	0.8713
CYCLE.SIF	14034	5.124	14034	5.169	0.9912
CZPROB.SIF	2336	0.2933	2336	0.3007	0.9753
D2Q06C.SIF	26402	25.64	26398	26.22	0.978
D6CUBE.SIF	1100	0.5642	1100	0.5811	0.9708
DEGEN2.SIF	1124	0.147	1124	0.1542	0.953
DEGEN3.SIF	9201	6.557	9201	6.61	0.9921
E226.SIF	708	0.03801	708	0.04129	0.9205
ETAMACRO.SIF	980	0.05054	980	0.05562	0.9086
FFFFF800.SIF	1350	0.09098	1350	0.09481	0.9596
FINNIS.SIF	368	0.01382	368	0.0153	0.9032
FIT1D.SIF	96	0.009465	96	0.009739	0.9719
FIT1P.SIF	1629	0.3351	1629	0.346	0.9687
FIT2D.SIF	202	0.1787	202	0.1803	0.9911
FIT2P.SIF	8191	14.37	8191	14.45	0.9941
FORPLAN.SIF	395	0.01781	395	0.0195	0.9135
GANGES.SIF	1356	0.1246	1356	0.1266	0.9845
GFRD-PNC.SIF	544	0.02977	544	0.03151	0.9447
GREENBEA.SIF	8757	3.649	8757	3.632	1.005

Name	Configuration-A		Configuration-D		Slow-down
	Iterations	Time	Iterations	Time	
GREENBEB.SIF	14329	6.584	14329	6.525	1.009
GROW15.SIF	4854	0.7732	4854	0.7967	0.9705
GROW22.SIF	9835	2.496	9835	2.533	0.9853
GROW7.SIF	1465	0.1207	1465	0.1283	0.9412
ISRAEL.SIF	493	0.0233	493	0.02418	0.9636
KB2.SIF	84	0.001675	84	0.001895	0.8839
LOTFI.SIF	319	0.00911	319	0.01022	0.8914
MAROS-R7.SIF	4486	23.97	4486	24.17	0.9919
MAROS.SIF	7231	1.499	7231	1.531	0.9791
MODSZK1.SIF	731	0.0555	731	0.05851	0.9485
NESM.SIF	2339	0.2846	2339	0.2879	0.9887
PEROLD.SIF	4167	1.444	4167	1.487	0.9712
PILOT-JA.SIF	5509	3.478	5509	3.599	0.9664
PILOT-WE.SIF	7061	2.299	7061	2.338	0.9832
PILOT.SIF	8682	34.57	8782	36.73	0.9412
PILOT4.SIF	1286	0.288	1286	0.3062	0.9408
PILOTNOV.SIF	1626	0.5988	1626	0.629	0.952
QAP8.SIF	19217	32.91	19217	33.43	0.9845
RECIPELP.SIF	48	0.000879	48	0.000971	0.9053
SC105.SIF	122	0.003517	122	0.004124	0.8528
SC205.SIF	215	0.009347	215	0.01028	0.9092
SC50A.SIF	50	0.00111	50	0.001362	0.815
SC50B.SIF	49	0.001084	49	0.001358	0.7982
SCAGR25.SIF	631	0.02983	631	0.03112	0.9585
SCAGR7.SIF	208	0.004601	208	0.00496	0.9276
SCFXM1.SIF	551	0.02317	551	0.02453	0.9446
SCFXM2.SIF	1133	0.08375	1133	0.08695	0.9633
SCFXM3.SIF	1759	0.1833	1759	0.1848	0.9918
SCORPION.SIF	362	0.0122	362	0.01285	0.9494
SCRS8.SIF	742	0.03975	742	0.04252	0.9347

Name	Configuration-A		Configuration-D		Slow-down
	Iterations	Time	Iterations	Time	
SCSD1.SIF	136	0.005368	136	0.00563	0.9535
SCSD6.SIF	400	0.02348	400	0.02611	0.8991
SCSD8.SIF	3113	0.4608	3113	0.4709	0.9785
SCTAP1.SIF	449	0.01468	449	0.01536	0.9553
SCTAP2.SIF	907	0.06747	907	0.07104	0.9497
SCTAP3.SIF	1234	0.1212	1234	0.1235	0.9814
SEBA.SIF	439	0.01703	439	0.02218	0.7681
SHARE1B.SIF	391	0.0143	391	0.01669	0.857
SHARE2B.SIF	195	0.004689	195	0.005061	0.9265
SHELL.SIF	599	0.03724	599	0.03953	0.9421
SHIP04L.SIF	378	0.01816	378	0.01996	0.9096
SHIP04S.SIF	381	0.01581	381	0.01655	0.9554
SHIP08L.SIF	685	0.06147	685	0.06393	0.9615
SHIP08S.SIF	647	0.04315	647	0.04483	0.9625
SHIP12L.SIF	1148	0.136	1148	0.1401	0.9705
SHIP12S.SIF	1089	0.0927	1089	0.09587	0.967
SIERRA.SIF	626	0.04839	626	0.05148	0.94
STAIR.SIF	462	0.1165	462	0.1401	0.8317
STANDATA.SIF	165	0.005971	165	0.006187	0.9651
STANDGUB.SIF	132	0.004857	132	0.005166	0.9402
STANDMPS.SIF	295	0.01167	295	0.01291	0.9042
STOCFOR1.SIF	99	0.00215	99	0.002394	0.8981
STOCFOR2.SIF	2249	0.4259	2249	0.4315	0.9872
STOCFOR3.SIF	17929	41.45	17929	41.48	0.9992
TRUSS.SIF	7739	4.658	7739	4.679	0.9954
TUFF.SIF	626	0.03487	626	0.03719	0.9377
VTP-BASE.SIF	346	0.01095	346	0.01187	0.9227
WOOD1P.SIF	625	0.2632	625	0.2739	0.9608
WOODW.SIF	2953	0.99	2953	0.9883	1.002

Appendix C

Low-level vector operation test results

C.1 Core i7-2640M tests

C.1.1 Vector addition
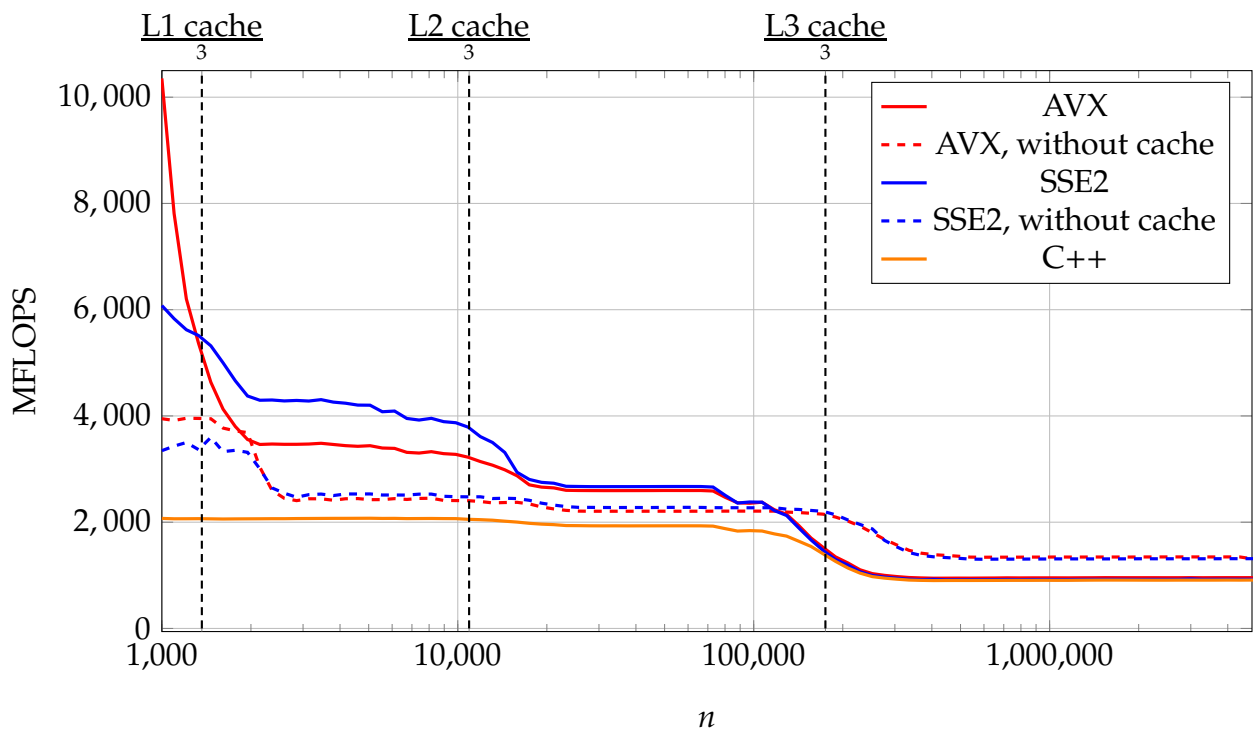
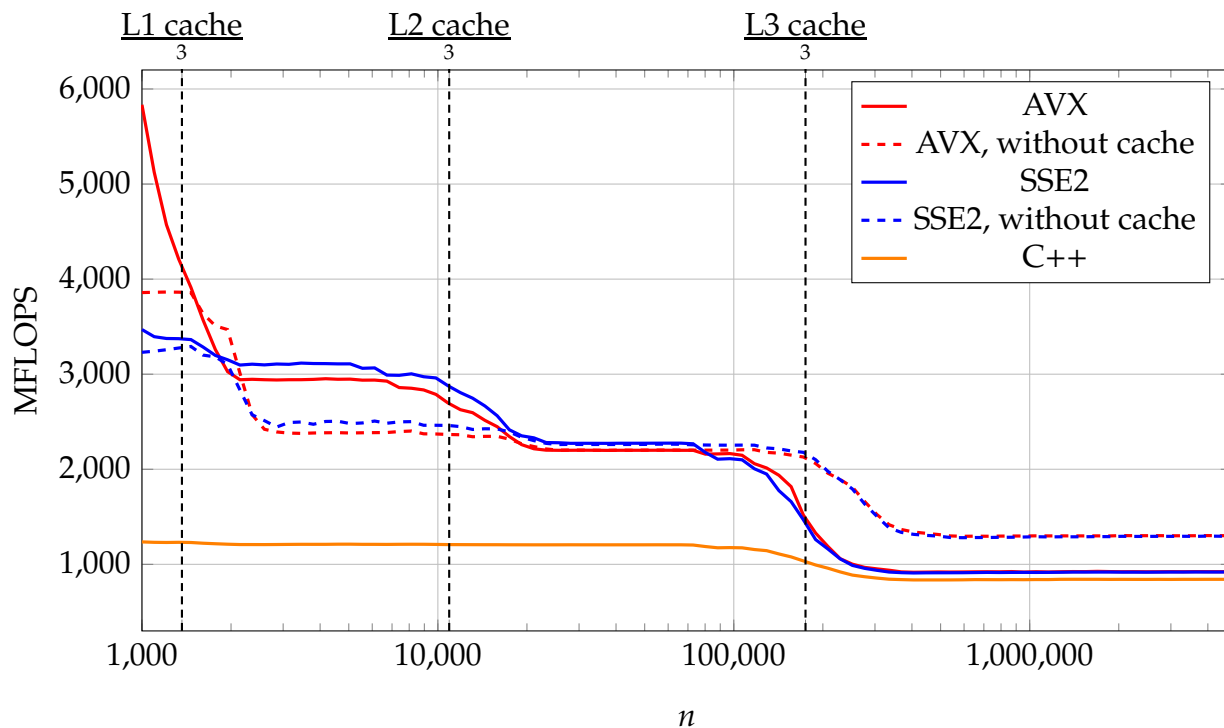Figure C.1: Performances of the unstable add vector implementations for three vectors

Figure C.2: Performances of the stable add vector implementations, using relative tolerance, for three vectors
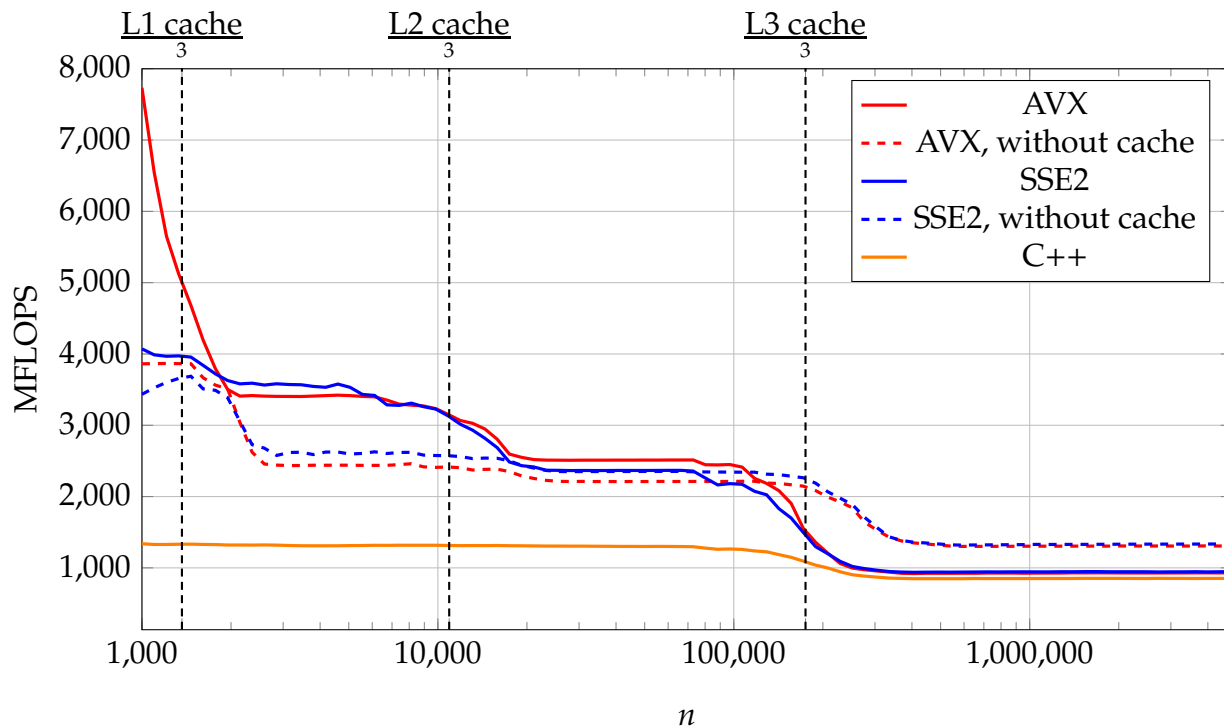

Figure C.3: Performances of the stable add vector implementations, using absolute tolerance, for three vectors
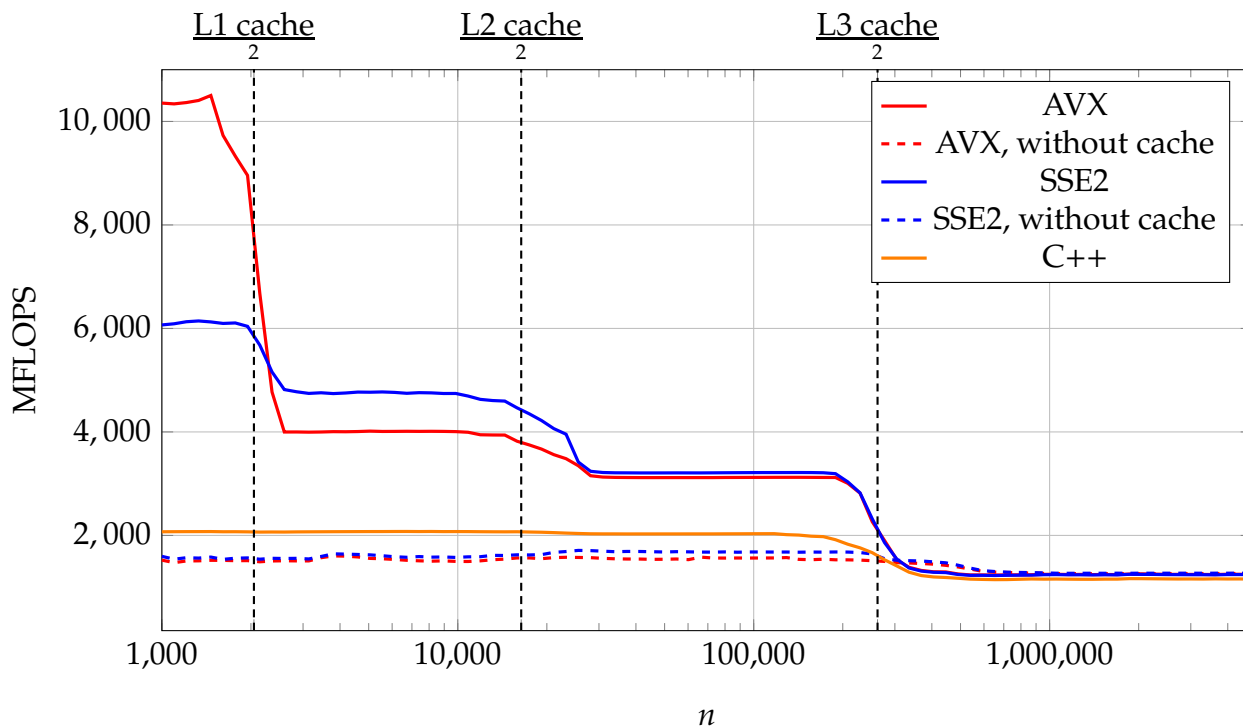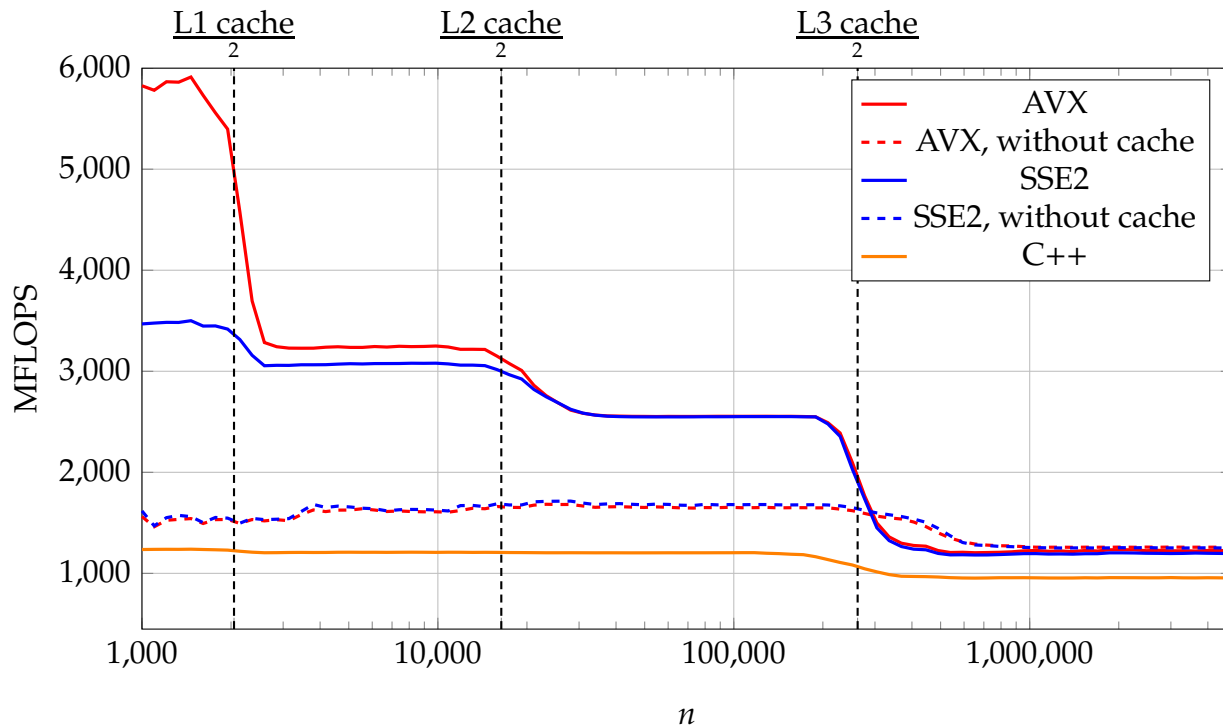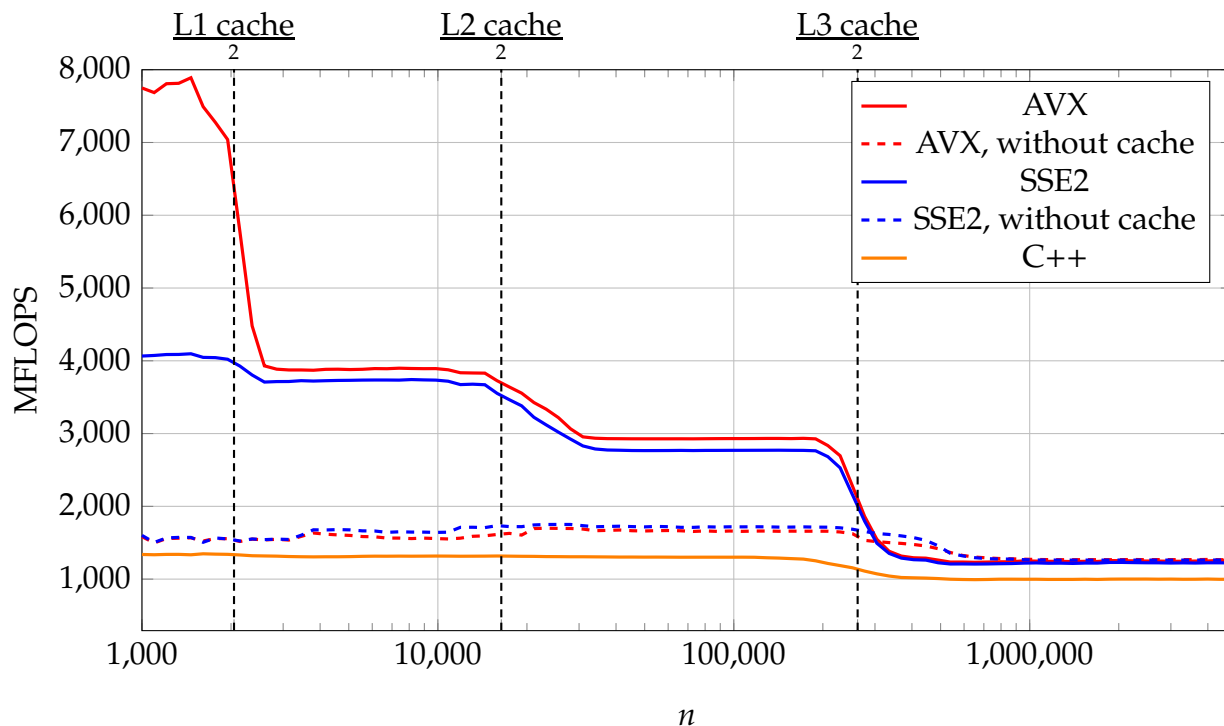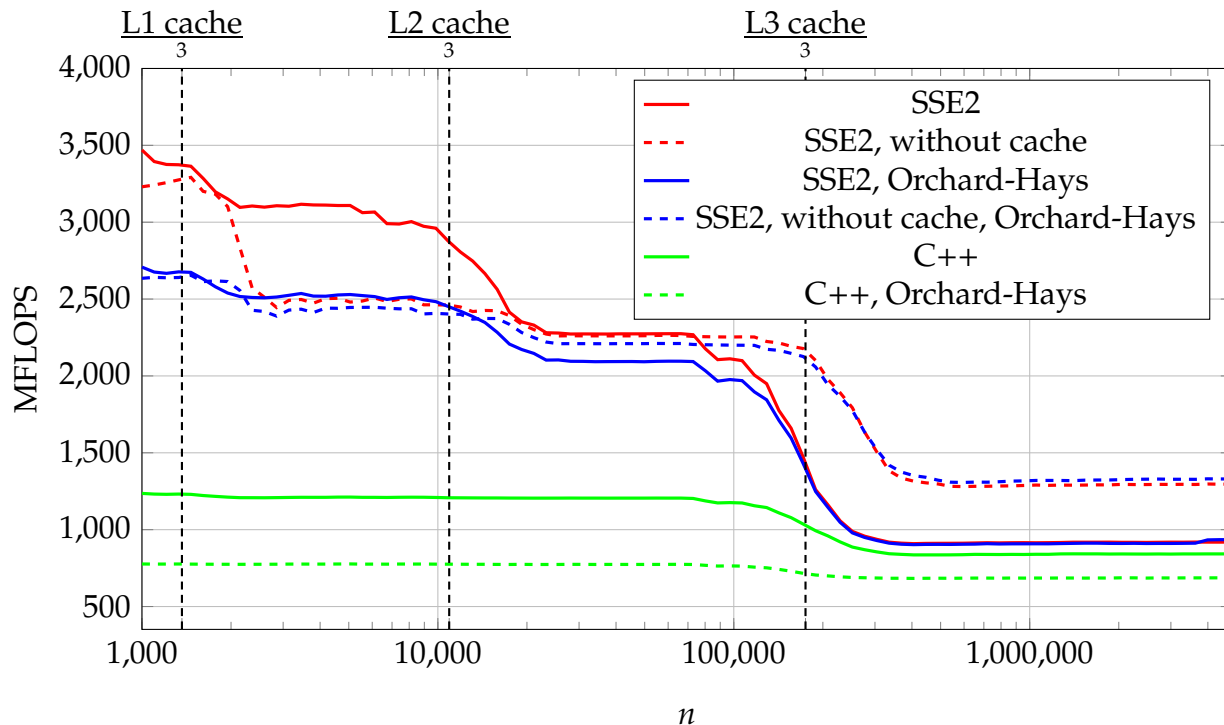
Figure C.4: Performances of the unstable add vector implementations for two vectors

Figure C.5: Performances of the stable add vector implementations, using relative tolerances for two vectors

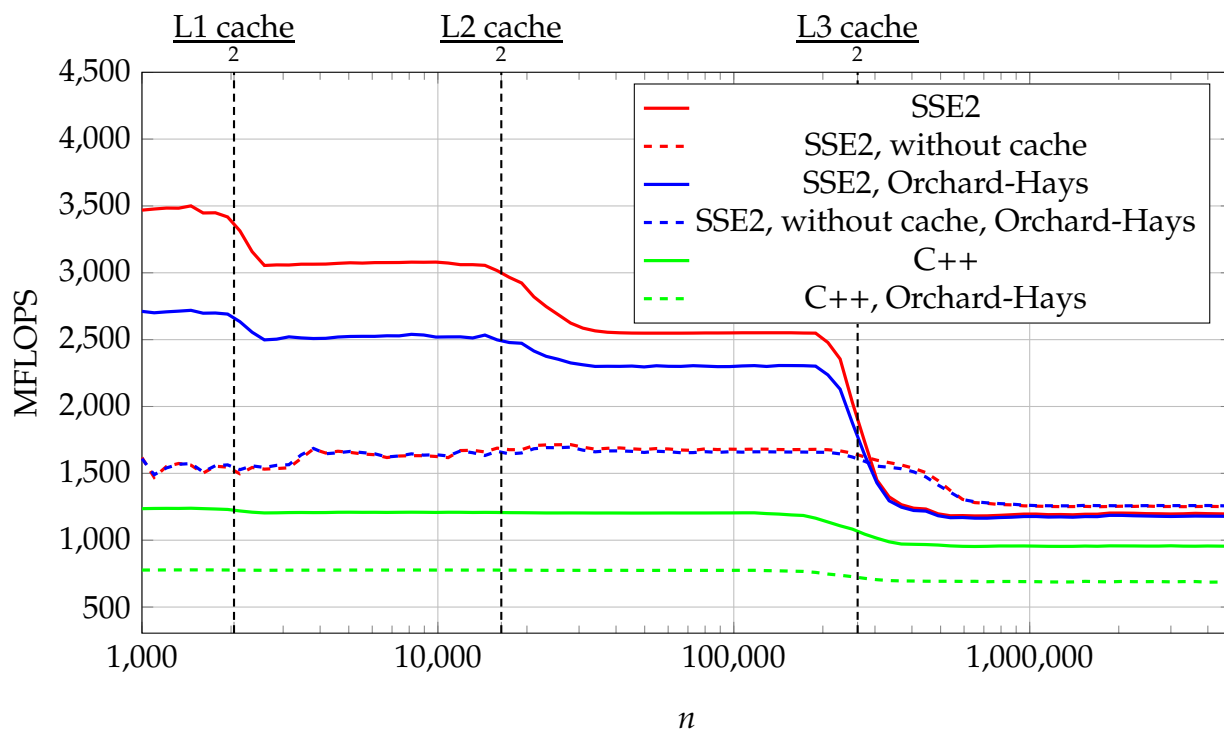Figure C.6: Performances of the stable add vector implementations, using absolute tolerance, for two vectors

Figure C.7: Performance comparison of our stable add implementations and the method of Orchard-Hays, with SSE2, using relative tolerance, for three vectors
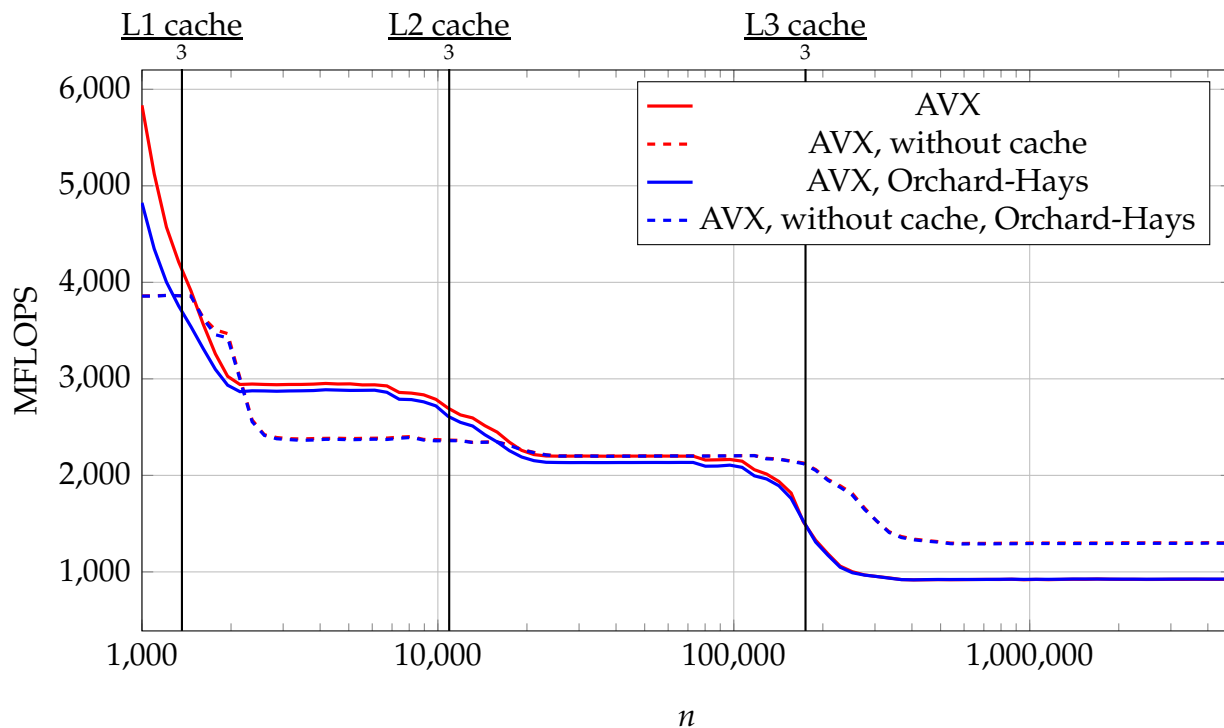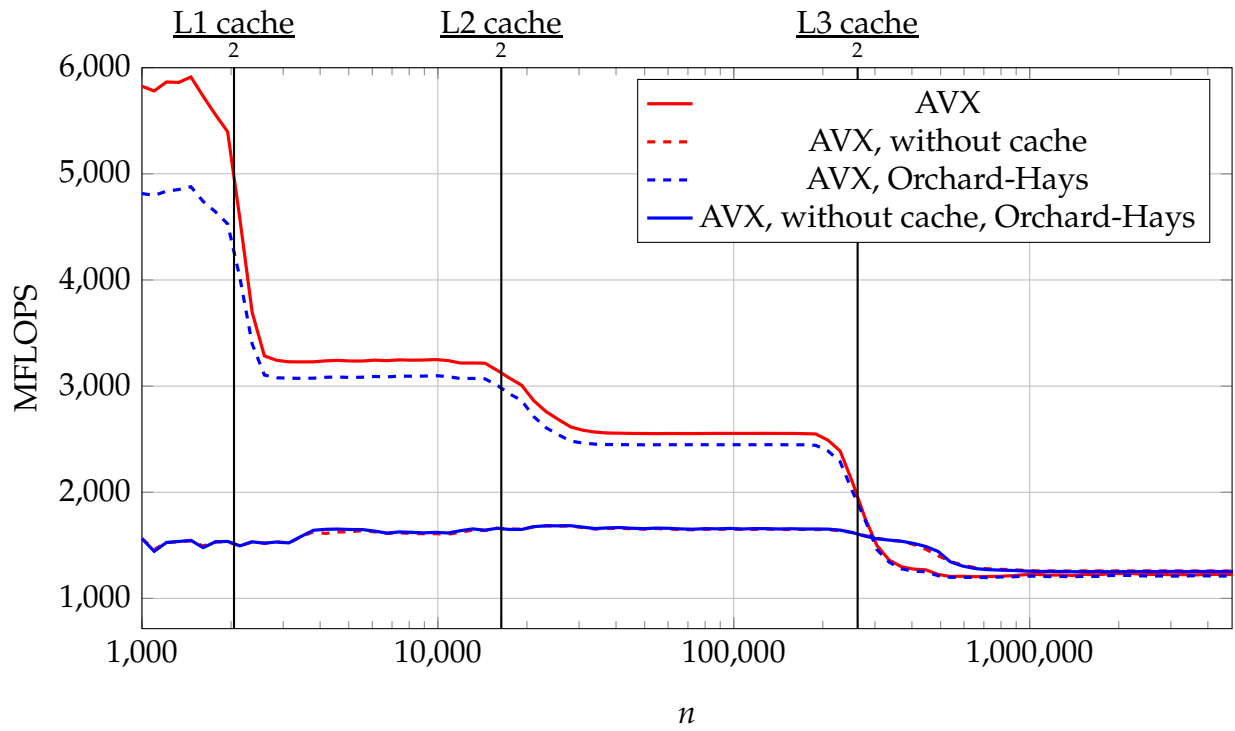
Figure C.8: Performance comparison of our stable add implementations and the method of Orchard-Hays, with AVX, using relative tolerance, for three vectors

Figure C.9: Performance comparison of our stable add implementations and the method of Orchard-Hays, with SSE2, using relative tolerance, for two vectors

Figure C.10: Performance comparison of our stable add implementations and the method of Orchard-Hays, with AVX, using relative tolerance, for two vectors
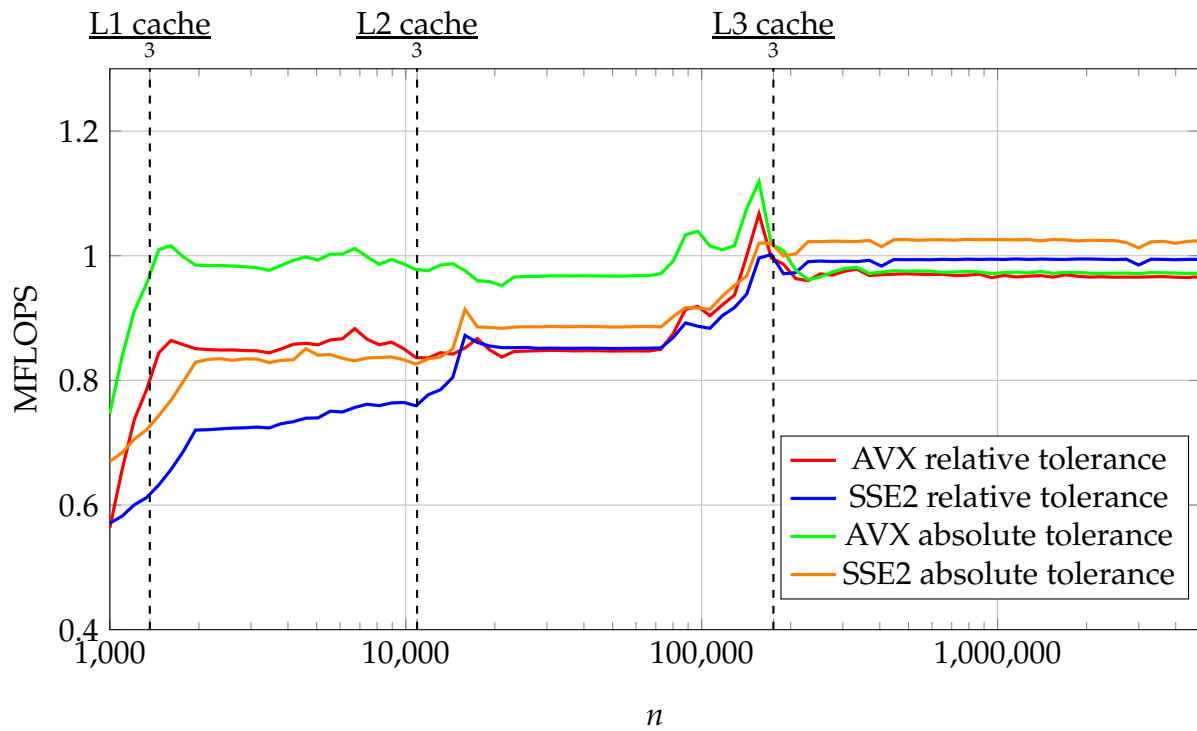

Figure C.11: Performance ratios relative to the naive versions, with 3 vectors
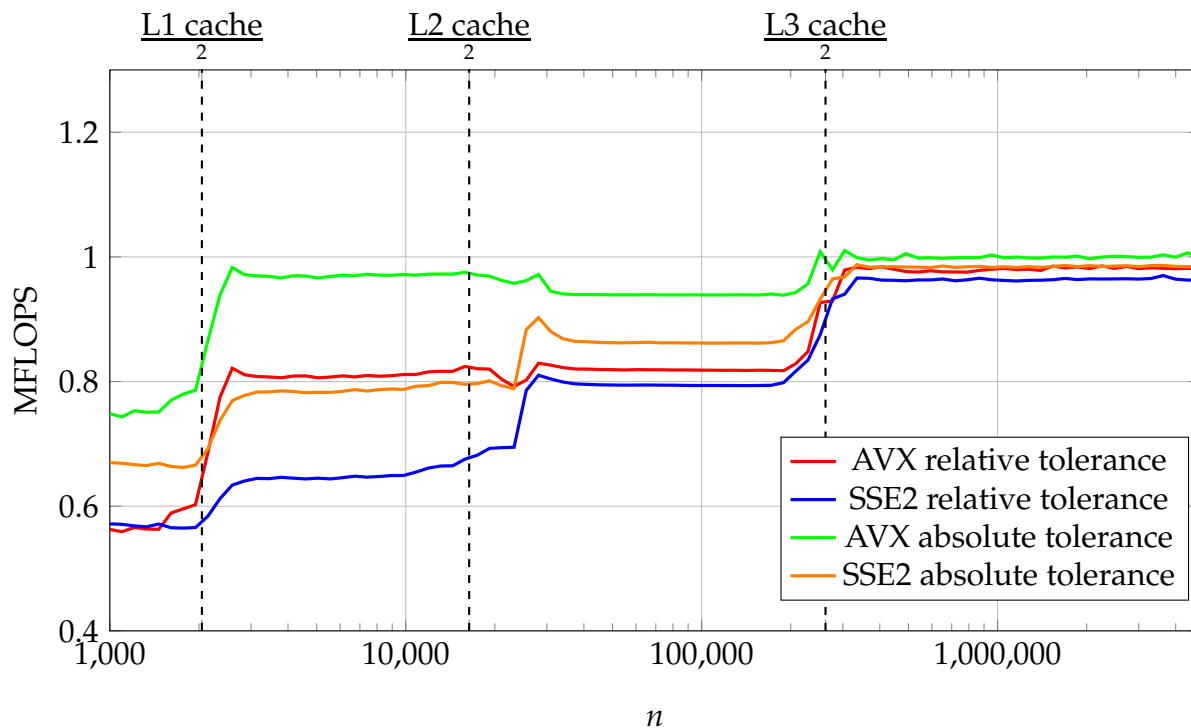
Figure C.12: Performance ratios relative to the naive versions, with 2 vectors
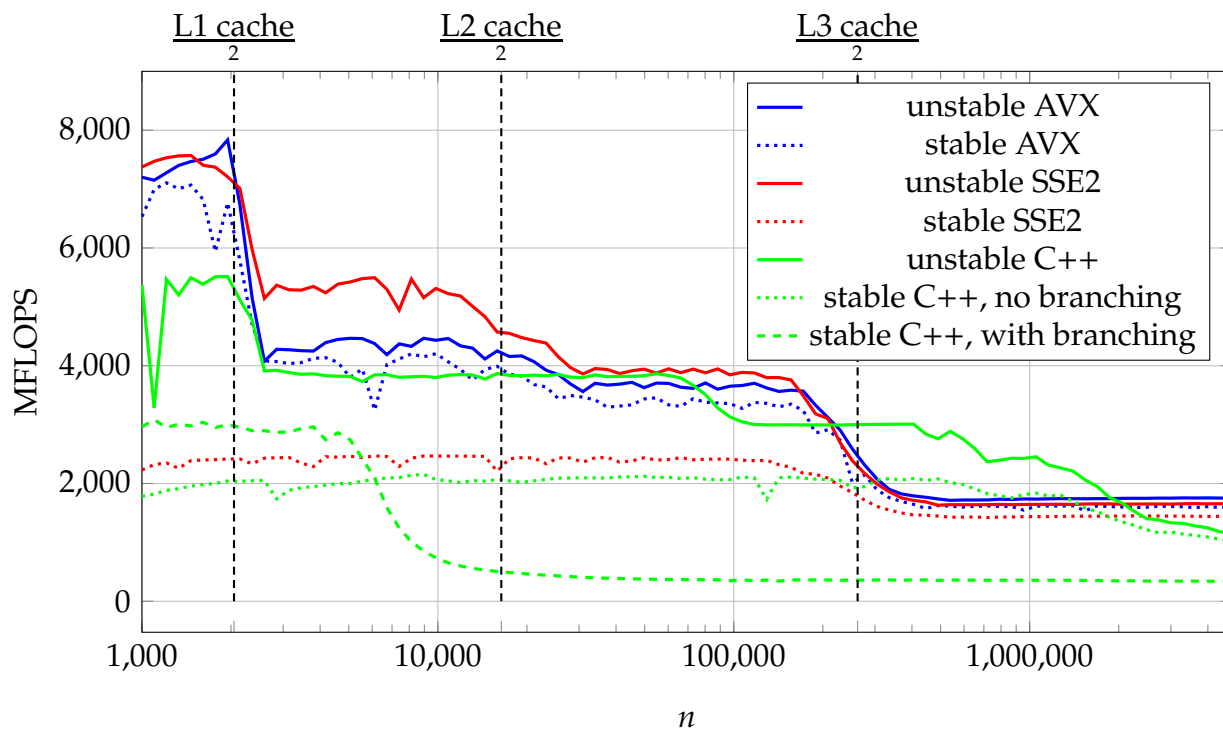
C.1.2   Dot product

Figure C.13: Performances of the dot product implementations

C.2 Xeon Platinum 8168 tests
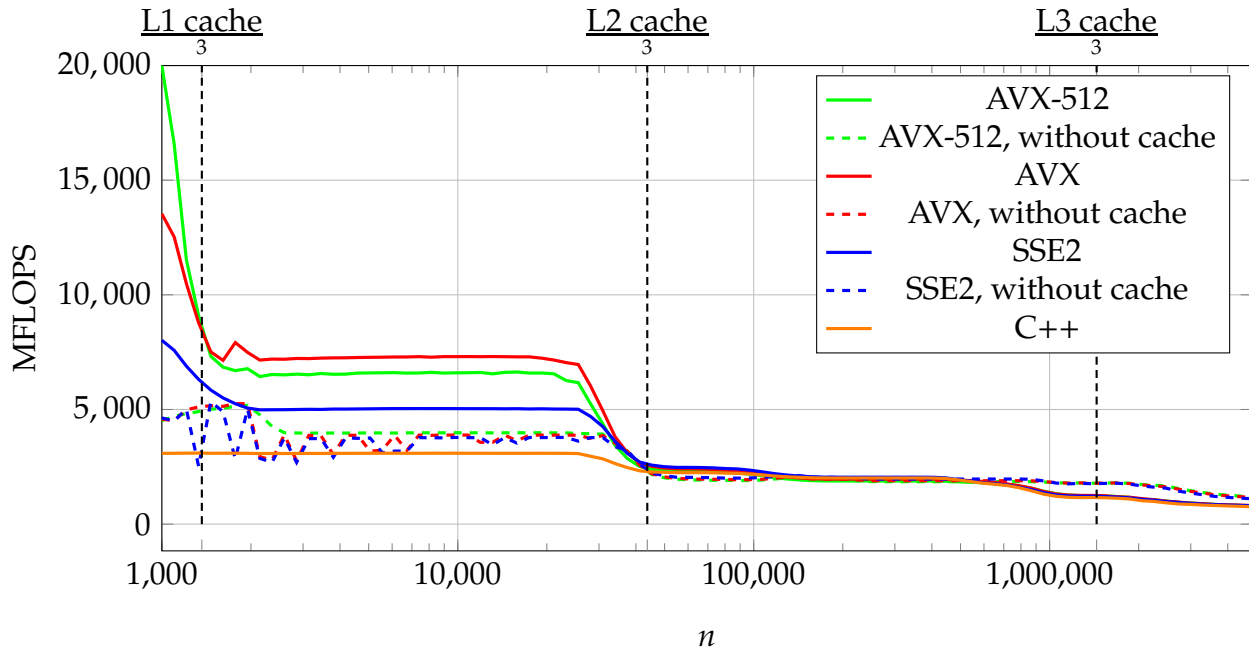
C.2.1 Vector addition

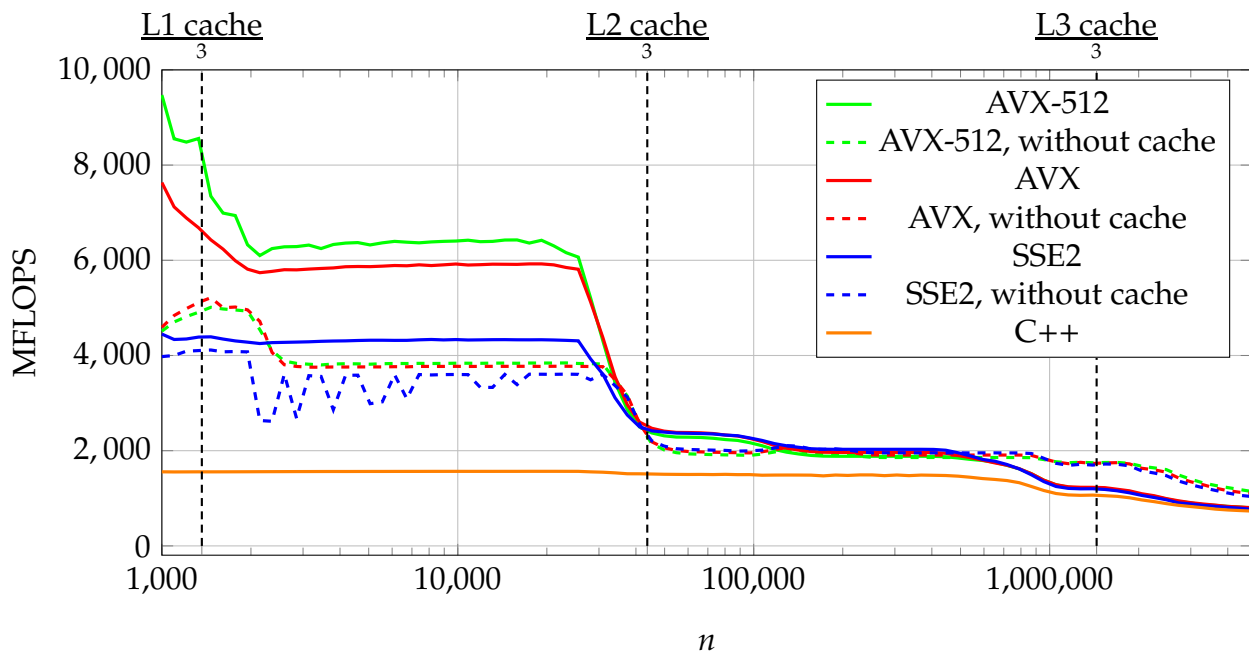Figure C.14: Performances of the unstable add vector implementations for three vectors, on Xeon Platinum

Figure C.15: Performances of the stable add vector implementations, using relative tolerance, for three vectors, on Xeon Platinum
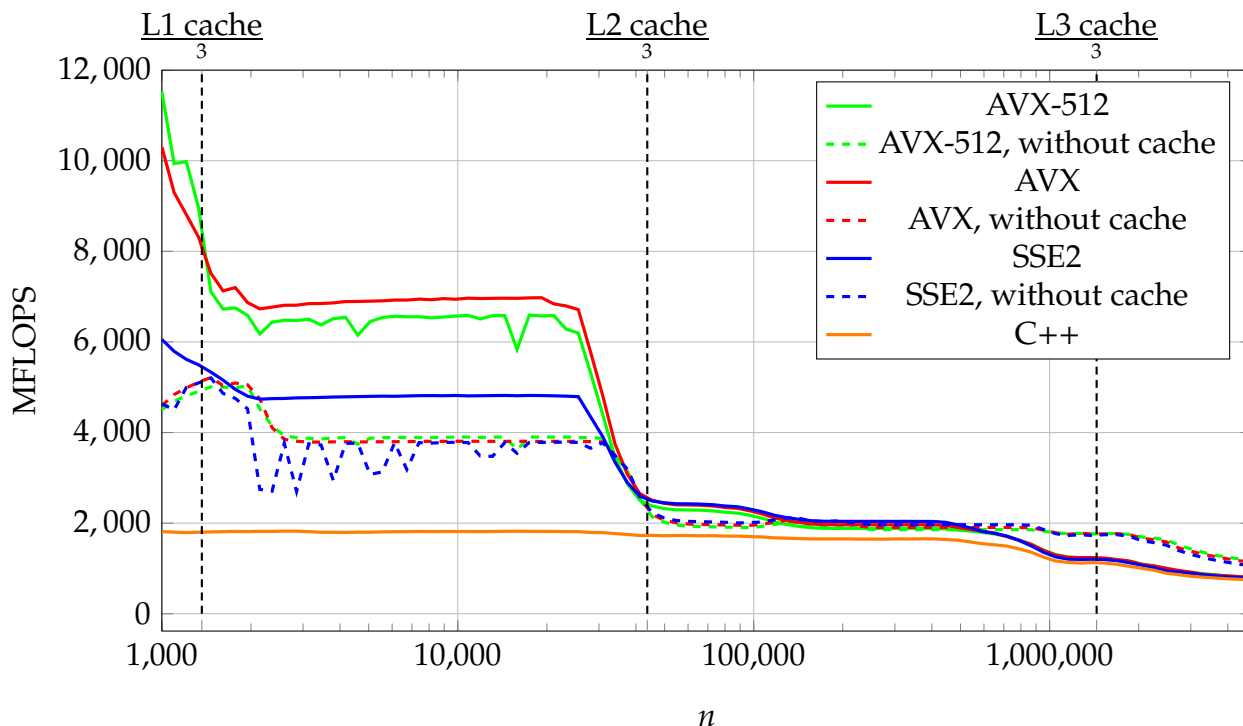
Figure C.16: Performances of the stable add vector implementations, using absolute tolerance, for three vectors, on Xeon Platinum
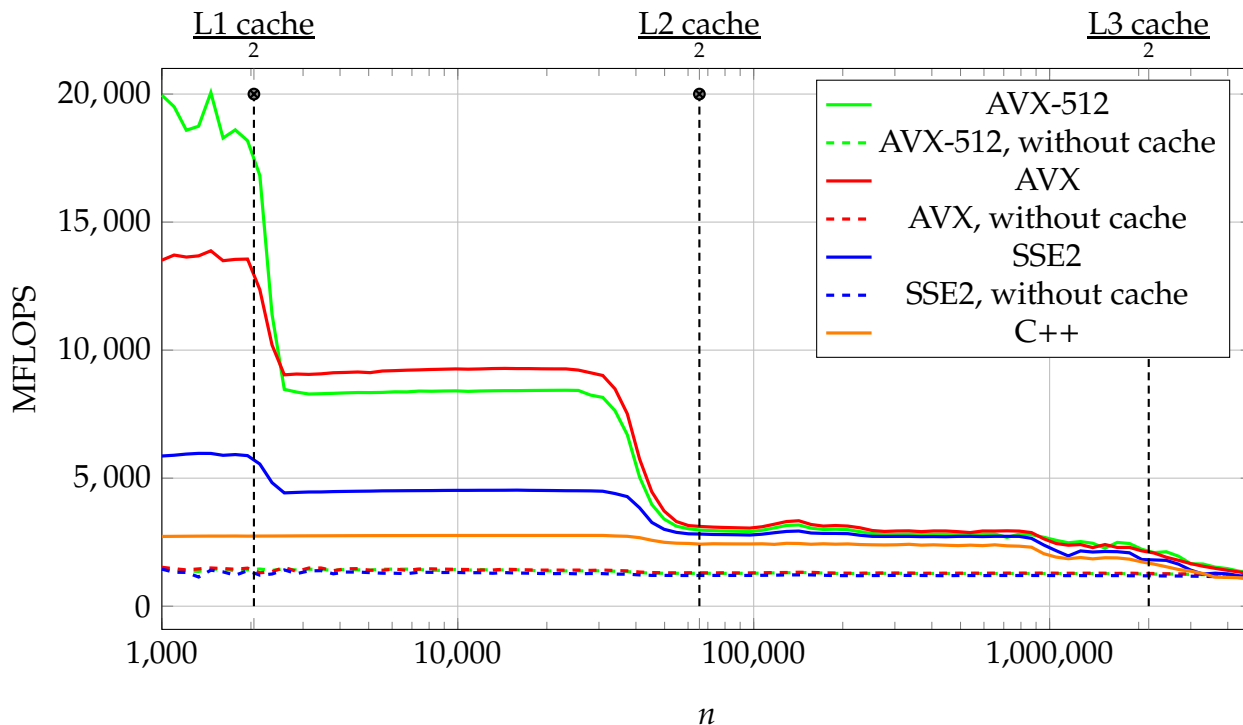

Figure C.17: Performances of the unstable add vector implementations for two vectors, on Xeon Platinum
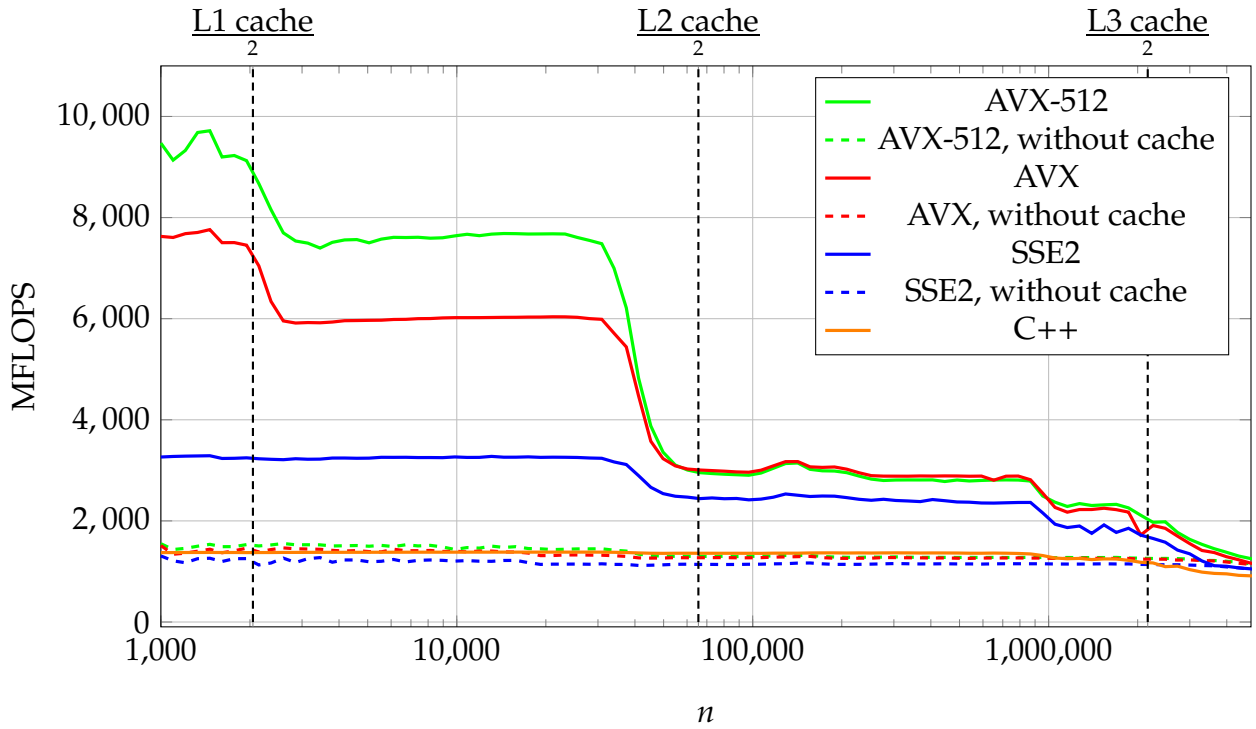
Figure C.18: Performances of the stable add vector implementations, using relative tolerances for two vectors, on Xeon Platinum
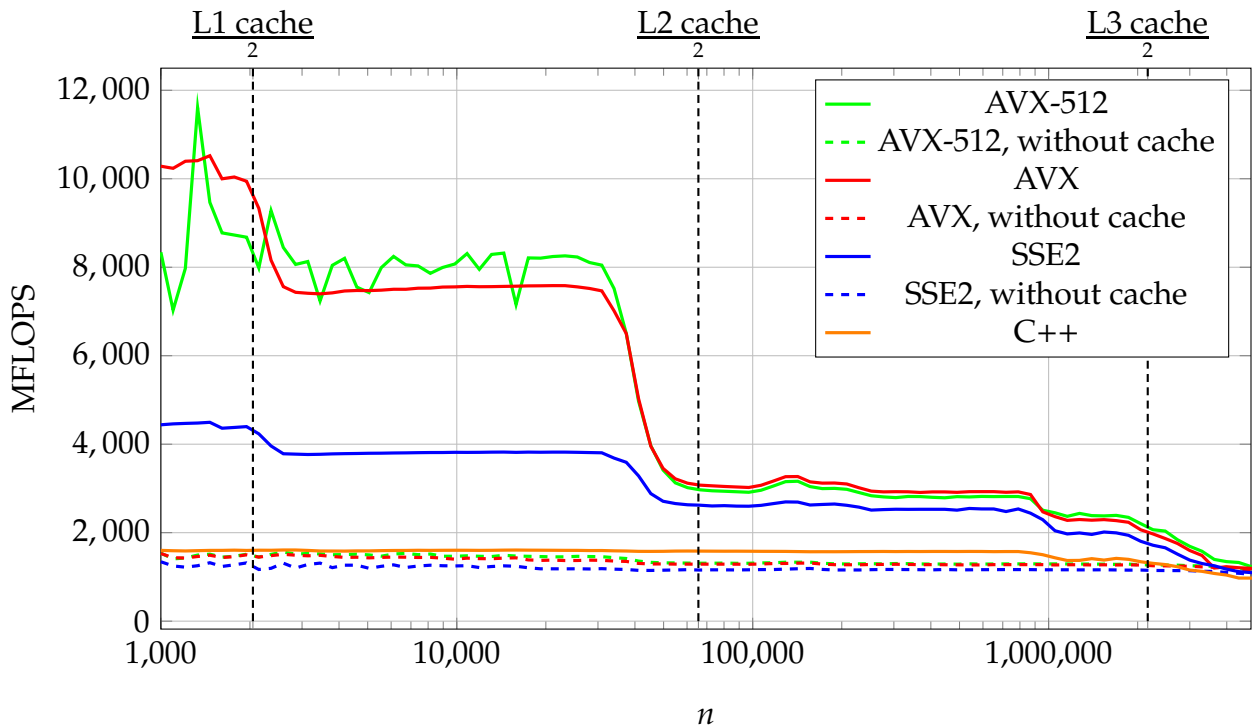
Figure C.19: Performances of the stable add vector implementations, using absolute tolerance, for two vectors, on Xeon Platinum
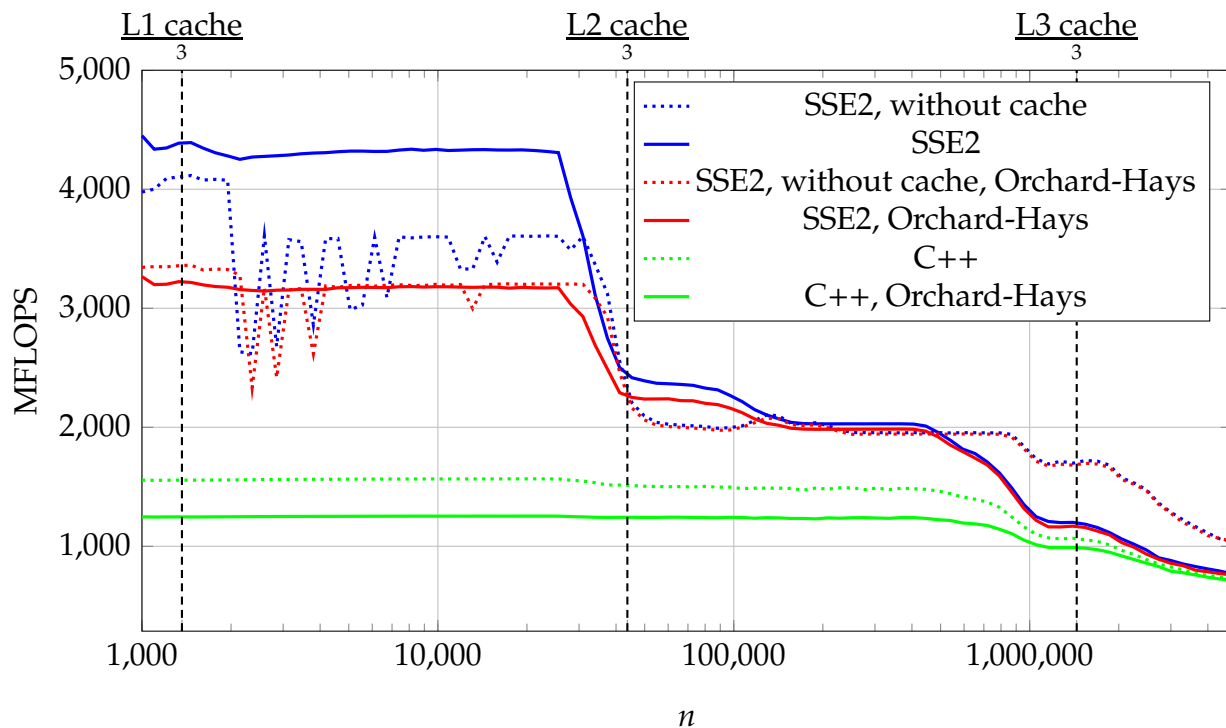
Figure C.20: Performance comparison of our stable add implementations and the method of Orchard-Hays, with SSE2, using relative tolerance, for three vectors, on Xeon Platinum
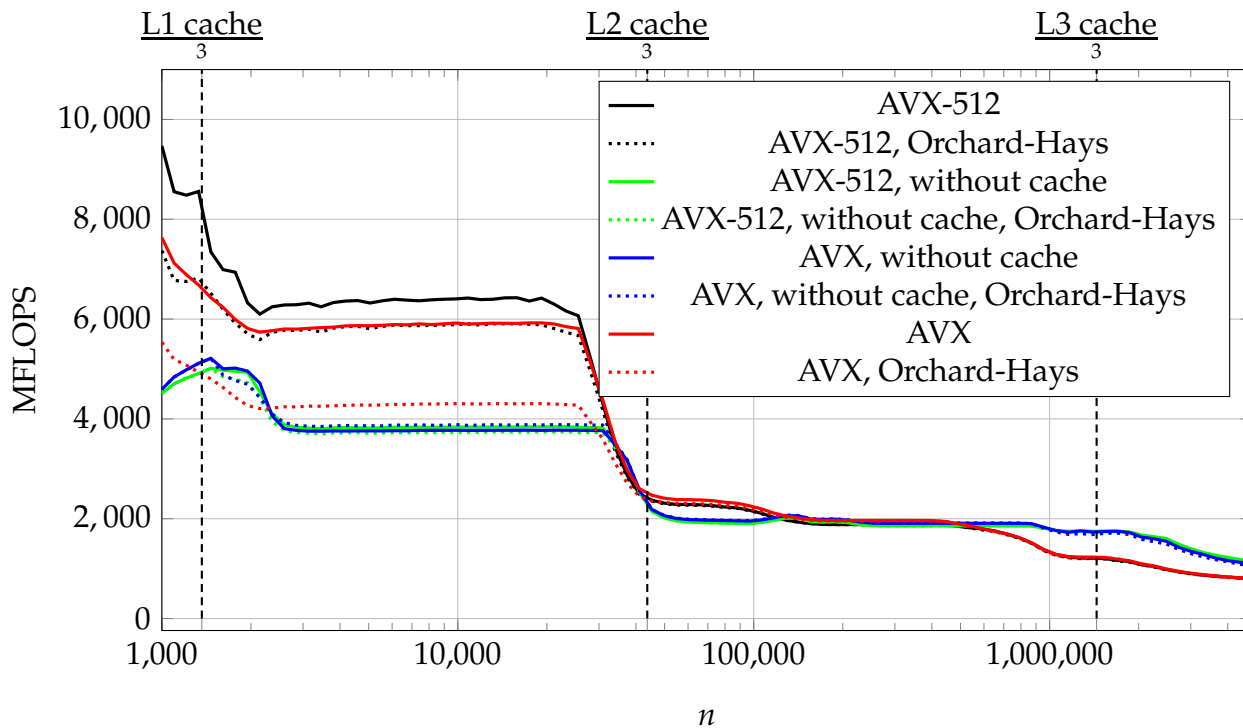
Figure C.21: Performance comparison of our stable add implementations and the method of Orchard-Hays, with AVX, using relative tolerance, for three vectors, on Xeon Platinum
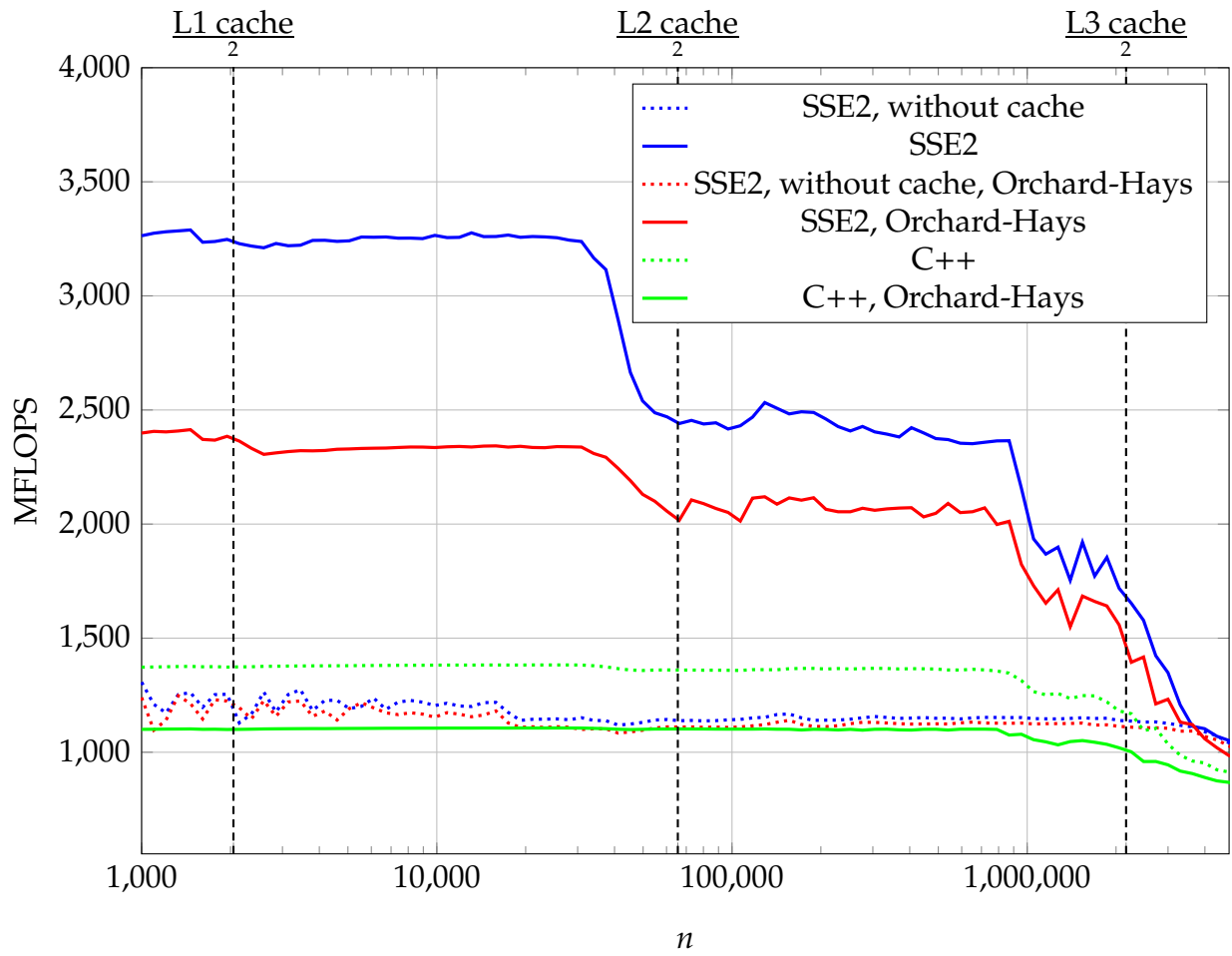
Figure C.22: Performance comparison of our stable add implementations and the method of Orchard-Hays, with SSE2, using relative tolerance, for two vectors, on Xeon Platinum
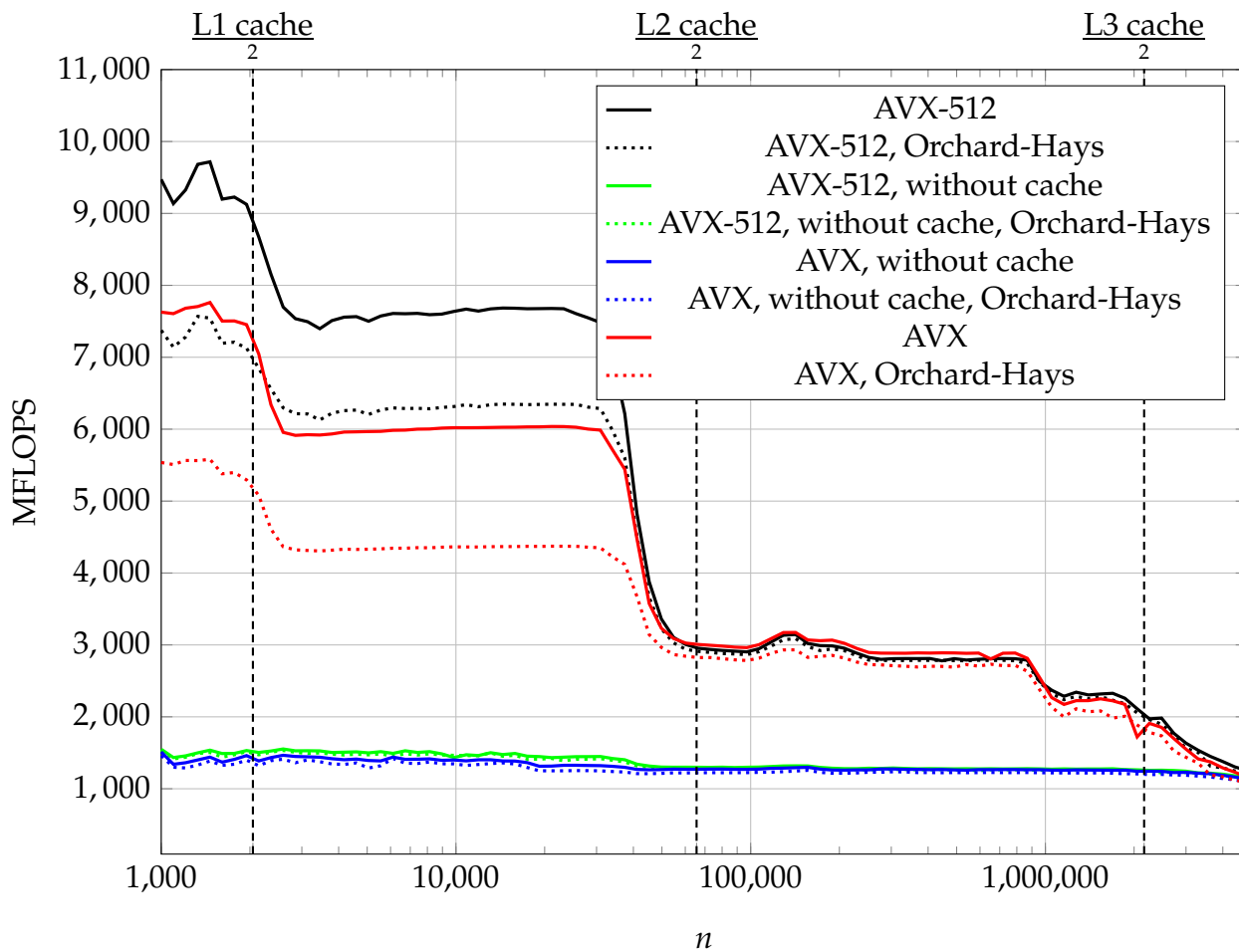
Figure C.23: Performance comparison of our stable add implementations and the method of Orchard-Hays, with AVX, using relative tolerance, for two vectors, on Xeon Platinum
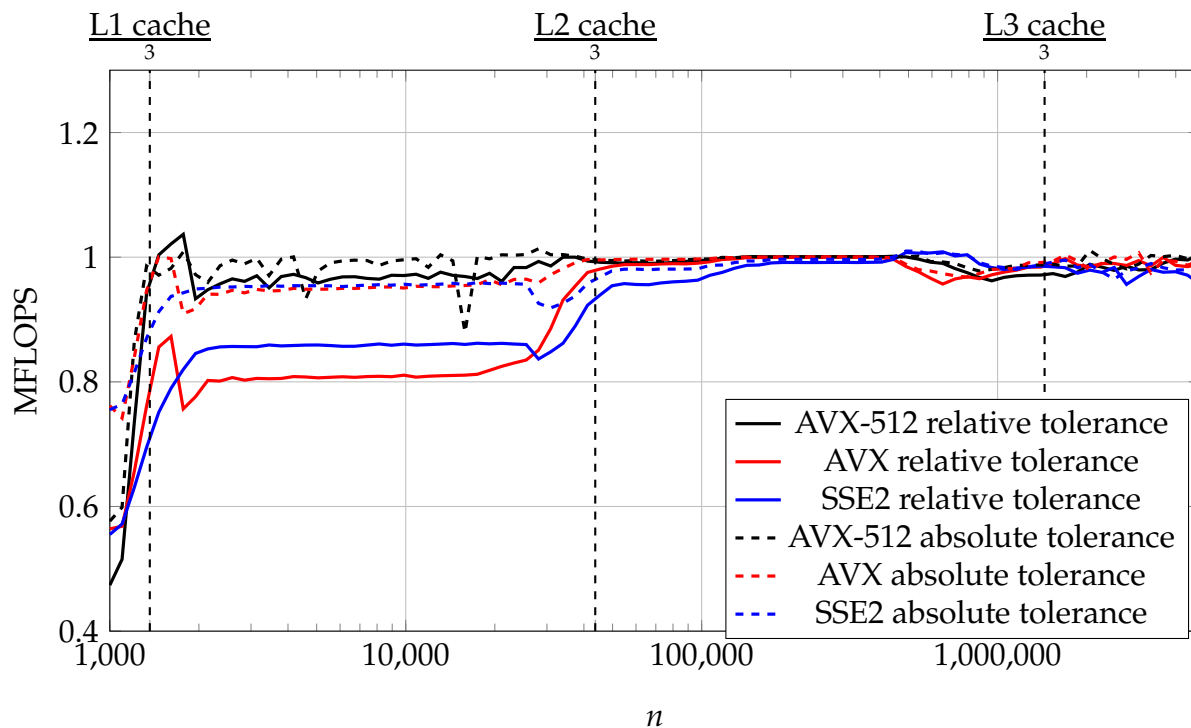
Figure C.24: Performance ratios relative to the naive versions, with 3 vectors, on Xeon Platinum
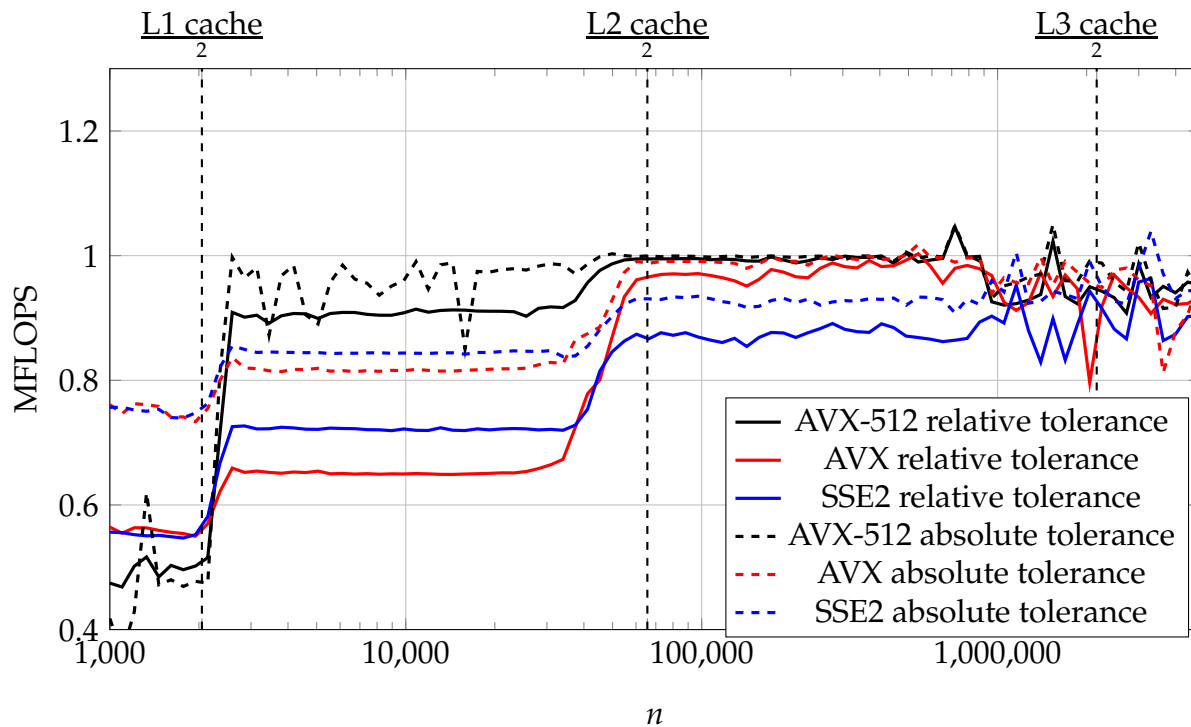
Figure C.25: Performance ratios relative to the naive versions, with 2 vectors, on Xeon Platinum

C.2.2 Dot product
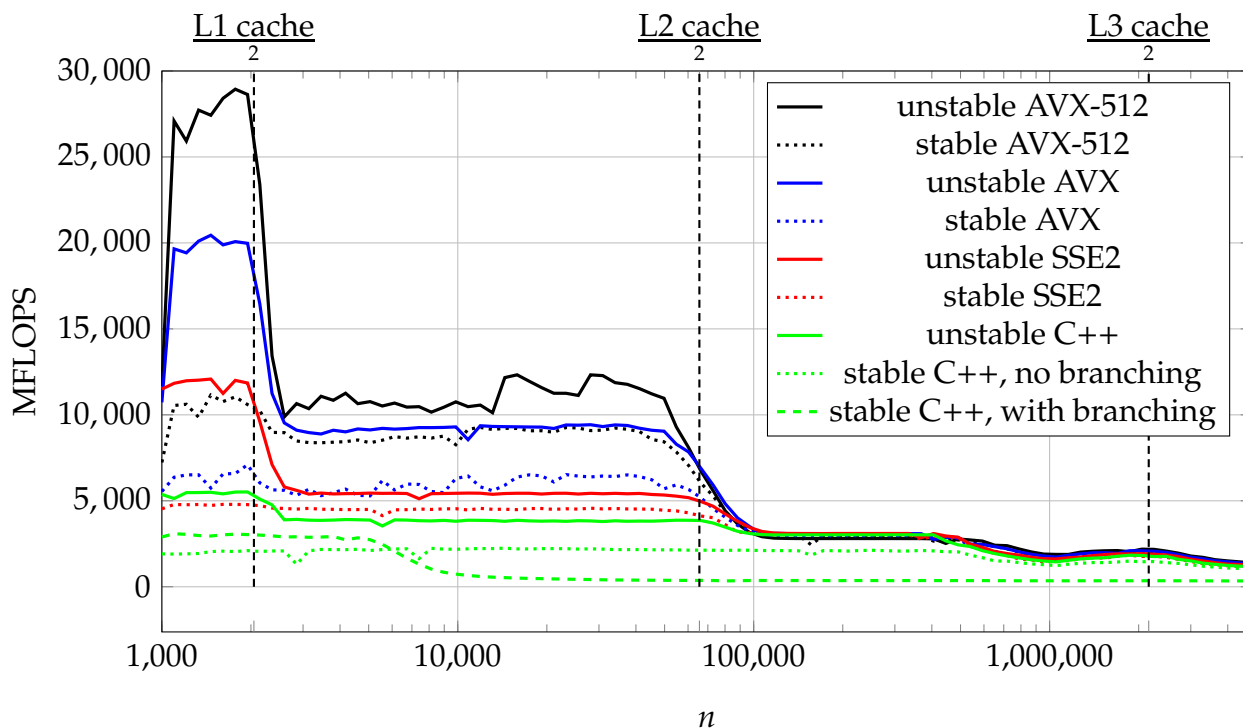
Figure C.26: Performances of the dot product implementations

Appendix D

Orchard-Hays and simplified adder tests

Name	Orchard-Hays		Simplified		Speedup	Iteration speedup
	Iterations	Time	Iterations	Time		
cre-a.mps	9904	2.612	9904	2.425	1.077	
cre-b.mps	119589	441.7	119589	422.6	1.045	
cre-c.mps	11559	3.02	11559	2.844	1.062	
cre-d.mps	100722	349.3	100722	341.8	1.022	
ken-07.mps	3219	0.6797	3219	0.6714	1.012	
ken-11.mps	22502	49.21	22502	47.77	1.03	
ken-13.mps	101266	574.2	101266	550.2	1.044	
osa-07.mps	889	0.559	889	0.506	1.105	
osa-14.mps	2115	4.188	2115	3.974	1.054	
osa-30.mps	3967	18.4	3967	16.83	1.093	1.093
osa-60.mps	9118	101.8	9118	99.57	1.022	1.022
pds-02.mps	2805	0.6491	2807	0.5895	1.101	1.102
pds-06.mps	11660	14.04	11230	12.58	1.116	1.075
pds-10.mps	22290	50.13	22746	49.24	1.018	1.039
pds-20.mps	81197	464.4	82198	459.2	1.011	1.024
25FV47.SIF	10675	3.088	10257	2.783	1.11	1.066
80BAU3B.SIF	4086	1.2	4086	1.117	1.075	1.075
ADLITTLE.SIF	155	0.002995	155	0.002878	1.041	
AFIRO.SIF	20	0.000426	20	0.000421	1.012	

Name	Orchard-Hays		Simplified		Speedup	Iteration speedup
	Iterations	Time	Iterations	Time		
AGG2.SIF	164	0.006222	164	0.005799	1.073	1.073
AGG3.SIF	173	0.006659	173	0.006318		1.054
BANDM.SIF	746	0.06559	746	0.06018		1.09
BEACONFD.SIF	164	0.004283	164	0.003961		1.081
BLEND.SIF	93	0.002064	93	0.001975		1.045
BNL1.SIF	3446	0.3803	3446	0.3536		1.075
BNL2.SIF	6398	1.53	6398	1.465		1.044
BOEING1.SIF	555	0.0299	555	0.02752		1.086
BOEING2.SIF	183	0.004508	183	0.004328	1.042	1.042
BORE3D.SIF	208	0.006667	208	0.006264	1.064	1.064
BRANDY.SIF	466	0.02313	466	0.02168		1.067
CAPRI.SIF	288	0.009483	288	0.008992		1.055
CYCLE.SIF	14034	5.197	15831	6.539	0.7948	0.8966
CZPROB.SIF	2336	0.2981	2336	0.2755		1.082
D2Q06C.SIF	26402	25.96	26049	24.59	1.056	1.042
D6CUBE.SIF	1100	0.5695	1100	0.5176	1.1	1.1
DEGEN2.SIF	1124	0.1501	1124	0.1408		1.066
DEGEN3.SIF	9201	6.62	9201	6.483		1.021
E226.SIF	708	0.03798	708	0.03607	1.053	1.053
ETAMACRO.SIF	980	0.05082	980	0.04842		1.05
FFFFF800.SIF	1350	0.09265	1350	0.08833		1.049
FINNIS.SIF	368	0.01399	368	0.0129	1.085	1.085
FIT1D.SIF	96	0.009433	96	0.00885		1.066
FIT1P.SIF	1629	0.3405	1629	0.3238		1.052
FIT2D.SIF	202	0.1781	202	0.1716		1.038
FIT2P.SIF	8191	14.44	8191	13.92		1.037
FORPLAN.SIF	395	0.01816	395	0.01715		1.059
GANGES.SIF	1356	0.1239	1356	0.1142		1.085
GFRD-PNC.SIF	544	0.0305	544	0.02775		1.099
GREENBEA.SIF	8757	3.579	9299	3.747	0.9551	1.014

Name	Orchard-Hays		Simplified		Speedup	Iteration speedup
	Iterations	Time	Iterations	Time		
GREENBEB.SIF	14329	6.725	14329	6.38		1.054
GROW15.SIF	4854	0.7719	4854	0.7443		1.037
GROW22.SIF	9835	2.497	9835	2.41	1.036	1.036
GROW7.SIF	1465	0.1231	1465	0.1154		1.067
ISRAEL.SIF	493	0.0237	497	0.02304	1.029	1.037
KB2.SIF	84	0.001655	84	0.001612		1.027
LOTFI.SIF	319	0.009172	319	0.008587		1.068
MAROS-R7.SIF	4486	24.02	4486	23.73	1.012	1.012
MAROS.SIF	7231	1.519	7231	1.452		1.046
MODSZK1.SIF	731	0.05513	731	0.0515		1.07
NESM.SIF	2339	0.2862	2698	0.2992	0.9565	1.103
PEROLD.SIF	4167	1.454	4129	1.36	1.069	1.059
PILOT-JA.SIF	5509	3.491	5509	3.324		1.05
PILOT-WE.SIF	7061	2.299	7061	2.213	1.039	1.039
PILOT.SIF	8682	34.86	9317	36.49	0.9554	1.025
PILOT4.SIF	1286	0.2913	1286	0.2733		1.066
PILOTNOV.SIF	1626	0.6002	1626	0.5753		1.043
QAP8.SIF	19217	32.9	19217	30.73		1.071
RECIPELP.SIF	48	0.000917	48	0.000868		1.056
SC105.SIF	122	0.003558	122	0.003443		1.033
SC205.SIF	215	0.00927	215	0.009001	1.03	1.03
SC50A.SIF	50	0.001119	50	0.001108		1.01
SC50B.SIF	49	0.001094	49	0.001077		1.016
SCAGR25.SIF	631	0.02981	631	0.02739		1.088
SCAGR7.SIF	208	0.004697	208	0.004605		1.02
SCFXM1.SIF	551	0.02351	551	0.02295		1.024
SCFXM2.SIF	1133	0.08426	1133	0.08114		1.038
SCFXM3.SIF	1759	0.1828	1759	0.1713		1.067
SCORPION.SIF	362	0.01213	362	0.01168		1.039
SCRS8.SIF	742	0.03953	742	0.03743		1.056

Name	Orchard-Hays		Simplified		Speedup	Iteration speedup
	Iterations	Time	Iterations	Time		
SCSD1.SIF	136	0.005385	136	0.004923		1.094
SCSD6.SIF	400	0.02351	400	0.02179		1.079
SCSD8.SIF	3113	0.4598	3113	0.4246		1.083
SCTAP1.SIF	449	0.01477	449	0.01371		1.078
SCTAP2.SIF	907	0.06805	907	0.06354		1.071
SCTAP3.SIF	1234	0.1239	1234	0.1106		1.121
SEBA.SIF	439	0.01728	439	0.01595	1.083	1.083
SHARE1B.SIF	391	0.01431	391	0.01366		1.048
SHARE2B.SIF	195	0.004642	195	0.004623		1.004
SHELL.SIF	599	0.03849	599	0.03417	1.127	1.127
SHIP04L.SIF	378	0.01864	378	0.01641		1.136
SHIP04S.SIF	381	0.01569	381	0.01422	1.103	1.103
SHIP08L.SIF	685	0.06218	685	0.05522		1.126
SHIP08S.SIF	647	0.0439	647	0.03897	1.126	1.126
SHIP12L.SIF	1148	0.1389	1148	0.124		1.12
SHIP12S.SIF	1089	0.09329	1089	0.0856	1.09	1.09
SIERRA.SIF	626	0.04867	626	0.04405		1.105
STAIR.SIF	462	0.1173	462	0.1119		1.048
STANDATA.SIF	165	0.006076	165	0.005433		1.118
STANDGUB.SIF	132	0.004834	132	0.004408	1.097	1.097
STANDMPS.SIF	295	0.01164	295	0.01069		1.09
STOCFOR1.SIF	99	0.002147	99	0.002175		0.9871
STOCFOR2.SIF	2249	0.4302	2249	0.4027		1.068
STOCFOR3.SIF	17929	41.79	17929	40.51		1.032
TRUSS.SIF	7739	4.713	7739	4.46		1.057
TUFF.SIF	626	0.03461	626	0.03313		1.045
VTP-BASE.SIF	346	0.01107	346	0.0106	1.044	1.044
WOOD1P.SIF	625	0.2672	625	0.2601		1.028
WOODW.SIF	2953	0.9926	2953	0.9234		1.075

Appendix E

Some low-level vector implementations

E.1 SSE2

```
1  template <class WRITER>
2  void denseToDenseAddRelSSE2_temp(
3      const double * __restrict__ a,
4      const double * __restrict__ b,
5      double * c,
6      size_t count, double lambda,
7      double relTolerance) {
8      const size_t rem1 = count % 4;
9      size_t i;
10     __m128d tolerance = _mm_set_pd1(relTolerance);
11     uint64_t mask = 0x7fffffffffffffffULL;
12     __m128d absMask = _mm_set1_pd(
13         *reinterpret_cast<double*>(&mask)  );
14     __m128d mul = _mm_set1_pd(lambda);
15     for (i = 0; i < count - rem1; i += 4) {
16         // c = a + b * lambda
17         __m128d dataA1 = _mm_load_pd(&a[i]);
18         __m128d dataB1 = _mm_load_pd(&b[i]);
19         __m128d dataC1 = _mm_add_pd(dataA1,
20                              _mm_mul_pd(dataB1, mul));
21         dataA1 = _mm_and_pd(dataA1, absMask);
```

```
22        __m128d absC1 = _mm_and_pd(dataC1, absMask);
23        dataA1 = _mm_mul_pd(dataA1, tolerance);
24        __m128d lessResult1 = _mm_cmplt_pd(dataA1, absC1);
25        dataC1 = _mm_and_pd(dataC1, lessResult1);
26        WRITER::write(c + i, dataC1);
27
28        __m128d dataA2 = _mm_load_pd(&a[i+2]);
29        __m128d dataB2 = _mm_load_pd(&b[i+2]);
30        __m128d dataC2 = _mm_add_pd(dataA2,
31                                    _mm_mul_pd(dataB2, mul));
32        dataA2 = _mm_and_pd(dataA2, absMask);
33        __m128d absC2 = _mm_and_pd(dataC2, absMask);
34        dataA2 = _mm_mul_pd(dataA2, tolerance);
35        __m128d lessResult2 = _mm_cmplt_pd(dataA2, absC2);
36        dataC2 = _mm_and_pd(dataC2, lessResult2);
37        WRITER::write(c + i + 2, dataC2);
38    }
39    for (; i < count; i++) {
40        __m128d dataA = _mm_set1_pd(a[i]);
41        __m128d dataB = _mm_set1_pd(b[i]);
42        __m128d dataC = _mm_add_pd(dataA,
43                                    _mm_mul_pd(dataB, mul));
44        dataA = _mm_and_pd(dataA, absMask);
45        __m128d absC = _mm_and_pd(dataC, absMask);
46        dataA = _mm_mul_pd(dataA, tolerance);
47        __m128d lessResult = _mm_cmplt_pd(dataA, absC);
48        dataC = _mm_and_pd(dataC, lessResult);
49        c[i] = dataC[0];
50    }
51 }
52
53 extern "C" void denseToDenseAddRelSSE2_cache(
54    const double * __restrict__ a,
```

```
55      const double * __restrict__ b,
56      double * c,
57      size_t count, double lambda,
58      double relTolerance) {
59      struct CACHE_WRITER {
60          inline static void write(double * address,
61              __m128d & value) {
62              _mm_store_pd(address, value);
63          }
64      };
65      denseToDenseAddRelSSE2_temp<CACHE_WRITER>(a, b, c,
66                                              count,
67                                              lambda,
68                                              relTolerance);
69 }
70
71 extern "C" void denseToDenseAddRelSSE2_nocache(
72      const double * __restrict__ a,
73      const double * __restrict__ b,
74      double * c,
75      size_t count, double lambda,
76      double relTolerance) {
77      struct NOCACHE_WRITER {
78          inline static void write(double * address,
79              __m128d & value) {
80              _mm_stream_pd(address, value);
81          }
82      };
83      denseToDenseAddRelSSE2_temp<NOCACHE_WRITER>(a, b, c,
84          count, lambda, relTolerance);
85 }
86
87 double denseToDenseDotProductStableSSE2(
```

```
88      const double * __restrict__ a,
89      const double * __restrict__ b,
90      size_t count) {
91      const size_t rem1 = count % 4;
92      size_t i = 0;
93      __m128d neg1 = {0, 0};
94      __m128d pos1 = {0, 0};
95      __m128d neg2 = {0, 0};
96      __m128d pos2 = {0, 0};
97      __m128d neg3 = {0, 0};
98      __m128d pos3 = {0, 0};
99      __m128d neg4 = {0, 0};
100     __m128d pos4 = {0, 0};
101     const __m128d zero = {0, 0};
102     for (i = 0; i < count - rem1; i += 4) {
103         // c = a + b * lambda
104         __m128d dataA1 = _mm_load_pd(&a[i]);
105         __m128d dataB1 = _mm_load_pd(&b[i]);
106         __m128d mul1 = _mm_mul_pd(dataA1, dataB1);
107         __m128d lessResult1 = _mm_cmplt_pd(zero, mul1);
108         neg1 = _mm_add_pd(neg1, _mm_and_pd(lessResult1, mul1));
109         pos1 = _mm_add_pd(pos1,
110                         _mm_andnot_pd(lessResult1, mul1));
111
112         __m128d dataA2 = _mm_load_pd(&a[i + 2]);
113         __m128d dataB2 = _mm_load_pd(&b[i + 2]);
114         __m128d mul2 = _mm_mul_pd(dataA2, dataB2);
115         __m128d lessResult2 = _mm_cmplt_pd(zero, mul2);
116         neg2 = _mm_add_pd(neg2, _mm_and_pd(lessResult2, mul2));
117         pos2 = _mm_add_pd(pos2,
118                         _mm_andnot_pd(lessResult2, mul2));
119     }
120     const size_t evenCount = count - (count % 2);
```

```
121      for (; i < evenCount; i += 2) {
122          __m128d dataA = _mm_load_pd(&a[i]);
123          __m128d dataB = _mm_load_pd(&b[i]);
124          __m128d mul = _mm_mul_pd(dataA, dataB);
125          __m128d lessResult = _mm_cmplt_pd(zero, mul);
126          neg1 = _mm_add_pd(neg1, _mm_and_pd(lessResult, mul));
127          pos1 = _mm_add_pd(pos1, _mm_andnot_pd(lessResult, mul));
128      }
129
130      neg1 = _mm_add_pd(neg1, neg2);
131      neg3 = _mm_add_pd(neg3, neg4);
132      neg1 = _mm_add_pd(neg1, neg3);
133
134      pos1 = _mm_add_pd(pos1, pos2);
135      pos3 = _mm_add_pd(pos3, pos4);
136      pos1 = _mm_add_pd(pos1, pos3);
137
138      __m128d neg = {0, 0};
139      __m128d pos = {0, 0};
140      for (; i < count; i++) {
141          __m128d dataA = _mm_set_pd1(a[i]);
142          __m128d dataB = _mm_set_pd1(b[i]);
143          __m128d mul = _mm_mul_pd(dataA, dataB);
144          __m128d lessResult = _mm_cmplt_pd(zero, mul);
145          neg = _mm_add_pd(neg, _mm_and_pd(lessResult, mul));
146          pos = _mm_add_pd(pos, _mm_andnot_pd(lessResult, mul));
147      }
148      return neg1[0] + neg1[1] + neg[0] +
149              pos1[0] + pos1[1] + pos[0];
150 }
```

E.2 AVX

```
template <class WRITER>
void denseToDenseAddRelAVX_temp(const double * __restrict__ a,
                                const double * __restrict__ b,
                                double * c,
                                size_t count, double lambda,
                                double relTolerance) {
    const size_t rem1 = count % 8;
    size_t i;
    __m256d tolerance = _mm256_set1_pd(relTolerance);
    uint64_t mask = 0x7fffffffffffffffULL;
    __m256d absMask = _mm256_set1_pd(
                        *reinterpret_cast<double*>(&mask) );
    __m256d mul = _mm256_broadcast_sd(&lambda);
    for (i = 0; i < count - rem1; i += 8) {
        // c = a + b * lambda
        __m256d dataA1 = _mm256_load_pd(&a[i]);
        __m256d dataB1 = _mm256_load_pd(&b[i]);
        __m256d dataC1 = _mm256_add_pd(dataA1,
                        _mm256_mul_pd(dataB1, mul ));
        dataA1 = _mm256_and_pd(dataA1, absMask);
        __m256d absC1 = _mm256_and_pd(dataC1, absMask);
        dataA1 = _mm256_mul_pd(dataA1, tolerance);
        __m256d lessResult1 = _mm256_cmp_pd(dataA1,
                                        absC1, _CMP_LT_OS);
        dataC1 = _mm256_and_pd(dataC1, lessResult1);
        WRITER::write(c + i, dataC1);

        __m256d dataA2 = _mm256_load_pd(&a[i+4]);
        __m256d dataB2 = _mm256_load_pd(&b[i+4]);
        __m256d dataC2 = _mm256_add_pd(dataA2,
                        _mm256_mul_pd(dataB2, mul ));
        dataA2 = _mm256_and_pd(dataA2, absMask);
```

```cpp
33        __m256d absC2 = _mm256_and_pd(dataC2, absMask);
34        dataA2 = _mm256_mul_pd(dataA2, tolerance);
35        __m256d lessResult2 = _mm256_cmp_pd(dataA2,
36                                        absC2, _CMP_LT_OS);
37        dataC2 = _mm256_and_pd(dataC2, lessResult2);
38        WRITER::write(c + i + 4, dataC2);
39    }
40    for (; i < count; i++) {
41        __m256d dataA = _mm256_set1_pd(a[i]);
42        __m256d dataB = _mm256_set1_pd(b[i]);
43        __m256d dataC = _mm256_add_pd(dataA,
44                    _mm256_mul_pd(dataB, mul ));
45        dataA = _mm256_and_pd(dataA, absMask);
46        __m256d absC = _mm256_and_pd(dataC, absMask);
47        dataA = _mm256_mul_pd(dataA, tolerance);
48        __m256d lessResult = _mm256_cmp_pd(dataA,
49                                        absC, _CMP_LT_OS);
50        dataC = _mm256_and_pd(dataC, lessResult);
51        c[i] = dataC[0];
52    }
53 }
54
55 extern "C" void denseToDenseAddRelAVX_cache(
56                                const double * __restrict__ a,
57                                const double * __restrict__ b,
58                                double * c,
59                                size_t count, double lambda,
60                                double relTolerance) {
61    struct CACHE_WRITER {
62        inline static void write(double * address,
63            __m256d & value) {
64            _mm256_store_pd(address, value);
65        }
```

```
66          };
67          denseToDenseAddRelAVX_temp<CACHE_WRITER>(a, b, c, count,
68                                                   lambda,
69                                                   relTolerance);
70  }
71
72  extern "C" void denseToDenseAddRelAVX_nocache(
73                                   const double * __restrict__ a,
74                                   const double * __restrict__ b,
75                                   double * c,
76                                   size_t count, double lambda,
77                                   double relTolerance) {
78          struct NOCACHE_WRITER {
79              inline static void write(double * address,
80                  __m256d & value) {
81                  _mm256_stream_pd(address, value);
82              }
83          };
84          denseToDenseAddRelAVX_temp<NOCACHE_WRITER>(a, b, c, count,
85                                                     lambda,
86                                                     relTolerance);
87  }
88  double denseToDenseDotProductStableAVX(
89          const double * __restrict__ a,
90          const double * __restrict__ b,
91          size_t count) {
92      const size_t rem1 = count % 8;
93      size_t i = 0;
94      __m256d neg1 = {0, 0, 0, 0};
95      __m256d pos1 = {0, 0, 0, 0};
96      __m256d neg2 = {0, 0, 0, 0};
97      __m256d pos2 = {0, 0, 0, 0};
98      __m256d neg3 = {0, 0, 0, 0};
```

```
99        __m256d pos3 = {0, 0, 0, 0};
100       __m256d neg4 = {0, 0, 0, 0};
101       __m256d pos4 = {0, 0, 0, 0};
102       const __m256d zero = {0, 0, 0, 0};
103       for (i = 0; i < count - rem1; i += 8) {
104           // c = a + b * lambda
105           __m256d dataA1 = _mm256_load_pd(&a[i]);
106           __m256d dataB1 = _mm256_load_pd(&b[i]);
107           __m256d mul1 = _mm256_mul_pd(dataA1, dataB1);
108           __m256d lessResult1 = _mm256_cmp_pd(zero,
109                                   mul1, _CMP_LT_OS);
110           neg1 = _mm256_add_pd(neg1, _mm256_and_pd(
111                                   lessResult1, mul1));
112           pos1 = _mm256_add_pd(pos1, _mm256_andnot_pd(
113                                   lessResult1, mul1));
114
115           __m256d dataA2 = _mm256_load_pd(&a[i + 4]);
116           __m256d dataB2 = _mm256_load_pd(&b[i + 4]);
117           __m256d mul2 = _mm256_mul_pd(dataA2, dataB2);
118           __m256d lessResult2 = _mm256_cmp_pd(zero,
119                                       mul2, _CMP_LT_OS);
120           neg2 = _mm256_add_pd(neg2, _mm256_and_pd(
121                                   lessResult2, mul2));
122           pos2 = _mm256_add_pd(pos2, _mm256_andnot_pd(
123                                   lessResult2, mul2));
124       }
125       const size_t secondCount = count - (count % 4);
126       for (; i < secondCount; i += 4) {
127           __m256d dataA = _mm256_load_pd(&a[i]);
128           __m256d dataB = _mm256_load_pd(&b[i]);
129           __m256d mul = _mm256_mul_pd(dataA, dataB);
130           __m256d lessResult = _mm256_cmp_pd(zero,
131                                       mul, _CMP_LT_OS);
```

```
132        neg1 = _mm256_add_pd(neg1, _mm256_and_pd(
133                                   lessResult, mul));
134        pos1 = _mm256_add_pd(pos1, _mm256_andnot_pd(
135                                   lessResult, mul));
136    }
137
138    neg1 = _mm256_add_pd(neg1, neg2);
139    neg3 = _mm256_add_pd(neg3, neg4);
140    neg1 = _mm256_add_pd(neg1, neg3);
141    pos1 = _mm256_add_pd(pos1, pos2);
142    pos3 = _mm256_add_pd(pos3, pos4);
143    pos1 = _mm256_add_pd(pos1, pos3);
144
145    __m256d neg = {0, 0, 0, 0};
146    __m256d pos = {0, 0, 0, 0};
147    for (; i < count; i++) {
148        __m256d dataA = _mm256_set1_pd(a[i]);
149        __m256d dataB = _mm256_set1_pd(b[i]);
150        __m256d mul = _mm256_mul_pd(dataA, dataB);
151        __m256d lessResult = _mm256_cmp_pd(zero,
152                                       mul, _CMP_LT_OS);
153        neg = _mm256_add_pd(neg, _mm256_and_pd(lessResult, mul));
154        pos = _mm256_add_pd(pos, _mm256_andnot_pd(
155                                   lessResult, mul));
156    }
157    return neg1[0] + neg1[1] + neg1[2] + neg1[3] + neg[0]
158            + pos1[0] + pos1[1] + pos1[2] + pos1[3] + pos[0];
159 }
```

E.3 AVX-512

```
template <class WRITER>
void denseToDenseAddRelAVX512_temp(const double * __restrict__ a,
                                   const double * __restrict__ b,
                                   double * c,
                                   size_t count, double lambda,
                                   double relTolerance) {
    const size_t rem1 = count % 16;
    size_t i;
    __m512d tolerance = _mm512_set1_pd(relTolerance);
    __m512d zero = _mm512_set1_pd(0.0);
    __m512d mul = _mm512_set1_pd(lambda);
    for (i = 0; (i + 0) < count - rem1; i += 16) {
        // c = a + b * lambda
        __m512d dataA1 = _mm512_load_pd(&a[i]);
        __m512d dataB1 = _mm512_load_pd(&b[i]);
        __m512d dataC1 = _mm512_add_pd(dataA1,
                        _mm512_mul_pd(dataB1, mul  ));
        dataA1 = _mm512_abs_pd(dataA1);
        __m512d absC1 = _mm512_abs_pd(dataC1);
        dataA1 = _mm512_mul_pd(dataA1, tolerance);
        __mmask8 lessResult1 = _mm512_cmp_pd_mask(dataA1,
                                absC1, _CMP_LT_OS);
        dataC1 = _mm512_mask_mov_pd(zero, lessResult1, dataC1);
        WRITER::write(c + i, dataC1);

        __m512d dataA2 = _mm512_load_pd(&a[i + 8]);
        __m512d dataB2 = _mm512_load_pd(&b[i + 8]);
        __m512d dataC2 = _mm512_add_pd(dataA2,
                        _mm512_mul_pd(dataB2, mul ));
        dataA2 = _mm512_abs_pd(dataA2);
        __m512d absC2 = _mm512_abs_pd(dataC2);
        dataA2 = _mm512_mul_pd(dataA2, tolerance);
```

```
33        __mmask8 lessResult2 =
34                    _mm512_cmp_pd_mask(dataA2, absC2, _CMP_LT_OS);
35            dataC2 = _mm512_mask_mov_pd(zero, lessResult2, dataC2);
36            WRITER::write(c + i + 8, dataC2);
37        }
38    for (; i < count; i++) {
39            __m512d dataA = _mm512_set1_pd(a[i]);
40            __m512d dataB = _mm512_set1_pd(b[i]);
41            __m512d dataC = _mm512_add_pd(dataA,
42                                            _mm512_mul_pd(dataB, mul));
43            dataA = _mm512_abs_pd(dataA);
44            __m512d absC = _mm512_abs_pd(dataC);
45            dataA = _mm512_mul_pd(dataA, tolerance);
46            __mmask8 lessResult =
47                    _mm512_cmp_pd_mask(dataA, absC, _CMP_LT_OS);
48            dataC = _mm512_mask_mov_pd(zero, lessResult, dataC);
49            c[i] = dataC[0];
50        }
51 }
52
53 extern "C" void denseToDenseAddRelAVX512_cache(
54                                    const double * __restrict__ a,
55                                    const double * __restrict__ b,
56                                    double * c,
57                                    size_t count, double lambda,
58                                    double relTolerance) {
59    struct CACHE_WRITER {
60        inline static void write(double * address,
61            __m512d & value) {
62            _mm512_store_pd(address, value);
63        }
64    };
65    denseToDenseAddRelAVX512_temp<CACHE_WRITER>(a, b, c, count,
```

```
66                                                   lambda,
67                                                   relTolerance);
68  }
69
70  extern "C" void denseToDenseAddRelAVX512_nocache(
71                                   const double * __restrict__ a,
72                                   const double * __restrict__ b,
73                                   double * c,
74                                   size_t count, double lambda,
75                                   double relTolerance) {
76      struct NOCACHE_WRITER {
77          inline static void write(double * address,
78              __m512d & value) {
79              _mm512_stream_pd(address, value);
80          }
81      };
82      denseToDenseAddRelAVX512_temp<NOCACHE_WRITER>(a, b, c, count,
83                                                  lambda,
84                                                  relTolerance);
85  }
86
87  double denseToDenseDotProductStableAVX512(
88          const double * __restrict__ a,
89          const double * __restrict__ b,
90          size_t count) {
91      const size_t rem1 = count % 16;
92      size_t i = 0;
93      __m512d neg1 = {0, 0, 0, 0, 0, 0, 0, 0};
94      __m512d pos1 = {0, 0, 0, 0, 0, 0, 0, 0};
95      __m512d neg2 = {0, 0, 0, 0, 0, 0, 0, 0};
96      __m512d pos2 = {0, 0, 0, 0, 0, 0, 0, 0};
97      const static __m512d zero = {0, 0, 0, 0, 0, 0, 0, 0};
98      for (i = 0; i < count - rem1; i += 16) {
```

```
99          // c = a + b * lambda
100         __m512d dataA1 = _mm512_load_pd(&a[i]);
101         __m512d dataB1 = _mm512_load_pd(&b[i]);
102         __m512d mul1 = _mm512_mul_pd(dataA1, dataB1);
103         __mmask8 lessResult1 = _mm512_cmp_pd_mask(
104                                     zero, mul1, _CMP_LT_OS);
105         neg1 = _mm512_mask_add_pd(neg1, lessResult1, neg1, mul1);
106         pos1 = _mm512_mask_add_pd(pos1, ~lessResult1, pos1,
107                                     mul1);
108
109         __m512d dataA2 = _mm512_load_pd(&a[i + 8]);
110         __m512d dataB2 = _mm512_load_pd(&b[i + 8]);
111         __m512d mul2 = _mm512_mul_pd(dataA2, dataB2);
112         __mmask8 lessResult2 = _mm512_cmp_pd_mask(
113                                     zero, mul2, _CMP_LT_OS);
114         neg2 = _mm512_mask_add_pd(neg2, lessResult2, neg2, mul2);
115         pos2 = _mm512_mask_add_pd(pos2, ~lessResult2, pos2,
116                                     mul2);
117     }
118     const size_t secondCount = count - (count % 8);
119     for (; i < secondCount; i += 8) {
120         __m512d dataA = _mm512_load_pd(&a[i]);
121         __m512d dataB = _mm512_load_pd(&b[i]);
122         __m512d mul = _mm512_mul_pd(dataA, dataB);
123         __mmask8 lessResult = _mm512_cmp_pd_mask(
124                                     zero, mul, _CMP_LT_OS);
125         neg1 = _mm512_mask_add_pd(neg1, lessResult, neg1, mul);
126         pos1 = _mm512_mask_add_pd(pos1, ~lessResult, pos1, mul);
127     }
128
129     neg1 = _mm512_add_pd(neg1, neg2);
130     pos1 = _mm512_add_pd(pos1, pos2);
131
```

```
132    for (; i < count; i++) {
133        __m512d dataA = _mm512_set1_pd(a[i]);
134        __m512d dataB = _mm512_set1_pd(b[i]);
135        __m512d mul = _mm512_mul_pd(dataA, dataB);
136        __mmask8 lessResult = _mm512_cmp_pd_mask(
137                              zero, mul, _CMP_LT_OS);
138        neg1 = _mm512_mask_add_pd(
139                    neg1, lessResult & 1, neg1, mul);
140        pos1 = _mm512_mask_add_pd(
141                    pos1, (~lessResult) & 1, pos1, mul);
142    }
143    double posScalar = _mm512_reduce_add_pd(pos1);
144    double negScalar = _mm512_reduce_add_pd(neg1);
145    return posScalar + negScalar;
146 }
```

Appendix F

Abstract in Japanese

シンプレックス法は、線型計画問題(LO)を解くために使用される最重要アルゴリズムの１つです。1950年以降シンプレックス法は、コンピューターハードウェアやアルゴリズムの開発と密接に関わりながら発展してきました。初期バージョンでは、小規模な問題しか解決できませんでしたが、現在のソフトウェアでは、膨大な数の決定変数や制限があっても問題を処理できることがあります。ですが、大規模な最適化モデルの解決は、いくつかの理由により困難なことがあります。一部の理由として、ソリューションアルゴリズムで必要な浮動小数点演算を使わざるを得ないことが挙げられます。これらの問題を軽減する大きな進歩がありましたが、解決の妨げとなるモデルに直面することはいまだ珍しくありません。また、スケーリング、摂動、非退化方法などのさまざまな数値的困難を伴うモデルに対処するための目覚ましい進歩もありました。これら方法によってタスク全体の完了が大きく遅れないことが重要です。ですが、従来の数表現では処理できない数字的なエラーがあるため、現在のソフトウェアであってもすべての線型計画問題を正しく解決することはできません。既存のソフトウェアを使っても、数値的困難のため、あるいは最適化に収束できないため、問題の解決に至らないといったような残念な結果になることがあります。ですがこれがきっかけとなり、デフォルトの浮動小数点数表現では特定のモデルが解決できないことを検出できる自動機能を、弊社のソフトウェアであるPannon Optimizerに組み込むことになりましたので、より正確な演算に切り替える価値はあります。プログラム出力の正確性に対する疑念も軽減されるのでユーザーにとっては大きな助けとなります。また、条件分岐を含まない安定した加算器を開発し、ドット積を実装したため、計算のオーバーヘッドが低くなります。さらに、シンプレックス法の時間のかかる初歩的な手順の１つである加速法についても説明いたします。数値アルゴリズムに関して議論する際は、速度効率に関する情報もご提供します。

Bibliography

[1] P. Alonso, J. Delgado, R. Gallego, and J. M. Peña. "Conditioning and accurate computations with Pascal matrices". In: *Journal of Computational and Applied Mathematics* 252 (2013). Selected papers on Computational and Mathematical Methods in Science and Engineering (CMMSE), pp. 21–26. ISSN: 0377-0427.

[2] Á. V. Álvarez. "High-performance decimal floating point units". In: 2009.

[3] E. Andersen and K. Andersen. "Presolving in Linear Programming". In: *Mathematical Programming* 71.2 (1995), pp. 221–245.

[4] M. Benichou, J. M. Gauthier, G. Hentges, and G. Ribiere. "The Efficient Solution of Large-scale Linear Programming Problems–some Algorithmic Techniques and Computational Results". In: *Math. Program.* 13.1 (Dec. 1977), pp. 280–322. ISSN: 0025-5610.

[5] G. Brearley A. Mitra and W. H. "Analysis of Mathematical Programming Problems Prior to Applying the Simplex Method". In: *Mathematical Programming* 8 (1975), pp. 54–83.

[6] R. P. Brent. "On the Precision Attainable with Various Floating-Point Number Systems". In: *IEEE Trans. Comput.* 22.6 (June 1973), pp. 601–607. ISSN: 0018-9340.

[7] F. Busaba, C. Krygowski, W. Li, E. M. Schwarz, and S. Carlough. "The IBM z900 decimal arithmetic unit". In: *Conference Record of Thirty-Fifth Asilomar Conference on Signals, Systems and Computers (Cat.No.01CH37256)* 2 (2001), 1335–1339 vol.2.

[8] P. E. Ceruzzi. "The Early Computers of Konrad Zuse, 1935 to 1945". In: *Annals of the History of Computing* 3 (1981), pp. 241–262.

[9] C. W. Clenshaw and F. W. J. Olver. "Beyond Floating Point". In: *J. ACM* 31.2 (Mar. 1984), pp. 319–328. ISSN: 0004-5411.

[10] W. J. Cody. "Static and Dynamic Numerical Characteristics of Floating-Point Arithmetic". In: *IEEE Transactions on Computers* C-22.6 (June 1973), pp. 598–601. ISSN: 2326-3814.

[11] M. Cornea, C. Anderson, J. Harrison, P. T. P. Tang, E. Schneider, and C. Tsen. "A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format". In: *IEEE Transactions on Computers* 58 (2007), pp. 148–162.

[12] M. F. Cowlishaw. "Decimal floating-point: algorism for computers". In: *Proceedings 2003 16th IEEE Symposium on Computer Arithmetic*. June 2003, pp. 104–111.

[13] M. F. Cowlishaw, E. M. Schwarz, R. M. Smith, and C. F. Webb. "A Decimal Floating-Point Specification". In: *15th IEEE Symposium on Computer Arithmetic (Arith-15 2001), 11-17 June 2001, Vail, CO, USA*. 2001, pp. 147–154.

[14] G. Dantzig. "Maximization of a linear function of variables subject to linear inequalities". In: *Activity analysis of production and allocation*. Ed. by T. Koopmans. New York: Wiley, 1951. Chap. XXI.

[15] G. Dantzig and W. Orchard-Hays. *Notes on linear programming: Part V. Alternative algorithm for the revised simplex method using product form of the inverse*. Research Memorandum RM-1268. The RAND Corporation, Nov. 1953.

[16] G. B. Dantzig. "Reminiscences about the Origins of Linear Programming". In: *Oper. Res. Lett.* 1.2 (Apr. 1982), pp. 43–48. ISSN: 0167-6377.

[17] G. B. Dantzig and W. Orchard-Hays. "The Product Form for the Inverse in the Simplex Method". In: *Mathematical Tables and Other Aids to Computation* 8.46 (1954), pp. 64–47.

[18] G. B. Dantzig and M. N. Thapa. *Linear Programming 1: Introduction*. Berlin, Heidelberg: Springer-Verlag, 1997. ISBN: 0387948333.

[19] T. J. Dekker. "A Floating-point Technique for Extending the Available Precision". In: *Numer. Math.* 18.3 (June 1971), pp. 224–242. ISSN: 0029-599X.

[20] J. Demmel. "Underflow and the Reliability of Numerical Software". In: *SIAM J. Sci. Stat. Comput.* 5.4 (Dec. 1984), pp. 887–919. ISSN: 0196-5204.

[21] R. Dorfman. "The Discovery of Linear Programming". In: *Annals of the History of Computing* 6.3 (July 1984), pp. 283–295. ISSN: 0164-1239.

[22] M. A. Erle, M. J. Schulte, and B. J. Hickmann. "Decimal Floating-Point Multiplication Via Carry-Save Addition". In: *18th IEEE Symposium on Computer Arithmetic (ARITH '07)*. June 2007, pp. 46–55.

[23] J. Farkas. "Theorie der einfachen Ungleichungen." In: *Journal für die reine und angewandte Mathematik* 124 (1902), pp. 1–27.

[24] R. Fourer, D. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Scientific Press series. Thomson/Brooks/Cole, 2003. ISBN: 9780534388096.

[25] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. "MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding". In: *ACM Trans. Math. Softw.* 33.2 (June 2007). ISSN: 0098-3500.

[26] L. Fox. "Computer Solution of Linear Algebraic Systems. By G. Forsythe and C. B. Moler. Pp. xi, 148. 1967. (Prentice-Hall.)" In: *The Mathematical Gazette* 53.384 (1969), pp. 221–222.

[27] D. Goldberg. "What Every Computer Scientist Should Know about Floating-Point Arithmetic". In: *ACM Comput. Surv.* 23.1 (Mar. 1991), pp. 5–48. ISSN: 0360-0300.

[28] D. Goldfarb and J. Reid. "A practicable steepest-edge simplex algorithm". In: *Mathematical Programming* 12 (1977), pp. 361–371.

[29] J. Gondzio. "Presolve Analysis of Linear Programs Prior to Applying an Interior Point Method". In: *INFORMS Journal on Computing* 9.1 (1997), pp. 73–91.

[30] P. Harris. "Pivot selection method of the DEVEX LP code". In: *Mathematical Programming* 5 (1973), pp. 1–28.

[31] M. Hassaballah, S. Omran, and Y. B. Mahdy. "A Review of SIMD Multimedia Extensions and their Usage in Scientific and Engineering Applications." In: *Comput. J.* 51.6 (Nov. 5, 2008), pp. 630–649.

[32] J. R. Hauser. "Handling Floating-Point Exceptions in Numeric Programs". In: *ACM Trans. Program. Lang. Syst.* 18.2 (Mar. 1996), pp. 139–174. ISSN: 0164-0925.

[33] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Second. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. xxx+680. ISBN: 0-89871-521-0.

[34] D. Hilbert. "Ein Beitrag zur Theorie des Legendre'schen Polynoms". In: *Acta Mathematica*. Vol. 18. 1. 1894, pp. 155–159.

[35] *IEEE standard for binary floating-point arithmetic*. Note: Standard 754–1985. New York: Institute of Electrical and Electronics Engineers, 1985.

[36] "IEEE Standard for Radix-Independent Floating-Point Arithmetic". In: *ANSI/IEEE Std 854-1987* (Oct. 1987), pp. 1–19. ISSN: null.

[37] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z*. Intel. June 2013.

[38] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A: System Programming Guide, Part 1*. Intel. June 2013.

[39] E. E. S. Jr. and A. G. Alexopoulos. "The Sign/Logarithm Number System". In: *IEEE Trans. Computers* 24.12 (1975), pp. 1238–1242.

[40] W. Kahan. "Pracniques: Further Remarks on Reducing Truncation Errors". In: *Commun. ACM* 8.1 (Jan. 1965), p. 40. ISSN: 0001-0782.

[41] W. Kahan. "Lecture Notes on the Status of IEEE-754". In: 1996.

[42] J. Kallrath. *Modeling Languages in Mathematical Optimization*. Applied Optimization. Springer US, 2004. ISBN: 9781402075476.

[43] L. Kantorovich. *Matematicheskie Metody Organizatsii i Planirovaniya*. 1939.

[44] N. Karmarkar. "A New Polynomial-Time Algorithm for Linear Programming". In: *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*. STOC '84. New York, NY, USA: Association for Computing Machinery, 1984, pp. 302–311. ISBN: 0897911334.

[45] L. Khachiyan. "Polynomial algorithms in linear programming". In: *USSR Computational Mathematics and Mathematical Physics* 20.1 (1980), pp. 53–72. ISSN: 0041-5553.

[46] N. G. Kingsbury and P. J. W. Rayner. "Digital filtering using logarithmic arithmetic". In: *Electronics Letters* 7.2 (Jan. 1971), pp. 56–58. ISSN: 0013-5194.

[47] V. Klee and G. J. Minty. "How Good is the Simplex Method". In: 1972.

[48] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-89684-2.

[49]   P. Kornerup and D. W. Matula. "Finite Precision Rational Arithmetic: An Arithmetic Unit". In: *IEEE Trans. Comput.* 32.4 (Apr. 1983), pp. 378–388. ISSN: 0018-9340.

[50]   P. Kornerup and D. W. Matula. "Finite precision lexicographic continued fraction number systems". In: *1985 IEEE 7th Symposium on Computer Arithmetic (ARITH)* (1985), pp. 207–213.

[51]   H. Kuki and W. J. Cody. "A Statistical Study of the Accuracy of Floating Point Number Systems". In: *Commun. ACM* 16.4 (Apr. 1973), pp. 223–230. ISSN: 0001-0782.

[52]   U. W. Kulisch. "Circuitry for generating scalar products and sums of floating-point numbers with maximum accuracy". 1986.

[53]   U. W. Kulisch. *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN: 3211838708.

[54]   J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra. "Exploiting the Performance of 32 Bit Floating Point Arithmetic in Obtaining 64 Bit Accuracy (Revisiting Iterative Refinement for Linear Systems)". In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06. Tampa, Florida: ACM, 2006. ISBN: 0-7695-2700-0.

[55]   C. Lemke. "The dual method of solving the linear programming problem". In: *Naval Research Logistics Quarterly* 1 (1954), pp. 36–47.

[56]   Liang-Kai Wang and M. J. Schulte. "Decimal floating-point division using Newton-Raphson iteration". In: *Proceedings. 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004*. Sept. 2004, pp. 84–95.

[57]   H. Markowitz. "The elimination form of the inverse and its applications to linear programming". In: *Management Science* 3 (1954), pp. 255–269.

[58]   I. Maros. "A general Phase-I method in linear programming". In: *European Journal of Operational Research* 23 (1986), pp. 64–77.

[59]   I. Maros. "A Piecewise Linear Dual Phase-1 Algorithm for the Simplex Method". In: *Computational Optimization and Applications* 26 (2003), pp. 63–81.

[60]   I. Maros. "A Piecewise Linear Dual Procedure in Mixed Integer Programming". In: *New Trends in Mathematical Programming*. Ed. by S. K. F. Giannesi and T. Rapcsák. Kluwer Academic Publishers, 1998, pp. 159–170.

[61]   I. Maros. *Computational techniques of the simplex method*. Kluwer Academic Publishers, 2003.

[62]   I. Maros and C. Mészáros. "A numerically exact implementation of the simplex method". English. In: *Annals of Operations Research* 58.1 (1995), pp. 1–17. ISSN: 0254-5330.

[63]   R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. USA: Prentice Hall PTR, 2008. ISBN: 0132350882.

[64]   I. MathWorks. *Symbolic Math Toolbox for Use with MATLAB: User's Guide*. MathWorks, Incorporated, 2005.

[65]   Matula and Kornerup. "Finite Precision Rational Arithmetic: Slash Number Systems". In: *IEEE Transactions on Computers* C-34.1 (Jan. 1985), pp. 3–18. ISSN: 2326-3814.

[66]   V. Menissier. *Arithmétique exacte: conception, algorithmique et performances d'une implémentation informatique en précision arbitraire*. 1994.

[67]   J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. 1st. Birkhäuser Basel, 2009. ISBN: 081764704X, 9780817647049.

[68]   J.-M. Muller, A. Tisserand, and A. Scherbyna. "Semi-Logarithmic Number Systems". In: *Proceedings of the 12th Symposium on Computer Arithmetic*. ARITH '95. USA: IEEE Computer Society, 1995, p. 201. ISBN: 0818670894.

[69]   A. Neumaier. "Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen". In: *Zeitschrift Angewandte Mathematik und Mechanik* 54.1 (Jan. 1974), pp. 39–51.

[70]   T. Ogita, S. M. Rump, and S. Oishi. "Accurate Sum and Dot Product". In: *SIAM J. Sci. Comput.* 26.6 (June 2005), pp. 1955–1988. ISSN: 1064-8275.

[71]   F. W. J. Olver and P. R. Turner. "Implementation of level-index arithmetic using partial table look-up". In: *1987 IEEE 8th Symposium on Computer Arithmetic (ARITH)*. May 1987, pp. 144–147.

[72]   W. Orchard-Hays. *Advanced linear-programming computing techniques*. New York: McGraw-Hill, 1968.

[73]   M. Pichat. "Correction d'une somme en arithmetique a virgule flottante". In: *Numerische Mathematik* 19.5 (1972), pp. 400–406. ISSN: 0945-3245.

[74]   D. M. Priest. "Algorithms for arbitrary precision floating point arithmetic". In: *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic*. June 1991, pp. 132–143.

[75]   D. M. Priest. "On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations". PhD thesis. USA, 1992.

[76]   B. Randell. "From Analytical Engine to Electronic Digital Computer: The Contributions of Ludgate, Torres, and Bush". In: *Annals of the History of Computing* 4.4 (Oct. 1982), pp. 327–341. ISSN: 0164-1239.

[77]   S. M. Rump. "A class of arbitrarily ill-conditioned floating-point matrices". In: (1991).

[78]   S. Rump, T. Ogita, and S. Oishi. "Accurate floating-point summation part I: Faithful rounding". English. In: *SIAM Journal of Scientific Computing* 31.1 (2008), pp. 189–224. ISSN: 1064-8275.

[79]   F. Sanglard. *Game Engine Black Book: DOOM v1.1*. Software Wizards.

[80]   F. Sanglard. *Game Engine Black Book: Wolfenstein 3D v2.1*. Software Wizards.

[81]   E. M. Schwarz, M. Schmookler, and S. D. Trong. "FPU implementations with denormalized numbers". In: *IEEE Transactions on Computers* 54.7 (July 2005), pp. 825–836. ISSN: 2326-3814.

[82]   C. Severance. "IEEE 754: An Interview with William Kahan". In: *Computer* 31.3 (Mar. 1998), pp. 114–115. ISSN: 1558-0814.

[83]   M. I. Smidla József. "Numerikusan szélsőségesen instabil feladatok megoldása a szimplex módszerrel". In: XXXI. Magyar Operációkutatási Konferencia (Cegléd, Magyarország). 2015.

[84]   M. I. Smidla József. "Sorfolytonos szorzat alakú bázis inverz reprezentáció a primál szimplex módszerben". In: XXX. Magyar Operációkutatási Konferencia (Balatonőszöd, Magyarország). 2013.

[85]   J. Smidla and I. Maros. "Stable vector operation implementations, using Intels SIMD architecture". In: *Acta Polytechnica Hungarica* 15.1 (2018).

[86] J. Smidla and G. Simon. "Accelerometer-based event detector for low-power applications". In: *Sensors* 13.10 (2013), pp. 13978–13997.

[87] J. Smidla and G. Simon. "Efficient accelerometer-based event detector in wireless sensor networks". In: *2013 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*. IEEE. 2013, pp. 732–736.

[88] J. Smidla, P. Tar, and I. Maros. "A numerically adaptive implementation of the simplex method". In: VOCAL (Veszprém, Hungary). 2014.

[89] J. Smidla, P. Tar, and I. Maros. "Adaptive Stable Additive Methods for Linear Algebraic Calculations". In: 20th Conference of the International Federation of Operational Research Societies (Barcelona, Spain). 2014.

[90] C. SPARC International Inc. *The SPARC Architecture Manual (Version 9)*. USA: Prentice-Hall, Inc., 1994. ISBN: 0130992275.

[91] C. SPARC International Inc. *The SPARC Architecture Manual: Version 8*. USA: Prentice-Hall, Inc., 1992. ISBN: 0138250014.

[92] P. H. Sterbenz. *Floating-point computation*. Prentice-Hall series in automatic computation. Englewood Cliffs, NJ: Prentice-Hall, 1974.

[93] I. Systems and T. Group. *Power ISA^{TM} Version 3.0 B*. Mar. 2017.

[94] S. R. T. Nishi and S. Oishi. "On the generation of very ill-conditioned integer matrices". In: *Nonlinear Theory and Its Applications* 2 (2 2011), pp. 226–245.

[95] A. Takahashi, M. I. Soliman, and S. Sedukhin. "Parallel LU-decomposition on Pentium Streaming SIMD Extensions." In: *ISHPC*. Ed. by A. V. Veidenbaum, K. Joe, H. Amano, and H. Aiso. Vol. 2858. Lecture Notes in Computer Science. Springer, Oct. 2003, pp. 423–430. ISBN: 3-540-20359-1.

[96] T. Terlaky. "A convergent criss-cross method". In: *Optimization* 16.5 (1985), pp. 683–690.

[97] T. Terlaky. "A finite crisscross method for oriented matroids". In: *Journal of Combinatorial Theory, Series B* 42.3 (1987), pp. 319–327. ISSN: 0095-8956.

[98] M. E. Thomadakis and J. Liu. *An Efficient Steepest-Edge Simplex Algorithm for SIMD Computers*. Tech. rep. College Station, TX, USA, 1997.

[99] J. Tomlin. "On pricing and backward transformation in linear programming". In: *Mathematical Programming* 6 (1974), pp. 42–47.

[100] J. Tomlin and J. Welch. "A pathological case in the reduction of linear programmings". In: *Operations Research Letters* 2 (1983), pp. 53–57.

[101] J. Tomlin and J. Welch. "Formal optimization of some reduced linear programming problems". In: *Mathematical Programming* 27.2 (1983), pp. 232–240.

[102] A. Vazquez and E. Antelo. "A Sum Error Detection Scheme for Decimal Arithmetic". In: *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*. July 2017, pp. 172–179.

[103] Á. Vázquez, E. Antelo, and J. D. Bruguera. "Fast Radix-10 Multiplication Using Redundant BCD Codes". In: *IEEE Transactions on Computers* 63.8 (Aug. 2014), pp. 1902–1914. ISSN: 2326-3814.

[104] A. Vazquez, E. Antelo, and P. Montuschi. "Improved Design of High-Performance Parallel Decimal Multipliers". In: *IEEE Transactions on Computers* 59.5 (May 2010), pp. 679–693. ISSN: 2326-3814.

[105] J. Vuillemin. "Exact Real Computer Arithmetic with Continued Fractions". In: *IEEE Trans. Comput.* 39.8 (Aug. 1990), pp. 1087–1105. ISSN: 0018-9340.

[106] J. E. Vuillemin. "On Circuits and Numbers". In: *IEEE Trans. Comput.* 43.8 (Aug. 1994), pp. 868–879. ISSN: 0018-9340.

[107] H. Wang, P. Wu, I. G. Tanase, M. J. Serrano, and J. E. Moreira. "Simple, Portable and Fast SIMD Intrinsic Programming: Generic Simd Library". In: *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. WPMVP '14. Orlando, Florida, USA: Association for Computing Machinery, 2014, pp. 9–16. ISBN: 9781450326537.

[108] L. Wang, M. J. Schulte, J. D. Thompson, and N. Jairam. "Hardware Designs for Decimal Floating-Point Addition and Related Operations". In: *IEEE Transactions on Computers* 58.3 (Mar. 2009), pp. 322–335. ISSN: 2326-3814.

[109] Waterloo Maple Inc. *Maple*. Waterloo, Ontario, Canada.

[110] Wolfram Research Inc. *Mathematica*. Champaign, IL, USA.

[111] S. Wolfram. *The Mathematica Book (4th Edition)*. New York, NY, USA: Cambridge University Press, 1999. ISBN: 0-521-64314-7.

Index