JOHANNES GUTENBERG
UNIVERSITÄT MAINZ

Efficient Certification of Feasibility and Objective Value of Linear Programs and its Applications

Dissertation
zur Erlangung des Grades
„Doktor der Naturwissenschaften“

am Fachbereich Physik, Mathematik und Informatik
der Johannes Gutenberg-Universität
in Mainz

vorgelegt von

Daniel Dumitriu

geboren in Iaşi, Rumänien

Mainz, 2012

D77 – Mainzer Dissertation

Datum des Kolloquiums: 6. September 2012

To my wife

Abstract

The use of linear programming in various areas has increased with the significant improvement of specialized solvers. Linear programs are used as such to model practical problems, or as subroutines in algorithms such as formal proofs or branch-and-cut frameworks. In many situations a certified answer is needed, for example the guarantee that the linear program is feasible or infeasible, or a provably safe bound on its objective value. Most of the available solvers work with floating-point arithmetic and are thus subject to its shortcomings such as rounding errors or underflow, therefore they can deliver incorrect answers. While adequate for some applications, this is unacceptable for critical applications like flight controlling or nuclear plant management due to the potential catastrophic consequences.

We propose a method that gives a certified answer whether a linear program is feasible or infeasible, or returns 'unknown'. The advantage of our method is that it is reasonably fast and rarely answers 'unknown'. It works by computing a safe solution that is in some way the best possible in the relative interior of the feasible set. To certify the relative interior, we employ exact arithmetic, whose use is nevertheless limited in general to critical places, allowing us to remain computationally efficient. Moreover, when certain conditions are fulfilled, our method is able to deliver a provable bound on the objective value of the linear program. We test our algorithm on typical benchmark sets and obtain higher rates of success compared to previous approaches for this problem, while keeping the running times acceptably small. The computed objective value bounds are in most of the cases very close to the known exact objective values.

We prove the usability of the method we developed by additionally employing a variant of it in a different scenario, namely to improve the results of a Satisfiability Modulo Theories solver. Our method is used as a black box in the nodes of a branch-and-bound tree to implement conflict learning based on the certificate of infeasibility for linear programs consisting of subsets of linear constraints. The generated conflict clauses are in general small and give good prospects for reducing the search space. Compared to other methods we obtain significant improvements in the running time, especially on the large instances.

Zusammenfassung

Der Einsatz von linearer Programmierung in verschiedenen Bereichen ist mit dem signifikanten Fortschritt spezialisierter Löser gewachsen. Lineare Programme finden Anwendung in der Modellierung von praktischen Problemen und als Unterprogramme in Algorithmen, wie zum Beispiel bei formalen Beweisen oder bei branch-and-cut Ansätzen. In vielen Situationen wird eine zertifizierte Antwort benötigt, in Form einer Garantie, dass das lineare Programm lösbar oder unlösbar ist, oder in Form einer beweisbar sicheren Schranke an den Zielfunktionswert. Die meisten verfügbaren Löser arbeiten mit Gleitkommaarithmetik und unterliegen somit den bekannten Defiziten wie Rundungsfehlern oder Unterläufen, welche dazu führen können, dass sie fehlerhafte Antworten liefern. Während dies für die meisten Anwendungen ausreichend ist, ist die Unsicherheit aufgrund der potenziell katastrophalen Folgen für kritische Anwendungen, wie zum Beispiel im Bereich der Flugsicherung oder Kraftwerkssteuerung, untragbar.

Wir präsentieren eine Methode, die eine zertifizierte Antwort auf die Frage ob ein lineares Programm lösbar oder unlösbar ist, liefert und ansonsten mit 'unbekannt' antwortet. Der Vorteil unserer Methode ist, dass sie eine vernünftige Laufzeit besitzt und nur selten 'unbekannt' zurückliefert. Sie berechnet eine sichere Lösung, welche die beste im relativen Inneren des Lösungsraums ist. Um das relative Innere sicher zu berechnen, verwenden wir exakte Arithmetik, deren Einsatz nur in kritischen Berechnungsschritten akzeptiert wird, damit der Algorithmus laufzeittechnisch effizient bleibt. Des Weiteren ist unsere Methode bei Erfüllbarkeit besonderer Bedingungen in der Lage eine beweisbare Schranke an den Zielfunktionswert des linearen Programms zu liefern. Wir testen unseren Algorithmus an typischen Benchmark-Instanzen und erzielen höhere Erfolgsraten im Vergleich zu vorherigen Ansätzen für dieses Problem, während die Laufzeit akzeptabel klein bleibt. Die berechneten Schranken liegen in den meisten Fällen sehr nah an den bekannten Zielfunktionswerten.

Die Einsatzfähigkeit unserer Methode zeigen wir zusätzlich durch die Entwicklung einer Variante, die die Resultate eines Satisfiability Modulo Theories Lösers verbessert. Unsere Methode wird als Black Box in den Knoten eines branch-and-bound Baums genutzt, um ein konfliktgesteuertes Lernverfahren zu unterstützen, das auf den Unlösbarkeits-Zertifikaten linearer Programme beruht, die aus Teilmengen linearer Nebenbedingungen bestehen. Die generierten Konflikt-

klauseln sind im Allgemeinen klein und liefern somit gute Beschränkungen des Suchraums. Verglichen mit anderen Methoden erhalten wir durch unseren Ansatz, insbesondere bei großen Instanzen, eine signifikante Reduzierung der Laufzeit.

Acknowledgments

First of all, I thank my advisor for introducing me to the fascinating area of optimization and offering me the chance to work with him, and for his guidance and help throughout my doctoral studies. His continuous support, opinions and advices were most important in carrying out this dissertation.

Many thanks to the members of the Institute for Computer Science at the Johannes Gutenberg University in Mainz. It has been a great pleasure and honor to be a part of this group. I am also indebted to the Algorithms and Complexity group at the Max Planck Institute for Computer Science in Saarbrücken, where I started my Ph.D. studies.

I want to thank my second reviewer for his instant commitment to evaluate this dissertation.

I am very thankful to all my friends and colleagues who contributed to this dissertation in many ways, through advices, support and fruitful discussions.

My deepest gratitude goes to my wife, for her patience and for always being near, and to my parents and my brother for all they have done for me.

Finally, I would like to recognize my sources of financial support: Max Planck Society (MPG) through its International Max Planck Research School for Computer Science, and the German Research Foundation (DFG) through its priority program "SPP 1307: Algorithm Engineering", under grants AL 1139/1–1 and AL 1139/1–2.

Daniel Dumitriu Mainz, 31 May 2012

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

If one would take statistics about which mathematical problem is using up most of the computer time in the world, then (not including database handling problems like sorting and searching) the answer would probably be linear programming.

— László Lovász [58] (1980)

Since none of the numbers which we take out from logarithmic and trigonometric tables admit of absolute precision, but are all to a certain extent approximate only, the results of all calculations performed by the aid of these numbers can only be approximately true ... It may happen, that in special cases the effect of the errors of the tables is so augmented that we may be obliged to reject a method, otherwise the best, and substitute another in its place.

— Carl Friedrich Gauss [36] (1809)

Linear programming has been increasingly used in various areas during the last seventy years. It started initially as a method for creating production plans ('programs') for the military in the context of World War II, and extended soon to many other branches of industry. To the present day it remains a main tool in many domains, especially in operations research. On the other hand, linear programming was the starting point in developing the more general optimization theory such as duality or the meaning of convexity, and helped to gain insights in the much related areas of integer and mixed integer programming.

Apart from their direct employment in practice, and encouraged by the continuous improvement of the specialized solvers, linear programs are used in many theoretical settings. For example, they help to investigate new ideas and either assess their validity, or discard them much faster than in a classical setting; they

provide an important aid in deciding conjectures, and even in proving formal theorems. Moreover, linear or (mixed) integer programs are used as 'subroutines' in many algorithms, whose correctness therefore depends on the reliability of the answers that can be delivered for those programs.

Often the interest does not consist in finding an exact, or even almost exact, solution for the program at hand, but in a guaranteed answer of qualitative type, e.g., whether the program is feasible or infeasible, or in reporting a safe bound on its objective value. Solvers for linear programs work usually with floating-point representations of the input, and are therefore subject to rounding errors that can have hazardous results, e.g., declare feasible an infeasible program. Most of the available solvers, including commercial ones, do not come with a guarantee, and admit that numerical difficulties can degrade dramatically the quality of solutions, up to the point of providing 'solutions' to otherwise infeasible problems.

While this kind of fallacies can completely invalidate for example a mathematical proof and thus require additional efforts, in the case of critical applications, e.g., flight controlling or transportation of dangerous goods, such errors are simply unacceptable due to the potential catastrophic consequences. For these reasons, a lot of effort has been put in the last years into finding methods that can provide exact solutions for linear programs, on the one side, and give provably reliable, correct qualitative answers, on the other.

1.1 Motivation

The number of applications where linear programming is used is continually increasing. There are several explanations for this development.

Many real-life problems can be modeled as linear programs, therefore efficient methods to solve them can have a significant impact in many areas. Especially in industry and the financial sector the importance of this fact has been long established, and the potential of operations research, in general, and linear programming in particular, to help increase productivity, improve resource consumption, and eventually minimize cost and maximize profit has been long recognized. For this reason, significant investments have been made into intense research in this area, which led to continuous improvements, especially concerning the quality of solvers.

Surprisingly enough, the interest for exact, certified solutions did not appear until recently, if we think about the long history of linear programming. One of the first papers to consider an exact method appeared in 1998 and is due to Gärtner [35], who analyzes several problems in computational geometry and underlines the importance of correctly computed results. To quote Gärtner:

> If the objective is to test whether a certain point lies inside a given 0-1-polytope (a problem which can be formulated as an LP), a wrong answer to this decision problem might have disastrous effects in an ambient algorithm, or lead to theoretical 'insights' which are none.

Dhiflaoui et al. [27] followed suite to recognize the annihilating effect that false answers can have on algorithms using linear programs internally, and develop a method that verifies the optimal bases returned by regular LP solvers, and repairs them in case they prove to be nonoptimal.

Most of the linear program solvers, including the majority of commercial ones, use floating-point arithmetic. There are several clear reasons why this is the case: this type of arithmetic is fast, provides results accurate *enough*, and it is specified in the international standard ISO 60559, causing it to be used nowadays by virtually all microprocessors.

Nevertheless, floating-point arithmetic is far from being perfect. Its limited representation size, inherent to any system of this type, means that most of the numbers are only represented with approximation. For example, it is well-known that the decimal number 0.1 cannot be represented exactly in base 2, i.e., $0.1_{10} = 0.0001100110011\ldots_{2}$. Therefore the operations can suffer from significant rounding errors; moreover, these errors, while usually small, can accumulate to produce large final deviations.

Another problem associated with floating-point arithmetic worth mentioning is that it is customary to deem two numbers equal, or one of them smaller than the other, by comparing their difference to a predefined threshold. One of the difficulties with this kind of reasoning is that the induced relation does not constitute an equivalence relation, i.e., if $x \sim y$ and $y \sim z$, then it does not necessarily follow that $x \sim z$. For an extended discussion on the properties and risks of floating-point arithmetic we point to the survey of Goldberg [39].

In this context, our affirmation that floating-point arithmetic is accurate *enough* depends on the needs of the application that uses it. While in many situations a limited error is acceptable, there are numerous examples where absolute precision is needed. We only mention here critical systems like flight control and surveillance of nuclear plants, but also seemingly less vital applications like computer optimization or chip design verification.

There are several cases when floating-point errors have had dramatic consequences. One of the most well-known example is the crash of Ariane 5 Flight 501 in 1996. The European space rocket Ariane 5 crashed 27 seconds after launch due to a software error that produced a hardware exception [70]. More specifically, a data conversion from a 64-bit floating-point to a 16-bit signed integer number, for which the detection mechanism had been turned off for efficiency reasons, had triggered an arithmetic overflow that caused the rocket to receive incorrect guiding instructions and consequently crash. The incident resulted in a loss of around 370 million US dollars [28].

Apart from the applications where one needs a specific quantitative accuracy, i.e., for the solution and objective value of a linear program, there are increasingly more situations where *one* single solution is needed, or more exactly, the certainty that such a solution exists or not, in other words, if the linear program at hand is feasible or infeasible. Many decision systems make use of such an answer to determine a specific action or select a particular resolution path. Often, a problem

is not handled by solving a single linear program, but a search tree is computed, where in each of its nodes a linear program is solved and its (in)feasibility, or bounds computed on its objective value are used to guide or stop the search; this is the case, for example, with the branch-and-cut algorithms. If for some optimization problems floating-point errors can create a solution slightly different from the optimum, in case of feasibility problems they might simply return the wrong answer.

Linear and integer program solvers can err both in the degree of accuracy of solutions, as well as in giving a correct answer to the seemingly easier problem of declaring an instance feasible or infeasible. For the former aspect, Steffy [79] gives an example involving a well-known benchmark instance for which only 6 out of 22 different configuration settings tested for various LP solvers gave an almost exact result; moreover, the values returned differed among themselves by as much as 3%. For the latter case, Neumaier and Shcherbina [69] showed that many commercial integer program solvers wrongly declared infeasible an integer program consisting of only 20 constraints and 20 variables.

We pointed so far several serious problems that can affect floating-point computations and might have dire consequences. Nevertheless, linear program solvers that are based on floating-point arithmetic continue to be extensively used in practice, even in highly sensitive or high-profile areas. The reasons for this state of things are threefold.

First, in many cases the speed is important enough to compensate for the lack of accuracy [16]. Often, an approximate but quickly obtained solution suffices in the given context; moreover, the problem definition itself can be affected by inaccuracies in the input data (see Section 1.1.2).

Second, state-of-the-art software, even based on floating-point arithmetic, does produce useful, *nearly* optimal results; if the definition of *nearly* for some specific application and instance satisfies the user requirements, then the results obtained can be successfully used. For example, Koch [56] computed exactly the objective values of the linear programs in the standard benchmark set known as the Netlib LP library [68] and discovered that both the commercial solver CPLEX [48] and the free SoPlex [78] found the optimal bases for most of the problems with their default settings, and for the rest also after the precision was increased.

Third, there are simply not enough fast, reliable tools that can replace with a satisfactory degree of success the established, inexact ones, and provide exact and certifiable answers. Given that alternatives to floating-point computations such as exact or interval arithmetic are orders of magnitude slower, and will arguably stay within this range in the near future, the focus in building reliable tools will need to be put on finding the mix of different computation types that gives the best tradeoff between quality of results and running time. In this respect, this dissertation can be regarded as an attempt to develop such a tool.

1.1.1 Applications

As mentioned earlier, linear and integer programs are emerging as an important tool for research in various scientific areas. In most of these, high accuracy and certifiability of results is of utmost importance.

For mathematics in particular, Bailey and Borwein [7] describe several uses of computer methods, e.g., to gain insight and intuition, visualize principles, discover new relationships, test conjectures, and explore promising candidate results. The most famous case of use of computers, including linear programs, to solve a mathematical problem is the Kepler conjecture. Hales' proof by exhaustion [43] involves extensive testing of different configurations, eventually requiring the solving of thousands of linear programs. After fourteen years, the proof is still not completely finished, and it is estimated that converting it to a fully formal one necessitates about twenty years of work. Obua [71] focused on the linear programs involved in the proof and used a floating-point solver and interval arithmetic to disprove many counterexamples to the conjecture.

Another use of linear programs is to compute theoretical lower and upper bounds for different problems. For example, Levi et al. [57] used linear programming to develop approximation algorithms for the capacitated facility location problem, and Resende et al. [76] proved lower bounds for the quadratic assignment problem.

Linear programs are also used in combinatorial auctions, a very lucrative area, where interested persons are allowed to bid for combinations of items, for details we point to the survey of de Vries and Vohra [24], and in compiler optimization. A variant of integer programming called constraint integer programming allows additional constraint types and is used, e.g., to solve chip design verification problems [1].

Although in some cases an appropriate level of accuracy is considered sufficient, most of the applications presented above need tools that can provide certain answers.

1.1.2 Inexact Computations

We mentioned so far briefly the disadvantages and even dangers that inexact computations can have on the outcome of any algorithm when it is implemented as a computer program, and explained shortly why the floating-point arithmetic, and algorithms using it, are nevertheless extensively used in practice. In this section we succinctly discuss the main causes for errors in numerical computations.

Sources of errors in computations

Apart from pure bugs introduced unwillingly in the process of programming, every computer program that makes use of numerical computations is subject to errors that can influence its outcome, sometimes dramatically. There are three major sources of errors [47]:

1. *Truncation errors*, also called *discretization errors*, appear when some terms in a computation are omitted; the simplest example is a Taylor series. They appear often in numerical algorithms, for example in Newton's method for nonlinear equations.

2. *Data uncertainty.* In general there are few situations where the data used is provably certain, especially in real-world instances. The most common source of uncertainty is given by errors in measurements, or their lack of accuracy. It is well known that, for example in industry or engineering, these are usually accurate only to a few digits – an important fact for our discussion, since most of the linear programs arising in practice stem from these areas.

   Other sources of uncertainty have to do with computers, or computer programs. For example, the simple fact of storing the data on the computer can induce errors (e.g., as already mentioned, the decimal number 0.1 is not exactly representable in binary). Finally, the input data can be obtained as the output data from another program, leading to error accumulation.

   The effects of errors introduced by uncertainty in the data can be analyzed using the perturbation theory.

3. *Rounding errors.* The most important source of computation errors is related to the way how numbers are represented and dealt with in computers. Every number system for computers has inherently shortcomings which are rooted in the finite precision arithmetic (see Section 2.2).

Rounding and truncation errors can be avoided by using symbolic manipulation packages like Maple [59], Mathematica [61], or MATLAB's [62] Symbolic Math Toolbox. The main disadvantage of this approach is its expensive running time, which can even become prohibitive sometimes, especially when such a procedure is supposed to be repeatedly used as a black box.

1.2 Contributions

The main contribution of this dissertation is the development of a method by which we can guarantee the answer we give to the types of questions about linear programs that we mentioned above. When such a guarantee cannot be given, the method returns 'unknown'; in our experiments this happened rarely. Beside certification, the method focuses on remaining computationally efficient. It is common to employ rational or interval arithmetic to avoid the problems raised by the errors that appear in floating-point computations. Unfortunately, this kind of arithmetic, while being exact, is orders of magnitude slower. This implies that in order to stay efficient, the algorithms should ideally restrict their use of rational arithmetic to the cases where it is absolutely needed. Our approach shifts the balance by using normal arithmetic for most of its part and switching to the

slow, exact variant only in a few places. That means that we refrain as much as possible from using exact but slow rational arithmetic. In the common setting where one uses linear programs as a procedure to be called many times, providing certified answers efficiently is of utmost importance. We evaluate our method in a typical experimental setting and discuss the results obtained, as well as possible improvements.

Part of this work has appeared as an article [2] in the proceedings of the *Eighth Symposium on Experimental Algorithms* and has been followed by an extended and improved article [3] that was published in the *Operations Research Letters* journal.

In the second part of this dissertation we additionally show the usability of our method by employing it in a typical black box setting. We use a variant of the method we developed to improve the overall solving performance of iSAT, a Satisfiability Modulo Theories solver that can solve Boolean combinations of linear and nonlinear constraints. The software iSAT is a tight integration of the well-known DPLL algorithm and interval constraint propagation allowing it to reason about linear and nonlinear constraints. Since interval arithmetic is known to be less efficient for solving linear programs, dealing with the subset of the linear constraints by using a solver capable of giving certified answers has good results. We perform experiments to test our approach and examine the results obtained.

The results of this work have been published as an article [4] in the proceedings of the *Fifth International Conference on Combinatorial Optimization and Applications*.

1.3 Outline

The remainder of this dissertation is structured as follows.

In Chapter 2 we present notions and results that are used in the course of the following discussion and thus help to better understand the subsequent chapters.

In Chapter 3 we describe our method that gives certified answers to the question whether a linear program is feasible or infeasible, or gives up when it cannot guarantee the answer, and computes safe bounds on the objective value. We develop the theory on which the method is based, and describe the implementation of a software application that we used in order to test it. We conclude with the experiments we performed and a discussion of the results.

In Chapter 4 we describe how our method can be used in practice to improve the results of an algorithm that is typical for the employment of linear programs and their solving methods, as a black box within the nodes of a branch-and-bound tree. We explain the general setting of the Satisfiability Modulo Theories and how our method has been employed, then we present the experiments made and conclude with the analysis of results.

Chapter 2

Preliminaries

This chapter is a presentation of notions and results that are important for the remainder of this dissertation. We start with some linear algebra facts, then we give an introduction on floating-point arithmetic, and conclude with the fundamentals of linear programming.

These results are not original contribution, but rather collected from various sources [12, 41, 47, 64, 67, 74]. They are only presented here for a better understanding of the subsequent chapters.

2.1 Some Linear Algebra

In the remainder of this dissertation "numbers" mean real numbers, unless otherwise stated.

2.1.1 Basic Facts

A *vector* $x \in \mathbb{R}^n$, $n \in \mathbb{N}$ is an ordered sequence of numbers called *components* or *entries*: $(x_1, x_2, \ldots, x_n)$, $x_i \in \mathbb{R}$, $i = 1, \ldots, n$.

We denote by $\mathbb{1}$ the vector with all components one. We use 0 interchangeably for the real number and for the vector or matrix with all components zero; the meaning should be clear from the context. The ith *unit vector* $\mathbf{e}_i$ has all components zero, with the exception of its ith component, which is 1.

A *hyperplane* is a set $\{x \in \mathbb{R}^n \mid a^T x = b\}$ with $a \neq 0$, $b \in \mathbb{R}$. A *half-space* is a set $\{x \in \mathbb{R}^n \mid a^T x \leqslant b\}$ with $a \neq 0$, $b \in \mathbb{R}$.

A *matrix* $A \in \mathbb{R}^{m \times n}$ is a rectangular array of real numbers with m rows and n columns; its rows and columns are vectors in $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively. If $m = n$ the matrix is called *square*. We usually write $A = (a_{ij})$, $i = 1, \ldots, m$, $j = 1, \ldots, n$, and denote by a_r the rth row of A, and by A_c the cth column of A.

Whenever we write "$\leqslant$" and "$\geqslant$" for a vector or a matrix, we will understand these relations component-wise, that is, holding for each entry.

A square matrix of size n consisting of 1s on the diagonal and 0s elsewhere is called *unit matrix* and is denoted by I_n; we simply write I when the size is clear.

The *transpose* of a matrix $A \in \mathbb{R}^{m\times n}$, denoted by $A^T \in \mathbb{R}^{n\times m}$, is the matrix obtained by using the rows of A as columns and vice versa.

A matrix $A \in \mathbb{R}^{n\times n}$ is *upper triangular* if $a_{ij} = 0$ for all $i > j$. The matrix is *lower triangular* if $a_{ij} = 0$ for all $i < j$.

We call a matrix $A \in \mathbb{R}^{m\times n}$ *sparse* if most of its entries are zero.

A square matrix A of size n is called *diagonally dominant* if $|a_{ii}| \geqslant \sum_{j\neq i}|a_{ij}|$ for $1 \leqslant i \leqslant n$ (*strictly* if the inequality is strict).

We call two square matrices A and B *equimodular* if $|a_{ij}| = |b_{ij}|$ for $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant n$.

A *permutation matrix* is a square matrix that has exactly one entry 1 in each row and column, and zeros elsewhere.

A *leading principal submatrix* of a square matrix A is a submatrix obtained by removing all but the first p rows and columns of A, where p is at most the size of A.

We can extend the definition of matrices to vectors by considering them as matrices with one column (respectively row). If not otherwise stated, "vector" will mean column vector, i.e., of size $m \times 1$.

Definition 2.1 (Linear dependency). The vectors $x_1, x_2, \ldots, x_n$ in $\mathbb{R}^m$ are *linearly dependent* if there exist real numbers $a_1, a_2, \ldots, a_n$, not all zero, such that

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = 0,$$

and *linearly independent* otherwise.

For $a_1, \ldots, a_n \in \mathbb{R}$, we call $a_1x_1 + a_2x_2 + \cdots + a_nx_n$ a *linear combination* of the vectors $x_1, x_2, \ldots, x_n$. If $a_1+a_2+\cdots+a_n = 1$, we call it a *convex combination*.

Definition 2.2 (Inverse and singular matrix). The *inverse* of a square matrix A, denoted by A^{-1}, is a matrix X such that $AX = I$. If A^{-1} exists, A is *nonsingular*, otherwise it is *singular*.

Definition 2.3 (Matrix rank). The *rank* of a matrix A in $\mathbb{R}^{m\times n}$, denoted by $\operatorname{rank}(A)$, is the maximum number of linearly independent row vectors of A.

We gave the definition of rank in terms of row vectors; for this reason this rank is also called *row rank*. One can similarly define the *column rank* of a matrix. The following theorem connects the two definitions:

Theorem 2.1. *The row rank and the column rank of a matrix are equal.*

Corollary 2.2. *The rank of a matrix A satisfies* $\operatorname{rank}(A) \leqslant \min\{m, n\}$ *and* $\operatorname{rank}(A) = \operatorname{rank}(A^T)$.

We say that a matrix has *full rank* if $\operatorname{rank}(A) = \min\{m, n\}$.

With this notion, we can alternatively define a square matrix as nonsingular if it has full rank.

2.1.2 Linear Systems

In this subsection we present a few known results on linear systems and algorithms for solving them.

Definition 2.4. A *linear system* is a set of linear equations that hold simultaneously:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases},$$

or in matrix notation

$$Ax = b,$$

where $A \in \mathbb{R}^{m \times n}$ is called the *coefficient matrix*, and $b \in \mathbb{R}^m$ is the *right-hand side vector* (or *free terms*).

The matrix $\bar{A} = [A \mid b]$ is called the *augmented matrix*.

The following theorem establishes a connection between the rank of the system matrices and the number of solutions of the system.

Theorem 2.3 (Rouché–Kronecker–Capelli). *Let $Ax = b$ be a system of linear equations with $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ and $\bar{A} = [A \mid b]$ be its augmented matrix. The system has solutions, i.e., is* consistent *if and only if* $\operatorname{rank}(A) = \operatorname{rank}(\bar{A})$. *Moreover:*

- *if* $\operatorname{rank}(A) = n$, *the solution is unique*
- *otherwise, there are infinitely many solutions.*

Corollary 2.4. *The maximum number of linearly independent solutions of the consistent linear system $Ax = b$ is* $n - \operatorname{rank}(A)$.

Methods for Solving Linear Systems

There are two categories of methods for solving linear systems, namely *direct* and *iterative* ones. The direct methods give the exact solution(s) after a finite, precise

sequence of operations, provided that they are executed in *exact* arithmetic; examples are Gaussian elimination, Gauss–Jordan elimination, and Cramer's rule. The best of this type of methods perform well for systems with up to a few thousand variables and equations. For much larger, especially sparse matrices, iterative methods are more efficient, e.g., Gauss–Seidel, successive overrelaxation or conjugate-gradient; they work by producing a sequence of successively better approximations of the solution.

We will focus on some direct methods that are relevant for our following discussion.

Definition 2.5 (Row echelon form[1]). A matrix is in *row echelon form* if it has the following properties:

1. all rows containing only zeros (if any) are at the bottom, and
2. for every two nonzero rows, the leading entry of the lower row is strictly to the right of the leading entry of the higher row,

where the *leading entry* is the leftmost nonzero entry of a row.

Every matrix can be transformed into row echelon form, but this is not unique, e.g., any multiple of a matrix in row echelon form is also in row echelon form.

Gaussian elimination (GE) is an algorithm for solving linear systems which works by bringing the augmented matrix into row echelon form through a series of *elementary row operations*:

1. Switching two rows
2. Multiplying a row by a nonzero constant
3. Adding a multiply of a row to another row.

Let $Ax = b$ be our system and $[A \mid b]$ having been transformed to the row echelon form $[A' \mid b']$. Then $A'x = b'$ and $Ax = b$ are equivalent systems, i.e., they have the same solution set. That means we can solve the transformed system instead, and because of its row echelon form, this can be done efficiently by back substitution.

We can obviously work separately on the system matrix and the right-hand side, with the advantage that we can solve more efficiently multiple systems that differ only in their free terms vector.

Plain GE uses only operations of type (2) and (3). It works column by column, from left to right and top to bottom, in a process called *pivoting*: an entry (the *pivot*) is chosen to become the leading entry in its row, and then used to make zero

[1] The name of this form is due to the characteristic "stair" shape formed by the leading entries (*échelon* is French for stair step or ladder rung).

all entries below the diagonal in its column. To this end the row is (optionally) divided by the pivot (this makes the pivot equal to 1) and convenient multiples of the row are added to the rows below. Basically, in each step k we work on the matrix that is obtained by removing the first $k-1$ rows and columns of the matrix from the previous step, and after the selection process the pivot is brought into position a_{kk}.

After step $k+1$ of the Gaussian elimination, we have

$$A^{(k+1)} = M_k A^{(k)} := \begin{pmatrix} I_{k-1} & & & & & \\ & 1 & & & & \\ & -m_{k+1,k} & 1 & & & \\ & -m_{k+2,k} & & \ddots & & \\ & \vdots & & & \ddots & \\ & -m_{n,k} & & & & 1 \end{pmatrix} \cdot A^{(k)},$$

where $A^{(1)} := A$ and $m_{i,k} = a_{ik}^{(k)} / a_{kk}^{(k)}$ is the multiplier for row i, $k+1 \leqslant i \leqslant n$.

We can write $M_k = I - m_k e_k^T$, where $e_i^T m_k = 0$ for $i \leqslant k$. Hence $M_k^{-1} = I + m_k e_k^T$, and M_k and M_k^{-1} are lower triangular, as well as their product. At the end we obtain

$$U := A^{(n)} = M_{n-1} A^{(n-1)} = \ldots = M_{n-1} M_{n-2} \ldots M_1 A,$$

which leads to

$$\begin{aligned} A &= M_1^{-1} M_2^{-1} \ldots M_{n-1}^{-1} U \\ &= \left(I + \sum_{i=1}^{n-1} m_i e_i^T \right) \cdot U = \begin{pmatrix} 1 & & & & \\ m_{21} & 1 & & & \\ \vdots & m_{32} & \ddots & & \\ \vdots & \vdots & & \ddots & \\ m_{n1} & m_{n2} & \ldots & m_{n,n-1} & 1 \end{pmatrix} \cdot U =: LU. \end{aligned}$$

In summary, Gaussian elimination computes a decomposition of A into a product of a lower and an upper triangular matrix, i.e., $A = LU$.

Theorem 2.5 (LU decomposition [85, Thm. 1.7.19]). *Let $A \in \mathbb{R}^{n \times n}$ be a matrix with all leading principal submatrices nonsingular. Then A can be uniquely decomposed as $A = LU$, where L is lower triangular and U is upper triangular.*

Plain Gaussian elimination takes approximately $\frac{2}{3}n^3$ operations (neglecting smaller terms) and is known as Gaussian elimination *without pivoting*. It only works if during the process we never encounter a zero pivot, which holds if all leading principal submatrices are nonsingular. In practice, due to rounding errors a candidate pivot will rarely be zero, but quite frequently will have a very small value, and dividing by this small and often incorrect value can lead to arbitrarily

inaccurate results. This means that the variant is not numerically stable, and in practice different pivoting strategies are used.

Gaussian elimination with *partial pivoting* works by selecting as pivot in step k the entry with the largest absolute value below the classical pivot. Let this be found in row r; rows r and k are switched before proceeding further. The cost for selecting the pivot is about $\frac{1}{2}n^2$ operations, significantly less when compared to the cost of the plain method. This variant calculates an LU decomposition in the form $PA = LU$, where P is a permutation matrix representing the row switches.

Gaussian elimination with *full pivoting* chooses as pivot in step k the largest entry in absolute value in the entire submatrix under consideration. Let this entry be found in row r and column c; *both* rows r and k and columns c and k are switched before going on. (This swaps the unknowns x_c and x_k, a fact that needs to be remembered.) In this case selecting the pivot is expensive, as it takes about $\frac{1}{3}n^3$ operations. This variant calculates an LU decomposition in the form $PAQ = LU$, where P and Q are permutation matrices representing the row and column switches, respectively.

Finally, there is one more pivoting strategy of interest to us. In the case when the system matrix is sparse, using the previous pivoting approaches can result in an arbitrarily large *fill-in*, i.e., a large number of entries that change from an initial zero to a nonzero value during the execution of the algorithm. This has a negative impact, primarily on the memory requirements, but also on the running time since the algorithm needs to touch significantly more entries.

Unfortunately, the problem of finding the sequence of pivots that minimizes the fill-in is NP-complete [87]. For this reason different heuristics are used in practice; for example, the so-called *Markowitz pivoting* [60], which selects at each step a pivot which minimizes the number of coefficients that will be modified at that step. This is done by computing for each entry of the submatrix under consideration the number of nonzero entries in its row, respectively column (excluding itself) and selecting as pivot the entry for which the product of the two counts is minimal. Notice that this means we can stop as soon as we have found an entry which is the only nonzero in its row or column. After selecting the pivot, we need to switch rows and columns correspondingly, as in complete pivoting. Since we look only at nonzero entries, computing the Markowitz numbers and choosing the pivot is usually not too expensive for sparse matrices and pays off by considerably reducing the fill-in.

This variant also computes a $PAQ = LU$ decomposition.

Definition 2.6 (Reduced row echelon form). A matrix is in *reduced row echelon form* if it has the following properties:

1. it is in row echelon form, and
2. every leading coefficient is 1 and it is the only nonzero in its column.

A matrix can be transformed into reduced row echelon form by an extension of Gaussian elimination called *Gauss–Jordan elimination* (GJE). The main dif-

ference to GE – besides dividing the pivot row by the pivot to get a leading 1 – is that in GJE we add convenient multiples of the pivot row not only to the rows below it, but also to those above it, in order to obtain zeros in the rest of the column.

Theorem 2.6. *Every matrix has a unique reduced row echelon form.*

Proposition 2.7. *The rank of a matrix is the number of nonzero rows in its reduced row echelon form.*

Gauss–Jordan elimination and Gaussian elimination have the same $O(n^3)$ complexity, but since the former takes about n^3 operations, it is 50% more expensive than the latter, and unless there are special considerations, Gaussian elimination is preferred for solving linear systems.

2.1.3 Matrix Norms

Usually the coefficients and terms in a linear system will be approximations, for example due to measurements or limitation of floating-point arithmetic (see Section 1.1.2). For this reason it is important to know what consequences small perturbations in the coefficients have on the solution of the system. To measure this effect we use matrix norms.

Definition 2.7. A *matrix norm* is a function $\|\cdot\|\colon \mathbb{R}^{n\times n} \to \mathbb{R}$ such that for all $A, B \in \mathbb{R}^{n\times n}$ and $\alpha \in \mathbb{R}$,

$$\begin{aligned}
\|A\| \geqslant 0 \quad (\text{zero iff } A = 0) &\qquad \text{(positive-definite property)}\\
\|\alpha A\| = |\alpha| \cdot \|A\| &\qquad \text{(absolute homogeneity)}\\
\|A + B\| \leqslant \|A\| + \|B\| &\qquad \text{(triangle inequality)}\\
\|AB\| \leqslant \|A\| \cdot \|B\| &\qquad \text{(submultiplicativity).}
\end{aligned}$$

Given a p-vector norm for $x \in \mathbb{R}^n$:

$$\|x\|_p = \begin{cases} \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p} & \text{for } 1 \leqslant p < \infty \\ \max_{1\leqslant i\leqslant n} |x_i| & \text{for } p = \infty \end{cases},$$

its subordinate matrix norm for $A \in \mathbb{R}^{n\times n}$ is

$$\|A\|_p = \max_{x\neq 0} \frac{\|Ax\|_p}{\|x\|_p}.$$

Some of the most commonly used matrix norms are:

$$\|A\|_1 = \max_{1\leqslant j\leqslant n} \sum_{i=1}^{n} |a_{ij}| \qquad \text{(maximum column sum)}$$

$$\|A\|_2 = \sqrt{\lambda_{\max}(A^T A)} \qquad \text{(spectral norm}^2\text{)}$$

$$\|A\|_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} |a_{ij}|^2} \qquad \text{(Frobenius norm)}$$

$$\|A\|_\infty = \max_{1\leqslant i\leqslant n} \sum_{j=1}^{n} |a_{ij}| \qquad \text{(maximum row sum)}$$

Theorem 2.8. *For any two vector norms $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$, we have*

$$r\|A\|_\alpha \leqslant \|A\|_\beta \leqslant s\|A\|_\alpha$$

for some positive numbers r and s, for all matrices $A \in \mathbb{K}^{m\times n}$.

In other words, all norms are *equivalent*; they induce the same topology on $\mathbb{K}^{m\times n}$.

For a matrix $A \in \mathbb{R}^{m\times n}$ and $r = \operatorname{rank}(A)$ we have:

$$\|A\|_2 \leqslant \sqrt{\|A\|_1\|A\|_\infty} \leqslant \sqrt{r}\|A\|_2$$
$$\|A\|_2 \leqslant \|A\|_F \leqslant \sqrt{n}\|A\|_2$$
$$\frac{1}{\sqrt{n}}\|A\|_\infty \leqslant \|A\|_2 \leqslant \sqrt{m}\|A\|_\infty$$
$$\frac{1}{\sqrt{m}}\|A\|_1 \leqslant \|A\|_2 \leqslant \sqrt{n}\|A\|_1.$$

In the remainder of this dissertation, unless otherwise explicitly stated, $\|A\|$ denotes the matrix infinity norm of A, $\|A\|_\infty$. A simple and useful notion for the sensitivity of the linear system $Ax = b$ is the condition number.

Definition 2.8. The *condition number* of a square matrix A, denoted by $\operatorname{cond}(A)$, is the number

$$\operatorname{cond}(A) = \begin{cases} \|A\| \cdot \|A^{-1}\| & \text{if } A \text{ is nonsingular,} \\ +\infty & \text{otherwise.} \end{cases}$$

[2] $\lambda_{\max}(A)$ denotes the largest eigenvalue of the matrix A, i.e., the largest number $\lambda \in \mathbb{R}$ such that $Ax = \lambda x$, where x is a nonzero vector of corresponding size.

The next theorem gives an upper bound on the solution error for a system when the coefficients and free terms are perturbed.

Theorem 2.9 ([85, Thm. 2.3.9]). *Let $A \in \mathbb{R}^{n\times n}$, and $x, b \in \mathbb{R}^n$. If A is non-singular, $b \neq 0$, $Ax = b$, $\operatorname{cond}(A) \cdot \|E\|/\|A\| < 1$, and $(A+E)(x+h) = b+d$, then*

$$\frac{\|h\|}{\|x\|} \leqslant \frac{\operatorname{cond}(A)\left(\dfrac{\|E\|}{\|A\|} + \dfrac{\|d\|}{\|b\|}\right)}{1 - \operatorname{cond}(A)\dfrac{\|E\|}{\|A\|}}.$$

Proof. From $Ax = b$ and $\|Ax\| \leqslant \|A\|\,\|x\|$ (submultiplicativity), we have

$$\|b\| \leqslant \|A\|\,\|x\|. \tag{2.1}$$

On the other hand, $(A+E)(x+h) = b+d$ yields

$$Ax + Ah + E(x+h) = b + d.$$

By using $Ax = b$ we obtain

$$\begin{aligned} Ah + E(x+h) &= d \\ Ah &= d - E(x+h) \\ h &= A^{-1}(d - E(x+h)) \end{aligned}$$

and, on taking norms and then using the definition $\operatorname{cond}(A) = \|A\|\,\|A^{-1}\|$:

$$\begin{aligned} \|h\| &\leqslant \|A^{-1}\|(\|d\| + \|E\|\|x+h\|) \\ &\leqslant \operatorname{cond}(A)\left(\frac{\|d\|}{\|A\|} + \frac{\|E\|}{\|A\|}(\|x\| + \|h\|)\right) \\ &\leqslant \operatorname{cond}(A)\frac{\|d\|}{\|A\|} + \operatorname{cond}(A)\frac{\|E\|}{\|A\|}\|x\| + \operatorname{cond}(A)\frac{\|E\|}{\|A\|}\|h\|. \end{aligned}$$

Reorganizing on the left-hand side all the terms containing $\|h\|$ produces

$$\begin{aligned} \left(1 - \operatorname{cond}(A)\frac{\|E\|}{\|A\|}\right)\|h\| &\leqslant \operatorname{cond}(A)\frac{\|d\|}{\|A\|} + \operatorname{cond}(A)\frac{\|E\|}{\|A\|}\|x\| \\ &= \operatorname{cond}(A)\|x\|\left(\frac{\|d\|}{\|A\|\,\|x\|} + \frac{\|E\|}{\|A\|}\right) \\ &\leqslant \operatorname{cond}(A)\|x\|\left(\frac{\|d\|}{\|b\|} + \frac{\|E\|}{\|A\|}\right). \qquad \text{(by 2.1)} \end{aligned}$$

Since we assumed $1 - \operatorname{cond}(A) \cdot \|E\|/\|A\| > 0$, we can divide both sides by $\big(1 - \operatorname{cond}(A) \cdot \|E\|/\|A\|\big) \cdot \|x\|$ and obtain

$$\frac{\|h\|}{\|x\|} \leqslant \frac{\operatorname{cond}(A)\left(\dfrac{\|E\|}{\|A\|} + \dfrac{\|d\|}{\|b\|}\right)}{1 - \operatorname{cond}(A)\dfrac{\|E\|}{\|A\|}}.$$

□

Estimating the norm of the inverse of a matrix

The obvious method to compute the norm of the inverse of a matrix is to compute first the inverse. This is a computationally expensive process which matches the complexity of matrix inversion [20, Chapter 28.4], i.e., $\Omega(n^2 \log n)$, where n is the matrix size. Although there are algorithms for matrix multiplication with a subcubic complexity, such as $O(n^{2.807})$ for Strassen's method or $O(n^{2.376})$ for Coppersmith–Winograd method, in practice their hidden constants are significantly large and the implementation is complex. For this reason Gaussian elimination is often used, yielding a $O(n^3)$ complexity. Yet in some cases we are interested only in a good enough, easier to compute approximation for the norm, i.e., an upper or lower bound on its value. In this section we will show how one such bound can be obtained.

We give two definitions important for the following discussion. A square matrix $A = (a_{ij})$ is an *H-matrix* if the comparison matrix $M(A) = (\mu_{ij})$ defined as

$$\mu_{ij} = \begin{cases} |a_{ij}| & \text{for } i = j, \\ -|a_{ij}| & \text{for } i \neq j \end{cases}$$

is an *M-matrix*, that is, if $\mu_{ij} \leqslant 0$ for $i \neq j$, $M(A)$ is nonsingular, and $M(A)^{-1} \geqslant 0$.

First we give a lemma due to Varah [83] that gives an upper bound for a specific type of matrix.

Lemma 2.10. *If $A \in \mathbb{R}^{n\times n}$ is strictly diagonally dominant, then*

$$\|A^{-1}\|_\infty \leqslant \frac{1}{\min\limits_{1\leqslant i\leqslant n}\left(|a_{ii}| - \sum\limits_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}|\right)}.$$

We prove the following lemma due to Varga [84], which establishes an upper bound for the case of H-matrices.

Lemma 2.11. *If $A \in \mathbb{R}^{n\times n}$ is a nonsingular H-matrix, then*

$$\|A^{-1}\|_\infty \leqslant \|M(A)^{-1}\|_\infty.$$

Proof. Denote $A = (a_{ij})$, and $A^{-1} = (\alpha_{ij})$. A is an H-matrix, hence $M(A)$ is an M-matrix, which is equivalent to stating that $U_A \neq \emptyset$, where

$$U_A = \{u \in \mathbb{R}^n \mid u > 0,\, M(A) \cdot u > 0,\, \|u\|_\infty = 1\}.$$

Let $f_A(u) = \min_{1 \leqslant i \leqslant n}(M(A) \cdot u)_i$. We have $f_A(u) > 0$ for any $u \in U_A$. It follows that there exists some $\hat{u} \in U_A$ such that $f_A(\hat{u}) = \max_{u \in \overline{U}_A} f_A(u) > 0$, where $\overline{U}_A$ is the closure of U_A.

From $M(A) \cdot u > 0$ it follows that

$$|a_{ii}| \cdot u_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \cdot u_j > 0, \quad \text{for } 1 \leqslant i \leqslant n.$$

Denoting $D = \operatorname{diag}(u_1, \ldots, u_n)$, this is equivalent to saying that AD is strictly diagonally dominant. Applying Lemma 2.10, we have

$$\|(AD)^{-1}\|_\infty \leqslant \frac{1}{\min\limits_{1 \leqslant i \leqslant n} \left(|a_{ii}| \cdot u_i - \sum\limits_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \cdot u_j \right)} = \frac{1}{\min\limits_{1 \leqslant i \leqslant n} (M(A) \cdot u)_i} = \frac{1}{f_A(u)}, \tag{2.2}$$

and furthermore

$$\begin{aligned} \|(AD)^{-1}\|_\infty = \|D^{-1}A^{-1}\|_\infty = \max_{1 \leqslant i \leqslant n} \sum_{j=1}^{n} \frac{|\alpha_{ij}|}{u_i} &\geqslant \frac{\max\limits_{1 \leqslant i \leqslant n} \sum_{j=1}^{n} |\alpha_{ij}|}{\max\limits_{1 \leqslant j \leqslant n} u_j} \\ = \frac{\|A^{-1}\|_\infty}{\max\limits_{1 \leqslant j \leqslant n} u_j} &= \frac{\|A^{-1}\|_\infty}{\|u\|_\infty} = \|A^{-1}\|_\infty. \end{aligned} \tag{2.3}$$

Using Equations (2.2) and (2.3) we obtain $\|A^{-1}\|_\infty \leqslant \dfrac{1}{f_A(u)}$. Since f_A is continuous on $\overline{U}_A$, we have

$$\|A^{-1}\|_\infty \leqslant \frac{1}{\max\limits_{u \in \overline{U}_A} f_A(u)}. \tag{2.4}$$

Obviously, Equation (2.4) holds for any matrix equimodular to A. Denoting by $\Omega(A)$ the set of matrices equimodular to A we can write

$$\sup_{B \in \Omega(A)} \|B^{-1}\|_\infty \leqslant \frac{1}{\max\limits_{u \in \overline{U}_A} f_A(u)}.$$

Let the vector $\hat{u}$ be defined as

$$\hat{u} = \frac{M(A)^{-1} \cdot \mathbb{1}}{\|M(A)^{-1} \cdot \mathbb{1}\|_\infty}.$$

We showed that $M(A)$ is a nonsingular M-matrix, hence $M(A)^{-1} \geqslant 0$, and $M(A)^{-1} \neq 0$, from where we get $\hat{u} > 0$. Moreover $M(A) \cdot \hat{u} = \mathbb{1}/\|M(A)^{-1}\|_\infty$, hence $\hat{u} \in U_A$. We have

$$f_A(\hat{u}) = \min_{1 \leqslant i \leqslant n} (M(A) \cdot \hat{u})_i = \min_{1 \leqslant i \leqslant n} \left(M(A) \cdot \frac{M(A)^{-1} \cdot \mathbb{1}}{\|M(A)^{-1} \cdot \mathbb{1}\|_\infty} \right)_i = \frac{1}{\|M(A)^{-1}\|_\infty}.$$

Clearly, $M(A)$ is equimodular to A, hence

$$\|M(A)^{-1}\|_\infty \leqslant \sup_{B \in \Omega(A)} \|B^{-1}\|_\infty \leqslant \frac{1}{\max_{u \in \overline{U}_A} f_A(u)} \leqslant \frac{1}{f_A(\hat{u})} = \|M(A)^{-1}\|_\infty,$$

from where it follows that

$$\|M(A)^{-1}\|_\infty = \frac{1}{\max_{u \in \overline{U}_A} f_A(u)}. \tag{2.5}$$

Using Equations (2.4) and (2.5) we obtain $\|A^{-1}\|_\infty \leqslant \|M(A)^{-1}\|_\infty$. □

We are now able to provide the main result (presented, for example, by Higham [46]) that gives an upper bound on the norm of the inverse of an upper triangular matrix.

Theorem 2.12. *Let T be a nonsingular upper triangular matrix. Then*

$$\|T^{-1}\|_p \leqslant \|M(T)^{-1}\|_p, \quad \text{for } p \in \{1, 2, \infty, F\}.$$

Proof. We will prove the theorem only for the case $p = \infty$.

Denote $T = (t_{ij})$, $T^{-1} = (\tau_{ij})$, $M(T) = (m_{ij})$, and $M(T)^{-1} = (\mu_{ij})$. T is nonsingular upper triangular, hence $t_{ii} \neq 0$ and $m_{ii} \neq 0$, $1 \leqslant i \leqslant n$. It follows that $\mu_{ii} = 1/m_{ii} \neq 0$, and $M(T)^{-1}$ exists, hence according to well-known characterizations of M-matrices (see, for example, the work of Berman and Plemmons [11, Chapter 6]), $M(T)$ is an M-matrix, and T is an H-matrix. We can apply Lemma 2.11 and complete the proof. □

Making straightforward changes it is easy to prove that Theorem 2.12 also holds for lower triangular matrices.

The value of $\|M(T)^{-1}\|_\infty$ can be computed using Algorithm 2.1 [46]. Thus an upper bound on $\|T^{-1}\|_\infty$ can be found using $O(n^2)$ operations.

$v_n = 1/|t_{nn}|$
for $i = n-1$ **to** 1 **do**
  $s = 1$
  for $j = i+1$ **to** n **do**
    $s = s + |t_{ij}| * v_j$
  end
  $v_i = s/t_{ii}$
end
$\|M(T)^{-1}\|_\infty = \|v\|_\infty$

Algorithm 2.1: Estimating $\|T^{-1}\|_\infty$ using $\|M(T)^{-1}\|_\infty$

2.2 Floating-point Arithmetic

Computers do not work with real numbers, but with approximations of them called *floating-point numbers*.

Definition 2.9. Let $x \in \mathbb{R}$ and $\tilde{x}$ an approximation of x. The *absolute error* of $\tilde{x}$ is

$$E_{\text{abs}}(\tilde{x}) = |x - \tilde{x}|,$$

and its *relative error* is

$$E_{\text{rel}}(\tilde{x}) = \frac{|x - \tilde{x}|}{|x|},$$

(undefined for $x = 0$), or equivalently

$$E_{\text{rel}}(\tilde{x}) = |\rho|,$$

where $\tilde{x} = x(1 + \rho)$.

In general, a floating-point number has the form

$$y = \pm m \cdot \beta^{e-t} \tag{2.6}$$

or alternatively

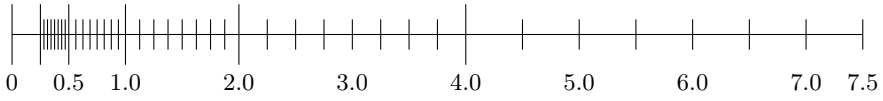
$$y = \pm\, .d_1 d_2 \ldots d_t \cdot \beta^e,$$

where β is the base, m is the significand, e is the exponent, and t is the precision (number of digits in the significand), all being integers, and d_i are digits in base β.

The significand satisfies $0 \leqslant m \leqslant \beta^t - 1$, and the exponent satisfies $e_{\min} \leqslant e \leqslant e_{\max}$. If $m \geqslant \beta^{t-1}$ is imposed in order to guarantee a unique representation for each number, the numbers are called *normal numbers*. Therefore there are $\beta^t \cdot (e_{\max} - e_{\min} + 1)$ representable nonzero normal floating-point numbers and their range is such that $|y| \in [\beta^{e_{\min}-1}, \beta^{e_{\max}}(1 - \beta^{-t})]$.

The *machine epsilon* $\varepsilon_M = \beta^{1-t}$ is the distance from 1.0 to the next larger floating-point number. The *unit roundoff* $u = \frac{1}{2}\varepsilon_M = \frac{1}{2}\beta^{1-t}$ is an important characteristic of a floating-point system.

An important feature of the floating-point system is that the numbers are *not* equally spaced. For example, consider the 41 representable floating-point numbers for the system with $\beta = 2$, $t = 4$, $e_{\min} = -1$, and $e_{\max} = 3$, which are arranged as follows:



Notice that at each power of β the spacing increases by a factor of β. The spacing for a floating-point number is given by the following lemma.

Lemma 2.13. *The spacing between two adjacent normal floating-point numbers x and y satisfies $\beta^{-1} \cdot \varepsilon_M \cdot |x| \leqslant |x - y| \leqslant \varepsilon_M \cdot |x|$.*

A floating-point system can be extended with *subnormal numbers* of the form $y = \pm m \cdot \beta^{e_{\min}-t}$, $0 < m < \beta^{t-1}$, i.e., numbers with the most significant digit d_1 equal to zero. The subnormal numbers fill the gap between 0 and the smallest normal number and are equally spaced, with spacing $\beta^{e_{\min}-t}$, but have fewer digits of precision. Therefore there are $\beta^{t-1} - 1$ subnormal numbers, the smallest being $\beta^{e_{\min}-t}$, while the smallest normal number is $\beta^{e_{\min}-1}$. For the case shown in the previous example, the subnormal numbers create for the interval $[0, 0.25]$ a structure similar to that depicted for the interval $[0.25, 0.5]$ for normal numbers.

For every real number x we associate a floating-point number $fl(x)$, defined as the floating-point number of the form (2.6) which is nearest to x; this mapping is called *rounding*. In case there are two floating-point numbers equally distanced from x, the tie can be broken in several ways; the most used is round-to-even, which selects the value having the least significant digit d_t even, as it has several desirable properties. We remark that rounding is monotonic, i.e., if $x \geqslant y$ then $fl(x) \geqslant fl(y)$.

There are two cases when rounding does not produce usable results. The first situation is called *overflow* and happens when the chosen value is too large to be represented in the floating-point system, i.e., if $|fl(x)| > \beta^{e_{\max}}(1 - \beta^{-t})$. The second is called *underflow* and happens when the value is too small to be represented, i.e., for the case when we also admit subnormal numbers, if $0 < |fl(x)| < \beta^{e_{\min}-t}$.

The following theorem shows that every representable real number can be approximated by a floating-point number with a relative error of at most u.

Theorem 2.14. *If $x \in \mathbb{R}$ is representable in a floating-point system, then*

$$fl(x) = x(1 + \delta), \qquad |\delta| < u.$$

Type	Size	t	$e_{\min}$	$e_{\max}$	Unit roundoff	Range
Single	32 bits	24	-126	$+127$	$2^{-24} \approx 5.96 \cdot 10^{-8}$	$10^{\pm 38}$
Double	64 bits	53	-1022	$+1023$	$2^{-53} \approx 1.11 \cdot 10^{-16}$	$10^{\pm 308}$

Table 2.1: Main floating-point types in IEEE 754

For algorithms and computer programs, it is customary to work with the following standard model. We consider the usual binary operations $+, -, \times, /$ on $\mathbb{R}$, respectively addition, subtraction, multiplication and division, and the corresponding operations on floating-point numbers denoted by $\oplus, \ominus, \otimes, \oslash$ respectively. Then

$$fl(x \circ y) = (x \circ y)(1 + \delta), \qquad |\delta| \leqslant u, \quad \circ \in \{+, -\times, /\}. \tag{2.7}$$

This means that the floating-point value corresponding to an elementary binary operation has the same relative error bound as the floating-point approximation of an usual real number. This model is valid for most computers, in particular for the most used computer standard for floating-point arithmetic, IEEE 754 (see Section 2.2.1).

2.2.1 IEEE 754

Most of the current microprocessors implement hardware floating-point based on the standard IEEE 754 [49, 50].

The main floating-point types defined by the standard and their most important properties are shown in Table 2.1. We mention that the single precision type is generally associated with C's `float` type, and the double precision type with C's `double` type. Since it can be inferred from its representation whether the number is normal or not, one extra bit is available for the significand.

Besides *normal* and *subnormal* numbers that follow the definitions given in the previous section, in the standard there are some other types of floating-point numbers:

- *Zeros.* The standard provides both a positive zero, $+0$, and a negative zero, -0. Zeroes are represented with an exponent and a significand of all 0s. Moreover, by definition in comparisons -0 is equal to $+0$.

  For computations involving reals, the existence of two signed zeros does not have considerable implications; the distinction becomes important for example when working with complex numbers.

- *Infinities.* There is a $+\infty$ and a $-\infty$, and they are usually generated by division by zero or overflow. Internally, an infinity has an exponent of all 0s and a significand of all 1s. Infinities obey the usual conventions, e.g., $\infty + \infty = \infty$, $x/\infty = 0$.

Exception	Example	Default result
Invalid operation	$0/0$, $0 \times \infty$, $\sqrt{-1}$	NaN
Overflow	$2^{1023} \times 2^{10}$	$\pm\infty$
Underflow	$2^{-1022} \times 2^{-10}$	Subnormal numbers
Divide by zero	$x/0$, when $x \neq 0$	$\pm\infty$
Inexact ("usual")	when $fl(x \circ y) \neq x \circ y$	Correctly rounded result

Table 2.2: Exceptions and default results in IEEE 754

- *NaN (Not-a-Number).* This is a special value representing the result of operations that cannot have a meaningful result in terms of a finite number or infinity. It has an exponent of all 1s and a nonzero significand.

  There are two categories of NaN: QNaN (quiet NaN), semantically denoting indeterminate operations and propagating freely, and SNaN (signalling NaN), denoting invalid operations and used to signal exceptions.

Programming languages have usually routines that can report on the type of a floating-point number. For example, in C the type can be queried with the functions `isnormal()`, `isnan()`, `isfinite()`, `isinf()`, or more generally with `fpclassify()`.

IEEE 754 also specifies five types of exceptions, presented in Table 2.2. It additionally requires subnormal numbers not to be flushed to zero, but represented.

The rounding is determined by the *rounding mode*. The standard defines four rounding modes:

- *round-to-*$+\infty$: $fl(x)$ is the least floating-point value greater than or equal to x
- *round-to-*$-\infty$: $fl(x)$ is the greatest floating-point value smaller than or equal to x
- *round-to-zero*: $fl(x)$ is the floating-point value of the same sign as x such that $|fl(x)|$ is the greatest floating-point value smaller than or equal to $|x|$
- *round-to-nearest*: $fl(x)$ is the floating-point value whose least significant bit d_t is zero.

The default rounding mode is round-to-nearest, as it has several useful properties, e.g., it is stable, i.e., $fl((((x+y)-y)+y)-y) = fl((x+y)-y)$, and tends to produce fewer ties when consecutive rounding operations are carried out on the same number.

In general, computer programs are allowed to change the rounding mode. C uses the routines `fegetround()` and `fesetround()` to respectively retrieve and set the rounding mode.

IEEE 754 arithmetic is monotonic, i.e., for floating-point numbers x, y, z, w, if $x \circ y \leqslant z \circ w$ then $x \circledcirc y \leqslant z \circledcirc w$. Monotonicity of arithmetic is important for some numerical algorithms.

The standard IEEE 754 does not comply exactly with Equation (2.7) since that model does not takes into account overflow and underflow. If we consider underflow, the model becomes

$$fl(x \circ y) = (x \circ y)(1 + \delta) + \eta, \quad |\delta| \leqslant u, \ |\eta| \leqslant \frac{1}{2}\beta^{e_{\min}-t}, \tag{2.8}$$

where η is nonzero and δ is zero if underflow occurs, otherwise δ is nonzero and η is zero. We can regard η as an absolute error for values very close to zero.

The standard specifies five operations: in addition to the four basic ones, it includes also the square root. It specifies *exact rounding*: the result of a floating-point operation is the same as if the operation had been carried out with real numbers and then rounded according to the chosen rounding mode, e.g., $x \oplus y = fl(x + y)$.

Floating-point arithmetic is not associative with respect to any of the four basic operations, i.e., $(x \circledcirc y) \circledcirc z \neq x \circledcirc (y \circledcirc z)$, where $\circledcirc \in \{\oplus, \ominus, \otimes, \oslash\}$. This can cause problems in symbolic computation (e.g., in compilers, proof analyzers) if associativity is assumed. For example, the GNU C compiler has an option `-funsafe-math-optimizations` whose enabling assumes the associativity of operations.

Finally, we mention one difficulty that may appear with floating-point numbers in computer programs. Processors like Intel 386, Pentium, Athlon contain a floating-point unit called "x87", which has registers of 80 bits (64 for the significand, 15 for the exponent), giving more precision than IEEE 754's double precision. By default, all operations in registers are carried out in this extended precision. Very often temporaries and variables are spilled from registers to memory, and this is done using the declared variable type, i.e., it can happen that an 80-bit value is spilled to a 64-bit cell. This means that the result of computations depend on register allocation, which can raise additional difficulties. One can ensure that the GNU C compiler always stores floating-point variables in memory by using its option `-ffloat-store`.

2.2.2 Rounding Errors

Since in computer programs all calculations involving real numbers are carried out with floating-point numbers, their results are only approximate, and rounding errors can affect them seriously.

Accuracy refers to the absolute or relative error of an approximate quantity. *Precision* is the accuracy with which the basic arithmetic operations $+, -, \times, /$ are performed, and for floating-point arithmetic is measured by the unit roundoff $u = \frac{1}{2}\beta^{1-t}$, i.e., 10^{-8} and 10^{-16} for IEEE 754's single and double precision, respectively. Accuracy is not limited by precision, and tends to be worse than the precision when multiple chained basic operations are involved.

Assume $y = f(x)$ and $\tilde{y}$ an approximation of y, and that there exists Δx such that $\tilde{y} = f(x + \Delta x)$. The value $\min |\Delta x|$ is called the *backward error*, and the value $|y - \tilde{y}|$ is called the *forward error* (both absolute, or relative if divided by $|x|$, respectively $|y|$).

A method is *backward stable* if it produces a small backward error for any x, i.e., $\tilde{y} = f(x + \Delta x)$ for some small Δx. A method is *forward stable* if it produces results having forward errors of the same size as the results produced by a backward stable method.

Backward error analysis looks for bounds on the backward error and views rounding errors as perturbations in the data, which are very common (see Section 1.1). In a way, if the backward error is smaller than the expected uncertainty in the data, no better result can be hoped for.

The connection between forward and backward error is given by the problem *conditioning*, i.e., the sensitivity of the solution with respect to the perturbations in the data. For f as above, its *condition number* is

$$c_f(x) = \left| \frac{x f'(x)}{f(x)} \right|.$$

A large condition number implies that small relative changes in the input can lead to large relative changes in the output. A problem with a large condition number is called *ill-conditioned*.

An useful rule of thumb is

$$\text{forward error} \lesssim \text{condition number} \times \text{backward error},$$

which means for example that for an ill-conditioned problem the forward error can be large.

One difficulty with floating-point computations is *cancellation*, which takes place when two almost equal numbers are subtracted. In this case very small numbers are produced, which can affect negatively subsequent operations (especially square roots and divisions); a well-known example is the formula for the roots of the quadratic equation. In some cases cancellation can be avoided by using a different, or a rewritten, equivalent mathematical formula.

Very often the instability of an algorithm is not due to the accumulation of a large number of rounding errors, but to only a few. For example, in computing Euler's number e using the formula $e = \lim_{n\to\infty}(1 + 1/n)^n$, the number $\frac{1}{n}$ cannot be exactly represented in binary when $n = 10^k$, and these errors account for the large error in the final result. Moreover, an algorithm can be stable for some problems, e.g., Gaussian elimination without partial pivoting (see Section 2.1.2) for upper Hessenberg matrices,[3] but unstable for others.

Usually, increasing the precision also increases the accuracy; this is the case for algorithms with an error bound proportional to the precision. However, there

[3]Matrix $A = (a_{ij})$ is upper Hessenberg if $a_{ij} = 0$ for $i > j + 1$.

is no rule that prescribes the increase in precision needed for a specific increase in accuracy. Furthermore, there are algorithms that do not benefit from increasing the precision.

In the course of the execution of a stable algorithm, rounding errors can cancel each other, and for some algorithms, rounding error can even be helpful.

Finally, we mention that rounding errors are not random, and it cannot be claimed that successive errors are independent; for example, it was observed that the digits in the representation of numbers follow the logarithmic distribution [9, 75]. Still, statistical analysis can provide insight about the effects of rounding. For instance, one rule of thumb is that an error estimate for an algorithm can be obtained by replacing the constants in a rounding error bound by their square root, i.e., for k roundoffs, by replacing k by $\sqrt{k}$ in an estimate of the maximum resultant error.

Numerical stability is an important property of algorithms, since otherwise they could produce incorrect results simply because of a very small change in the input data. For this reason, special care should be taken in designing (and implementing) such algorithms, for example, to avoid unnecessary underflows, overflows, and cancellations, or to transform a problem in such a way that it becomes well-conditioned, or to use equivalent, numerically stable formulas.

2.3 Linear Programming

2.3.1 Introduction

Linear programming was first developed by Kantorovich [51] and Dantzig [22] and deals with the problem of finding a point that minimizes (or maximizes) a linear function subject to a set of requirements. In its most general form, a *linear program* (LP) looks as follows:

$$\begin{array}{lrcl} \min & c^T x & & \\ \text{s.t.} & A'x & \leqslant & b' \\ & A''x & = & b'' \\ & A'''x & \geqslant & b''' \\ & x & \geqslant & \ell \\ & x & \leqslant & u, \end{array}$$

where $A' \in \mathbb{R}^{m' \times n}$, $b' \in \mathbb{R}^{m'}$, $A'' \in \mathbb{R}^{m'' \times n}$, $b'' \in \mathbb{R}^{m''}$, $A''' \in \mathbb{R}^{m''' \times n}$, $b''' \in \mathbb{R}^{m'''}$, and $x, c, \ell, u \in \mathbb{R}^n$.

The function to be minimized is called *objective function*; the next three rows are the *constraints*, and the last two are the *bounds*.

A maximization problem, e.g., $\max c^T x$ is equivalent to $\min -c^T x$. A constraint $a_i' x \leqslant b_i'$ is the same as $-a_i' x \geqslant -b_i'$, and an equality can be replaced by two symmetric inequalities. Finally, we can see the bounds as constraints. With

these considerations we can rewrite the previous linear program in *canonical form*:

$$\begin{array}{lrcl} \min & c^T x & & \\ \text{s.t.} & Ax & \geqslant & b, \end{array}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and $x, c \in \mathbb{R}^n$.

A more convenient form from a computational point of view is the *standard form*:

$$\begin{array}{lrcl} \min & c^T x & & \\ \text{s.t.} & Ax & = & b \\ & x & \geqslant & 0. \end{array}$$

Any problem in canonical form can be reduced to standard form. For inequality constraints, one introduces new *slack variables* s_i by replacing $a_i x \geqslant b_i$ by $a_i x - s_i = b_i$, $s_i \geqslant 0$. Variables with nonzero bounds are replaced by conveniently defined new variables, e.g., $x_i \geqslant \ell_i$ by $x_i' := x_i - \ell_i$, $x_i' \geqslant 0$. Finally, free (unbounded) variables are replaced by two related variables, $x_i := x_i^+ - x_i^-$, with bounds $x_i^+ \geqslant 0$ and $x_i^- \geqslant 0$.

We give now a number of definitions that are important for the following discussion.

A *polyhedron* is the intersection of half-spaces, i.e., a set $\{x \in \mathbb{R}^n \mid Ax \geqslant b\}$, where A is a matrix and b is a vector of agreeing dimensions.

A set S is *convex* if for any points $x, y \in S$ and any scalar $\lambda \in [0, 1]$ we have $\lambda x + (1 - \lambda)y \in S$. It follows that polyhedra are convex, and any convex combination of points in a convex set lies in the set.

A *polytope* is a bounded polyhedron, i.e., a polyhedron P for which there is a constant $U \in \mathbb{R}$ such that $P \subseteq [-U, U]^n$.

A point $x \in P$ is an *extreme point* of a polyhedron P if there exist no two points $y, z \in P$, $y, z \neq x$ and $\lambda \in [0, 1]$ such that $x = \lambda y + (1 - \lambda)z$.

A point $x \in P$ is a *vertex* if there exists some c such that for all $y \in P$, $y \neq x$ we have $c^T x < c^T y$.

A *solution* of a linear program is a point $x \in \mathbb{R}^n$ that satisfies all the constraints. It follows that the solution set of a linear program is a polytope.

Definition 2.10 (LP basis). Consider a linear program in standard form, and assume $m \leqslant n$ and $\operatorname{rank}(A) = m$. Let $B = \{B_1, \ldots, B_m\} \subseteq \{1, \ldots, n\}$ and $N = \{1, \ldots, n\} \setminus B$ be sets of indices, and $A_B \in \mathbb{R}^{m \times m}$ the submatrix consisting of the columns in A given by indices in B. If A_B is nonsingular, it is called a *basis*.

The solution $x_B = A_B^{-1} b_B, x_N = 0$ is called a *basic solution*. Moreover, it is called *basic feasible* if it satisfies the nonnegativity constraints, too.

Lemma 2.15. *Let $P = \{x \in \mathbb{R}^n \mid Ax = b, x \geqslant 0\}$, let $\bar{x}$ be one of its extreme points, and* $\operatorname{rank}(A) = m$. *There there exists a basis A_B such that $\bar{x} = A_B^{-1} b_B$.*

An inequality constraint is *active* at a point if it holds with equality at that point.

A basic solution x is called *degenerate* if more than n constraints are active at x.

Theorem 2.16. *Let P be a nonempty polyhedron and $x \in P$. The following are equivalent:*

- *x is a vertex*
- *x is an extreme point*
- *x is a basic feasible solution.*

Theorem 2.17 (Optimality of extreme points). *Consider the LP $\{\min c^T x, x \in P\}$, where $P = \{Ax \geqslant b\}$. Suppose P has at least an extreme point. Then, either the optimal cost is $-\infty$, or there exists an extreme point which is optimal.*

2.3.2 Duality

The *dual* of the linear program (called *primal*) in canonical form

$$\begin{array}{lrcl} \min & c^T x & & \\ \text{s.t.} & Ax & \geqslant & b \end{array}$$

is defined as the following linear program:

$$\begin{array}{lrcl} \max & b^T y & & \\ \text{s.t.} & A^T y & = & c \\ & y & \geqslant & 0. \end{array}$$

The definition of the dual can be motivated algebraically, geometrically and economically. Assume we want to prove a lower bound on z, the optimal value of the primal, by taking linear combinations of constraints $a_i x \geqslant b_i$ (e.g., by multiplying each of them by y_i) and looking to obtain a bound on $c^T x$. To ensure $y^T Ax \geqslant y^T b$ we need $y \geqslant 0$. As we want $c^T x \geqslant y^T Ax$, we need $A^T y = c$, since x is unbounded. This lower bound works for all feasible x, hence we have $z \geqslant b^T y$. The best possible of these bounds is exactly the solution of the dual LP.

Proposition 2.18. *The dual of the dual is the primal.*

This tight connection between the primal and the dual is exploited in numerous applications and algorithms, as well as in theoretical results. We give in the following some of the most important facts that show the relation between the two linear programs.

Weak and strong duality

First, there is a simpler result that says that the objective value for any dual solution lower bounds the objective value for any primal solution.

Theorem 2.19 (Weak duality). *If x is a feasible solution to the primal and y is a feasible solution to the dual, then*

$$b^T y \leqslant c^T x.$$

From weak duality follows an important fact that can be used to show the infeasibility of a linear program.

Corollary 2.20. *If one of the primal and dual is unbounded, the other is infeasible.*

In fact, the connection between the optimal solutions of the primal and dual is even stronger. The following theorem is one of the most important results in linear programming.

Theorem 2.21 (Strong duality). *If a linear program has an optimal solution, then its dual also has an optimal solution. Moreover, the respective optimal objective values are equal.*

Taking this into consideration, we can enumerate the four possible situations for the primal and dual.

Corollary 2.22. *Let P be a linear program and D be its dual. Then there are only the following possibilities:*

1. *The objective values of P and D are finite and equal.*
2. *P is unbounded and D is infeasible.*
3. *D is unbounded and P is infeasible.*
4. *Both P and D are infeasible.*

Complementary slackness

An important relation between the optimal solutions of the primal and dual says that if an inequality condition $a_i x \geqslant b_i$ is not tight for some optimal primal solution x, i.e., not satisfied at equality, then the corresponding dual variable y_i must be 0. This is the same as saying that in the optimal solutions at least one of the primal slack variable s_i and dual variable y_i is 0, or conversely, one of the dual slack variable w_j and primal variable x_j is 0.

Theorem 2.23 (Complementary slackness). *Let x be a feasible solution for the primal, and y a feasible solution for the dual; x and y are optimal solutions for the respective problems if and only if*

$$y_i(a_i^T x - b_i) = 0, \quad \text{for all } i = 1, \ldots, m,$$
$$x_j(c_j - y^T A_j) = 0, \quad \text{for all } j = 1, \ldots, n.$$

Notice that it can be the case that *both* the slack variable and the corresponding dual variable are zero. In order to have exactly one of the two equal to zero, additional conditions are needed.

If we are given a nondegenerate optimal basic feasible solution for the primal, we can use the complementary slackness conditions to compute an optimal solution for the dual.

2.3.3 Farkas' Lemma

An important application of linear programming duality is the possibility to prove that a linear program is infeasible by showing the feasibility of another related linear program. Moreover, in the process we obtain a *certificate* for the infeasibility of the former LP. The result is known as Farkas' lemma and predates linear programming, having been proved for systems of linear equations [30, 31].

Theorem 2.24 (Farkas' lemma). *Let A be a matrix in $\mathbb{R}^{m\times n}$ and b a vector in $\mathbb{R}^m$. Then, exactly one of the following is true:*

1. *There exists $x \in \mathbb{R}^n$, $x \geqslant 0$ such that $Ax = b$.*
2. *There exists $y \in \mathbb{R}^m$, $y \geqslant 0$ such that $A^T y \geqslant 0$ and $b^T y < 0$.*

Intuitively, if a linear system has no solution, we can find a vector y such that multiplying both the left-hand side and right-hand side by y produces a contradiction. The vector y is then the certificate of infeasibility (also called Farkas certificate) for the initial linear system. Notice that by convenient scaling we can always derive the contradiction in the form $0 = -1$.

Farkas' lemma is one variant of the so-called *theorems of the alternative*, i.e., containing two statements from which either one or the other is true, but not both. The best-known of them is due to Motzkin [65]:

Theorem 2.25 (Motzkin's transposition theorem). *Let A, B be matrices in $\mathbb{R}^{m\times n}$ and b, c be vectors in $\mathbb{R}^m$. The following are equivalent:*

1. *The system $Ax \leqslant b$, $Bx < c$ has a solution.*
2. *For all vectors $y, z \in \mathbb{R}^m$ with $y \geqslant 0$, $z \geqslant 0$:*
$$A^T y + B^T z = 0 \implies b^T y + c^T z \geqslant 0 \quad \text{and}$$
$$A^T y + B^T z = 0, z \neq 0 \implies b^T y + c^T z > 0.$$

Farkas' lemma can be stated for differently looking linear systems. It can be applied to linear programs by disregarding the objective function. In the following, we derive its form for LPs in general form.

Farkas' lemma for general form LPs

We saw that an LP $Ax \leqslant b$, $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ is infeasible if and only if the following LP is feasible: $A^T y = 0$, $y^T b < 0$, $y \geqslant 0$ ($y \in \mathbb{R}^m$).

A general form LP

$$\begin{aligned} A_1 x &\leqslant b_1 \\ A_2 x &= b_2 \\ A_3 x &\geqslant b_3 \\ x &\leqslant u \\ x &\geqslant \ell \end{aligned} \qquad \text{or equivalently} \qquad \begin{aligned} A_1 x &\leqslant b_1 \\ A_2 x &\leqslant b_2 \\ -A_2 x &\leqslant -b_2 \\ -A_3 x &\leqslant -b_3 \\ Ix &\leqslant u \\ -Ix &\leqslant -\ell \end{aligned}$$

can be written as

$$(A_1 \quad A_2 \quad -A_2 \quad -A_3 \quad I \quad -I)^T \cdot x = (b_1 \quad b_2 \quad -b_2 \quad -b_3 \quad u \quad -\ell)^T,$$

where

- $A_1 \in \mathbb{R}^{m_\leqslant \times n}$, $A_2 \in \mathbb{R}^{m_= \times n}$, $A_3 \in \mathbb{R}^{m_\geqslant \times n}$,
- $b_1 \in \mathbb{R}^{m_\leqslant}$, $b_2 \in \mathbb{R}^{m_=}$, $b_3 \in \mathbb{R}^{m_\geqslant}$, $u \in \mathbb{R}^n$, $\ell \in \mathbb{R}^n$,
- $y_1 \in \mathbb{R}^{m_\leqslant}$, $y_2 \in \mathbb{R}^{m_=}$, $y_3 \in \mathbb{R}^{m_=}$, $y_4 \in \mathbb{R}^{m_\geqslant}$, $y_5 \in \mathbb{R}^n$, $y_6 \in \mathbb{R}^n$,
- $m_\leqslant$ is the number of smaller-than inequalities, $m_=$ is the number of equalities, $m_\geqslant$ is the number of greater-than inequalities, hence $m_\leqslant + m_= + m_\geqslant = m$.

By Farkas' lemma, this LP is infeasible if and only if the following LP is feasible:

$$\begin{aligned} (A_1 \quad A_2 \quad -A_2 \quad -A_3 \quad I \quad -I) \cdot (y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5 \quad y_6)^T &= 0 \\ (y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5 \quad y_6) \cdot (b_1 \quad b_2 \quad -b_2 \quad -b_3 \quad u \quad -\ell)^T &< 0 \\ y_1, y_2, y_3, y_4, y_5, y_6 &\geqslant 0 \end{aligned}$$

or equivalently

$$\begin{aligned} A_1^T y_1 + A_2^T y_2 - A_2^T y_3 - A_3^T y_4 + y_5 - y_6 &= 0 \qquad (n \text{ constraints}) \\ b^T y_1 + b_2^T y_2 - b_2^T y_3 - b_3^T y_4 + u^T y_5 - \ell^T y_6 &< 0 \qquad (1 \text{ constraint}) \\ y_1, y_2, y_3, y_4, y_5, y_6 &\geqslant 0 \end{aligned}$$

This LP has $n + 1$ constraints and $2n + m + m_=$ variables.
If we replace y_2, y_3 by some free variables $y' := y_2 - y_3$, we obtain the LP

$$\begin{aligned} A_1^T y_1 + A_2^T y' - A_3^T y_4 + y_5 - y_6 &= 0 \\ b^T y_1 + b_2^T y' - b_3^T y_4 + u^T y_5 - \ell^T y_6 &< 0 \\ y_1, y_2, y_3, y_4, y_5, y_6 &\geqslant 0 \\ y' &\text{ free} \end{aligned}$$

which has $n + 1$ constraints and $2n + m$ variables.

In case of infinite bounds, the corresponding components of y_5 or y_6 are set to 0.

2.3.4 Methods for Solving Linear Problems

In this subsection we give a short description of the best known classes of algorithms for solving LPs.

We spend more time on the simplex algorithm for two reasons. First, we use it extensively in our approach. Second, in spite of a prohibitive theoretical complexity, well-implemented simplex-based methods are among the best performing LP solvers in practice. We present shortly the main alternatives. The ellipsoid method was the first one to have a polynomial-time complexity, although the practical implementations perform very poorly. The interior-point methods are also polynomial-time, and are solid competitors for simplex-based methods. In practice one or the other performs better, depending on the specific types of LP problems.

We mention that it is still an open problem whether linear programming admits a strongly polynomial-time algorithm, i.e., an algorithm whose complexity is polynomial also in the numerical values, or size, of the input data.

The simplex algorithm

The simplex algorithm was developed by Dantzig [22] and its basic idea is to move from a basic feasible solution to another yielding a better objective value, until an optimum has been found, or the problem is proved unfeasible or unbounded, as depicted in Figure 2.1. This is possible because we optimize a convex objective

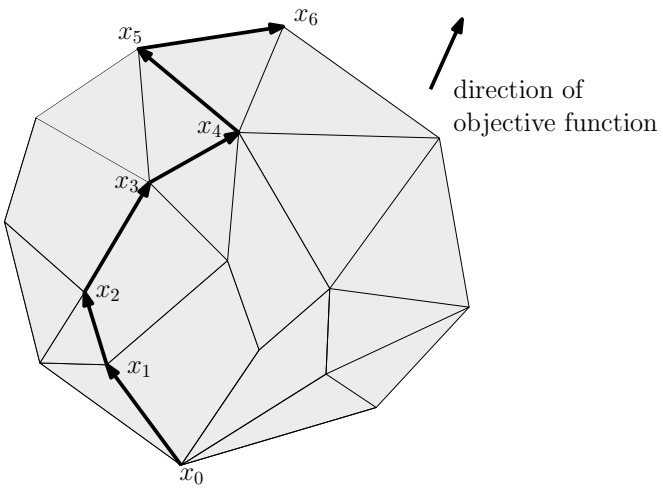


Figure 2.1: The simplex algorithm.

We start with a feasible extreme point x_0 and move along the polyhedral edges to a better basic feasible solution until we reach x_6, where no further improvement can be made.

function over a convex set. The feasible set is an n-dimensional simplex, hence the algorithm's name.

It follows from Theorem 2.17 that in order to find an optimal solution for an LP, it is sufficient to restrict oneself to vertices of the associated polytope, or equivalently to the basic feasible solutions.

The algorithm starts with a feasible extreme point, i.e., with a basis $B^{(0)}$ consisting of the columns $A_{B_1^{(0)}}, \ldots, A_{B_m^{(0)}}$ and an associated basic feasible solution $x^{(0)}$. In every iteration i, it attempts to find a new feasible basis. To this end, it looks for a variable that can enter the basis (the *entering* variable) by calculating the *reduced costs* $\bar{c}_j$ for all nonbasic indices $j \in N^{(i)}$:

$$\bar{c}_j = c_j - c_{B^{(i)}}^T {B^{(i)}}^{-1} A_j.$$

If all the reduced costs are nonnegative, then no variable can be chosen that improves the objective values, and the current solution is optimal.

Otherwise, it looks for a variable that can leave the basis (the *leaving* variable) by computing $u = {B^{(i)}}^{-1} A_j$. If $u \leqslant 0$, the optimal cost is $-\infty$ and the problem is unbounded. If there are positive entries in u, then it picks variable x_ℓ for which

$$\frac{x_\ell}{u_\ell} = \min_{k \in B^{(i)},\, u_k > 0} \frac{x_k}{u_k}.$$

In case both steps have been successful, the next iteration starts with the new basis $B^{(i+1)} = B^{(i)} \cup \{j\} \setminus \{\ell\}$. Since there are a finite number of bases and the objective value decreases in every iteration, we consider every basis at most

once, hence the algorithm terminates. We can give the following theorem:

Theorem 2.26. *For a linear program with a nonempty feasible region and whose every basic feasible solution is nondegenerate, the simplex algorithm:*

1. *terminates after a finite number of iterations, and*
2. *at termination, it either finds an optimal solution, or decides that the problem is unbounded.*

During the execution of the simplex algorithm there are a number of decisions that need to be made. In the following we discuss shortly the most important existing alternatives.

If there are several candidates for the entering variable, there exist several strategies. The simplest is the smallest subscript rule, also known as Bland's rule [14], which chooses the smallest j with negative reduced cost. This saves the effort to compute the remaining reduced costs, but it can increase the number of iterations; other criteria with good results are *Devex*, proposed by Harris [44], and *steepest edge*, proposed by Goldfarb and Reid [40]. A similar discussion applies to choosing the leaving variable.

The simplex algorithm can be subject to *cycling*: a sequence of basis changes that lead back to a previous basis, causing the algorithm to enter an endless loop. This can happen with degenerate basic feasible solutions, which appear especially in large linear programs, or by an unfortunate choice of the entering or leaving variable. Cycling is guaranteed to be avoided if one uses, e.g., Bland's rule. Since in practice these rules lead to more iterations, they are usually switched to only when it is suspected that the algorithm cycles.

Although subject to previous considerations the simplex algorithm terminates, and in practice it was observed to usually do so in $O(m \log n)$ iterations, it can take in the worst case an exponential number of iterations since there are $\binom{n}{m}$ possible bases. One example of such a worst-case instance is the Klee–Minty cube [55]:

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} 10^{n-j} x_j \\ \text{s.t.} \quad & \left(2 \cdot \sum_{j=1}^{n-1} 10^{i-j} x_j\right) + x_i \leqslant 100^{i-1}, \quad i = 1, \ldots, n \\ & x_1, \ldots, x_n \geqslant 0. \end{aligned}$$

This describes a linear program whose feasible region is a slightly perturbed n-dimensional hypercube for which the simplex algorithm needs $2^n - 1$ iterations under the basic pivoting rule. A variation of the cube needs to visit all its vertices even under Bland's rule [66].

The simplex algorithm needs a starting feasible solution. Assuming that $b \geqslant 0$ in the original LP (possibly by multiplying some constraints by -1), a feasible so-

lution can be found by introducing the variables $y \in \mathbb{R}^m$ and solving the *auxiliary problem*

$$\begin{array}{lrcl} \min & y \cdot \mathbb{1}^T & & \\ \text{s.t.} & Ax + y & = & b \\ & x & \geqslant & 0 \\ & y & \geqslant & 0, \end{array}$$

which can be done with the simplex method starting with the feasible solution $x = 0$, $y = b$. This is called Phase I simplex. It can be shown that the original problem has a feasible solution if and only if the objective value of the auxiliary problem is zero. In this case necessarily $y = 0$ and x is a feasible solution for the original problem that can be used (possibly after driving the y variables out of the basis) in the so-called Phase II simplex.

In practice one uses a slightly different implementation that aims at a better memory and time complexity. It refrains from computing B^{-1} every time from scratch and leads to the *revised simplex* method.

Other improvements in practice are obtained by preprocessing the input LP before starting the solving algorithm. By *simplifying* one eliminates variables whose values are fixed by some of the other variables, as well as empty rows, empty columns, and redundant rows. By *scaling* the rows of the LP matrix are divided by specific values, then the columns are correspondingly scaled, with the goal of improving the numerical stability of the LU decompositions.

We described so far the *primal simplex* algorithm, which moves from one feasible basis for the primal problem to another. In addition there exists a *dual simplex* algorithm that uses feasible bases for the dual problem. In practice the two versions are used alternatively, depending on the specific situation.

The ellipsoid method

As mentioned in the previous section, simplex-based methods are effective in solving most LPs appearing in applications. However, we showed that for particular problems, these methods have exponential complexity.

The first method for solving linear programs to have theoretically proved polynomial time complexity was the *ellipsoid method*, invented by Khachyian [54], and although computationally impractical, it helped in proving many new theoretical results. In the following we will give a brief overview of the algorithm.

The method works by solving the *strict membership problem*: given $m, n \in \mathbb{Z}$, $A \in \mathbb{Z}^{m \times n}$, and $m \in \mathbb{Z}^m$, find a point in $P_< = \{x \in \mathbb{R}^n \mid Ax < b\}$ or conclude that $P_< = \emptyset$.

A symmetric matrix $D \in \mathbb{R}^{n \times n}$ is *positive definite* if $x^T D x > 0$ for all $x \in \mathbb{R}^n$, $x \neq 0$.

An *ellipsoid* with center $y \in \mathbb{R}^n$ is a set $E(D, y) = \{x \in \mathbb{R}^n \mid (x - y)^T \cdot D^{-1} \cdot (x - y) \leqslant 1\}$, where $D \in \mathbb{R}^{n \times n}$ is a positive definite matrix.

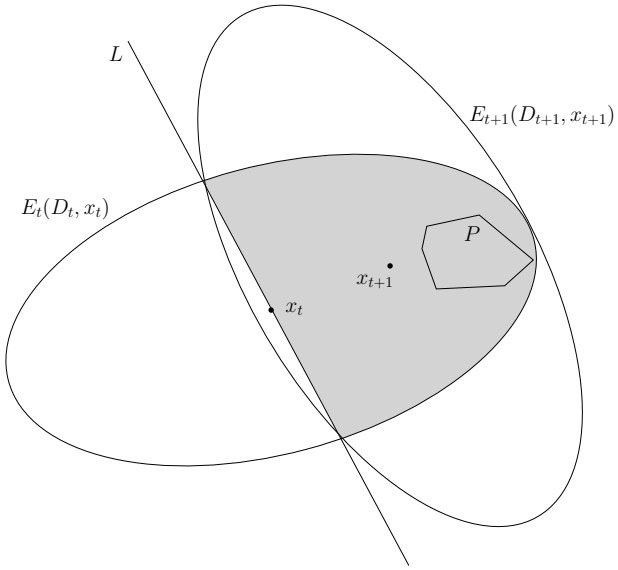


Figure 2.2: The ellipsoid method.

The ellipsoid $E_t(D_t, x_t)$ contains the feasible set P, but its center x_t does not lie in P. We can find a line L through x_t such that P is contained in the shaded half-ellipsoid that L delimits. The new ellipsoid $E_{t+1}(D_{t+1}, x_{t+1})$ contains this half-ellipsoid and has a smaller volume than E_t.

The main idea is to generate a sequence of smaller and smaller ellipsoids that contain the polyhedron P, see Figure 2.2 for a graphical depiction. We start with a suitably chosen $E_0(D_0, x_0)$. At each step t, if the center of the ellipsoid x_t lies in the polyhedron, we found a solution and stop, otherwise x_t violates at least one constraint $a_i^T x \geqslant b_i$, i.e., $a_i^T x_t < b$, and one can find a half-space $\{x \in \mathbb{R}^n \mid a_i^T x \geqslant a_i^T x_t\}$ passing through x_t, such that the polyhedron is contained in the half-ellipsoid given by the intersection of the starting ellipsoid E_t and the half-space. This is called the *separation problem*.

By an important geometric property of ellipsoids, one can find a new ellipsoid $E_{t+1}(D_{t+1}, x_{t+1})$ covering the half-ellipsoid (hence also the polyhedron), where

$$x_{t+1} = x_t - \frac{1}{n+1} \cdot \frac{D_t a_i}{\sqrt{a_i^T D_t a_i}},$$

and

$$D_{t+1} = \frac{n^2}{n^2-1}\left(D_t - \frac{2}{n+1} \cdot \frac{D_t a_i a_i^T D_t^T}{a_i^T D_t a_i}\right).$$

The volume of E_{t+1} is only a fraction of the volume of E_t, more precisely $\operatorname{vol}(E_{t+1})/\operatorname{vol}(E_t) \leqslant e^{-1/2(n+1)}$. This ensures that the method terminates after a polynomial number of steps, namely $O(n^6 \log(nU))$, where U is an upper bound on the magnitude of the integer entries.

The ellipsoid method as presented solves the *feasibility problem*, i.e., deciding whether a linear system $Ax \geqslant b$, $x \in \mathbb{R}^n$ is feasible, or equivalently, whether its associated polyhedron P is nonempty.

To use the method for general optimization problems, observe that an LP $\{\min c^T x,\ Ax \geqslant b\}$ and its dual $\{\max b^T y,\ A^T y = c,\ y \geqslant 0\}$ have optimal solutions if and only if the system $\{c^T x = b^T y,\ Ax \geqslant b,\ A^T y = c,\ y \geqslant 0\}$ is feasible, and we can apply the ellipsoid method to this linear system.

We skip the technical details involved when the LP is not bounded or its matrix does not have full rank, as well as the complexity calculation when the problem data is not integer.

An interesting property of the ellipsoid method is that it can deal with an exponential number of constraints, because the separation problem does not depend on m. This is also visible in the complexity of the method. It is also possible to use the simplex method for this kind of problems by using a technique called *column generation* on the dual problem. We start with a subset of the constraints and solve the LP with simplex, then solve the separation problem for the optimal solution x^*; if x^* is in the polyhedron, it is also the optimal solution for the original problem. Otherwise, we add one or more violated constraints and iterate, with the advantage that the old optimal basis is still dual feasible, hence the next optimal solution is found rather quickly.

Interior-point methods

While the simplex algorithm finds the optimal solution by moving on the boundary of the feasible set, interior-point methods start inside the feasible set and proceed towards the optimal corner. They were introduced to the wider scientific audience by Karmarkar [52] and perform especially well with large scale problems whose matrices are very sparse or have special structure.

We only briefly mention here the most important classes of interior-point algorithms (see Figure 2.3 for an illustration) as they do not play an important role in the reminder of our discussion.

The *affine scaling algorithm* [8, 82] starts with a feasible solution in the interior of the feasible set, then forms an ellipsoid centered at that point and optimizes the objective function subject to the ellipsoid. The optimal solution found is used as the center of a new ellipsoid, and the process is repeated. In this way the method solves a sequence of optimization problems over ellipsoids contained in the feasible set and stops as soon as some optimality tolerance criterion in terms of the duality gap (i.e., the distance between the objective values of a primal and a dual solution) is fulfilled.

It was observed that as soon as it approaches the boundary of the feasible set, the affine scaling algorithm makes smaller and smaller improvements thus requiring a large number of iterations. To alleviate this problem, the potential reduction algorithm [88] uses a *logarithmic barrier function* as objective function that includes, besides a logarithmic term to quantify the duality gap, extra terms

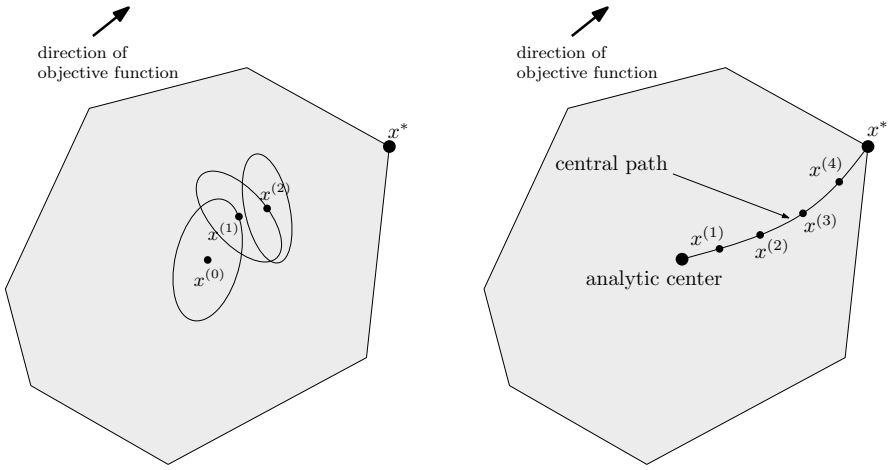


Figure 2.3: Interior-point methods.

On the left, the affine scaling algorithm: each ellipsoid is contained in the feasible set. The optimum solution for an ellipsoid becomes the center of the next ellipsoid. On the right, the path following algorithm starts at the analytic center of the feasible set and moves to the next points given by successive values of t until it reaches the optimal solution x^*.

that penalize being to close to the boundary. This ensures a faster convergence.

The *path following algorithms* [63] use the same barrier approach to ensure that their solution approaches the wanted minimum. Consider the problem $\{\min c^T x,\ Ax \geqslant b\}$, and define the logarithmic barrier function

$$\varphi(x) = \begin{cases} \sum_{i=1}^{m} -\log(a_i^T x - b_i) & \text{if } Ax > b, \\ +\infty & \text{otherwise} \end{cases}.$$

The algorithm will start at the *analytic center* of the feasible set, defined as $\operatorname{argmin}_x \varphi(x)$, and compute successive points

$$x^*(t) = \operatorname*{argmin}_x (t \cdot c^T x + \varphi(x))$$

for $t > 0$ following a *central path* until it reaches the optimum x^*.

Chapter 3

Certifying Feasibility and Objective Value of Linear Programs

In this chapter we describe a method that gives certified answers to the question whether a linear program is feasible or infeasible and computes safe bounds on the objective value, at the same time using expensive rational arithmetic only sparingly.

We develop the theory on which the method is based, and describe the implementation of a software application that we used in order to test it, to conclude with the experiments we performed, and a discussion of the results.

3.1 Introduction

Solvers for linear programs became more and more efficient in recent years, triggering a significant increase in the use of linear programs in various areas of applications. For example, they are used in software and hardware verification [17, 25, 26], where the safety of a system is certified by solving LPs. It is often the case that a problem is not handled by solving a single linear program, but a search tree is computed, where in each node of it a linear program is solved and the feasibility or infeasibility of the LP, or bounds computed on its objective value are used to guide or stop the search; we can mention here, for example, the branch-and-cut algorithms.

State-of-the-art LP solvers rely on floating-point arithmetic. We pointed out in Sections 1.1 and 2.2 that calculations using floating-point numbers can have arbitrarily wrong results, thus possibly causing the invalidation of entire procedures. Our goal is to certify the feasibility or infeasibility of a linear program, and in case the program is feasible, to compute safe bounds on its objective value efficiently.

Notice that we are considering linear programs which typically are solved

while parsing a search tree and hence they are not numerically too complicated, and we want to find a certified answer with a bound close to the correct answer, in most of the cases with no or only negligible overhead. Only in the few remaining examples a slow, exact solver has to be used, or the algorithm has to deal without a certified solution in some other way.

The chapter is organized as follows. We start with a description on the previous work in this area. In Section 3.3 we explain our method, then in Section 3.5 we shortly sketch the possibility of using the solution of a previously solved LP. We give a few details about the implementation of our approach in Section 3.6. In Section 3.7 we report on the experiments we performed to test our approach and discuss the results obtained.

3.2 Previous Work

In this chapter we assume that we are considering a minimization problem, unless otherwise explicitly stated. Previous work in this area goes in two different directions:

Dhiflaoui et al. [27] propose to take the basis computed by the LP solver and certify its feasibility and/or optimality with rational arithmetic. This approach has later been implemented more efficiently by Koch [56] and Applegate et al. [6].

Koch implemented a verification tool called `perPlex` based on the simplex algorithm that similarly to the LP solver SoPlex uses a sparse LU decomposition with Markowitz pivoting. The software takes a linear program and a basis given in terms of basic variables, and computes exactly the primal and dual solutions using rational arithmetic; it can also verify if the given basis is feasible and/or optimal.

Applegate et al. implement a software called *QSopt_ex* that uses the well-known GMP library to perform floating-point calculations with arbitrary precision. They start with the best native precision, then switch to 128 bits and continue to dynamically increase the precision by 50% until they obtain a basis whose corresponding solution satisfies all the constraints. The tool can also verify infeasibility or unboundedness. For their tests they use an LU decomposition with rational arithmetic whose computation is very time-consuming.

This approach has two major drawbacks. First, if the basis is infeasible, nothing can be said about the feasibility of the LP and one has either to increase the precision of the exact solver and solve again, or to switch to an exact solver. Second, it takes much more time to solve the basis system with rational arithmetic than solving the LP, especially for large and complicated LPs. On the positive side, these methods compute very accurate bounds, e.g., if the floating-point solver happens to find the correct basis (as it is the case for most problems) it finds and certifies the optimal solution.

Steffy [79] describes a new method for computing dual bounds for mixed integer programs, called project-and-shift, whose main idea is similar to ours, but

his method needs the exact solution of an LP at the root node of the branch-and-bound tree and an exact LU factorization that typically require a long running time.

A different approach is to compute bounds purely with floating-point arithmetic. This approach was introduced by Neumaier and Shcherbina [69] to compute lower bounds for LPs where all variables are lower and upper bounded and later improved by Keil and Jansson [53]. Still for most of the instance from the linear programming benchmark set Netlib [68] no bound could be computed, that is, for only 40% of them an upper bound could be computed.

We want to extend the floating-point approach to be able to handle linear programs with unbounded variables that are numerically not too complicated, at the same using rational arithmetic only when necessary, without solving large systems of linear equations.

With our approach we are able to compute objective value bounds for 94% of the Netlib instances. For two other benchmark sets that we used we could solve 100% and 97% of the instances, respectively. The relative error of the computed objective bound for 73% of all tested instances is less than 1%. For details we refer to Section 3.7.

3.3 Certifying the Feasibility of an LP

In this section, we describe our approach to certify the feasibility of a linear program. More precisely, we describe an algorithm that either states that the given LP is feasible or it answers that it cannot certify feasibility. In the latter case one can try to certify infeasibility with a similar approach (see Section 3.4). If this fails too, neither the feasibility nor infeasibility of the LP can be certified with our approach and one has to use an exact solver.

3.3.1 The General Idea

We first fix some notation. We are considering a linear system of the form

$$\begin{aligned} A'x &\leqslant b' \\ A''x &= b'', \end{aligned}$$

where $x \in \mathbb{R}^n$. Let $A = \binom{A'}{A''}$ and $b = \binom{b'}{b''}$. For a subset B of the (in)equalities let A_B be the rows of A corresponding to the (in)equalities in B and b_B the corresponding entries of b.

We use $\approx, \preceq, \not\approx$ to denote equations that are evaluated with floating-point arithmetic, i.e., we make our decisions depending on the outcome of some expression evaluated with floating-point arithmetic and typical ε-comparisons but we have no guarantee that the computed values are correct.

Assume we want to certify the feasibility of the linear system

$$\begin{aligned} A'x &\leqslant b' \\ A''x &= b''. \end{aligned}$$

We use a state-of-the-art LP solver to compute a feasible floating-point solution. Typically, we get basis B, i.e., a subset of the (in)equalities of A containing all equations such that $x^* = A_B^{-1} b_B$ has a unique solution that satisfies all inequalities. Notice that this is not the standard form for a linear program and the corresponding basis that are typically discussed in textbooks; we use it as it is better suited to describe our approach. Moreover, it is easy to obtain this kind of an LP and corresponding basis from standard form.

As we use floating-point arithmetic, we get $x^{\approx}$ with $x^{\approx} \approx A_B^{-1} b_B$. We then use a solver for systems of linear equations that gives safe error bounds (see Section 3.3.2) to obtain a vector of intervals $x^{[]} = (x^{\ell}, x^{u}) \in (\mathbb{R} \times \mathbb{R})^n$ with $A_B^{-1} b_B \in x^{[]}$. If all vectors in $x^{[]}$ satisfy all remaining inequalities, we are done. This can be checked easily using interval arithmetic. As the vector of intervals contains typically rather small intervals, we can certify the feasibility if the true vector x^* satisfies the inequalities strictly, which can be easily checked using interval arithmetic. Unfortunately, many linear programs are degenerated, hence some further inequalities are satisfied with equality.

Therefore, we use the LP solver to find an optimal basis B for the linear program

$$\begin{aligned} \max \quad & \delta \\ \text{s.t.} \quad & A'x + \mathbb{1}\delta \leqslant b' \\ & A''x = b'' \end{aligned}$$

i.e., we try to find a feasible point of the linear system that is as far from tight at the inequalities as possible.

This modified approach typically works if $\delta \succ 0$, but clearly fails, if the optimal value of the linear program is 0. Notice that the dual of the linear program above is

$$\begin{aligned} \min \quad & p^T b' + q^T b'' \\ \text{s.t.} \quad & p^T A' + q^T A'' = 0 \\ & p^T \mathbb{1} = 1 \\ & p \geqslant 0 \end{aligned}$$

and hence we get a dual solution of the linear program above that gives us a vector $(p^{\approx}, q^{\approx})$ with

$$\begin{aligned} p^{\approx T} A' + q^{\approx T} A'' &\approx 0 \\ p^{\approx T} b' + q^{\approx T} b'' &\approx 0 \\ p^{\approx} &\not\approx 0. \end{aligned}$$

If there were no floating-point errors, we would know that all inequalities with nonzero multiplier have to be satisfied with equality in any solution of the linear program. This can be seen as follows: assume (p^*, q^*) is a dual feasible solution

and x^* is a feasible solution that satisfies at least one inequality strictly, i.e., $a_i^T x^* < b_i$ for some i with $p_i^* \neq 0$. Then we have

$$\begin{aligned} 0 &= \sum_i p_i^* A_i x^* + \sum_j q_j^* A_j x^* \\ &< \sum_i p_i^* b_i + \sum_j q_j^* b_j \\ &= 0, \end{aligned}$$

which is obviously a contradiction.

We simply assume the correctness of the computations and transform the inequalities with nonzero dual value $p^{\approx}$ to equalities and iterate. We remark that a wrong result of the floating-point computations can lead to transforming too many inequalities to equalities, i.e., *reduce* the feasible region of the LP. Hence these fixings can lead to an infeasible LP although the original LP is feasible, but not the other way around.

Notice that at least one of the new equalities is redundant and we want to remove it. In order to make this removal safe (i.e., so that we do not make an infeasible linear system feasible), we have to check that these equalities are indeed redundant with rational arithmetic. The same holds for redundant equalities that appear in the initial LP formulation. Since the vector $(p^{\approx}, q^{\approx})$ typically contains only a very small number of nonzero entries, the rational arithmetic is not too costly in this step.

Furthermore, if the LP solver returns a basis that contains an equality, this could be caused by a linear dependency among the linear equations, either in the original linear program or due to the transformation, and we have to certify whether this is the case before we remove these equalities; see Section 3.3.3 for details. For the Netlib instances this is the case for 24 of the 95 LPs.

We are aware that one could solve a single linear program to find all inequalities that are always satisfied with equality by using the approach of Freund et al. [33]. Given the linear system $Ax \leqslant b$ and an optimum solution $(x^*, \delta^*, \lambda^*)$ for the linear program

$$\begin{array}{lrcl} \max & \mathbb{1}^T \delta & & \\ \text{s.t.} & Ax + \delta - \lambda b & \leqslant & 0 \\ & 0 \leqslant \delta & \leqslant & \mathbb{1} \\ & \lambda & \geqslant & 1, \end{array}$$

where $\delta \in \mathbb{R}^m$ and $\lambda \in \mathbb{R}$, the always-active constraints are those corresponding to the indices i for which δ_i^* is zero. However, this could lead to rounding errors, and the alternative of solving the LP with exact arithmetic might be prohibitively slow. We did not implement this approach and cannot report on results.

3.3.2 Computing Safe Intervals for the Solution of a System of Linear Equations

In this section we discuss how we compute safe intervals for the solution of a system of linear equations for the case when we are given a nonsingular square matrix A_B. In the next section we discuss the case when our system contains redundant equalities.

In the following we describe Theorem 2.9 which uses real numbers. All computations performed in order to apply the theorem are done with interval arithmetic, where we assume that we can convert rational numbers into safe (i.e., over-approximating) intervals.

In a state-of-the-art LP solver, a highly tuned (in efficiency and numerical stability) algorithm for computing the LU decomposition is implemented. In our computations we use the SoPlex [78, 86] LP solver, since it allows us to access its LU decomposition, in contrast to other solvers like Gurobi [42] or CPLEX [48].

We need the decomposition in order to use a perturbation bound given, among others, by Watkins [85, Th. 2.3.9]. Let $A \in \mathbb{R}^{n\times n}$ be an invertible matrix, $Ax = b$, $b \neq 0$, and consider A and b having been perturbed by E and d respectively, i.e., we have the perturbed system

$$(A + E)(x + h) = b + d.$$

We want to obtain safe bounds for the solution x, that is, to upper bound the error $\|h\|$.

In our case A is the product of the computed triangular matrices L and U, E is the difference between A and the true (rational) basis matrix, b is the floating-point representation of the right-hand side and d the difference between b and the (rational) right-hand side. The main point on the theorem is that the bound is computed based on the condition number of the matrix A, i.e., the approximate basis, and not on the true basis.

Assuming that the condition $\mathrm{cond}(A) \cdot \|E\|/\|A\| < 1$ holds, then

$$\frac{\|h\|}{\|x\|} \leqslant \frac{\mathrm{cond}(A)}{1 - \mathrm{cond}(A) \cdot \|E\|/\|A\|} \cdot \left(\frac{\|E\|}{\|A\|} + \frac{\|d\|}{\|b\|}\right),$$

from where it follows that

$$\|h\| \leqslant \frac{\mathrm{cond}(A) \cdot \|x\|}{1 - \mathrm{cond}(A) \cdot \|E\|/\|A\|} \cdot \left(\frac{\|E\|}{\|A\|} + \frac{\|d\|}{\|b\|}\right). \tag{3.1}$$

As mentioned before, the LP solver SoPlex gives access to its LU decomposition of the basis matrix. We use this decomposition and inequality (3.1) to efficiently compute $\bar{h}$, an upper bound for $\|h\|$, with interval arithmetic. We then "perturb" the solution of $Ax = b$ accordingly (that is, we set $\bar{x}^{[]} := x^{[]} + [-\bar{h}, \bar{h}]$) in order to get safe bounds for x.

We compared two methods for computing $\|h\|$. The first method uses the

value of $\|A^{-1}\|$ computed exactly: we solve the linear systems

$$((A^{-1})^T)_i \cdot L \cdot U = \mathbf{e}_i^T$$

for $1 \leqslant i \leqslant n$, where $\mathbf{e}_i$ is the ith unit vector, and obtain the rows of A^{-1}, which leads straightforward to $\|A^{-1}\|$.

The second method uses an upper bound for $\|A^{-1}\|$ by following the approach given by Higham [46], where different estimates for $\|L^{-1}\|$ of an approximate LU decomposition are used to obtain rigorous error bounds.

We proved in Theorem 2.12 that for a triangular matrix T, the relation $\|T^{-1}\| \leqslant \|M(T)^{-1}\|$ holds, where

$$M(T)_{ij} = \begin{cases} |T_{ij}| & \text{for } i = j, \\ -|T_{ij}| & \text{for } i \neq j, \end{cases}$$

and that $\|M(T)^{-1}\|$ can be computed easily, as shown in Section 2.1.3.

Using

$$\begin{aligned} \|A^{-1}\| &= \|U^{-1} \cdot L^{-1}\| \\ &\leqslant \|U^{-1}\| \cdot \|L^{-1}\| \\ &\leqslant \|M(U)^{-1}\| \cdot \|M(L)^{-1}\|, \end{aligned}$$

we get an upper bound for $\|A^{-1}\|$.

In the experiments we performed, the second method is as expected faster, but using an upper bound for $\mathrm{cond}(A)$ causes the condition $\mathrm{cond}(A) \cdot \|E\|/\|A\| < 1$ to be more often unsatisfied, which leads to more cases where our approach fails; furthermore, the computed safe bound is weaker in this case (see Section 3.7).

3.3.3 Overdetermined Linear Systems

In order to use the method outlined in Section 3.3.2, we have to detect and remove all redundant equalities, i.e., a subset of equalities that are linearly dependent on the remaining equalities and after their deletion the remaining equalities are linearly independent.

Before removing an equality from our system, we have to make sure that it is indeed redundant. For this check, floating-point arithmetic is not sufficient, since a wrong answer would make us remove an equality that is not redundant, and hence possibly enlarge the feasible region. Therefore we have to test redundancy with rational arithmetic.

To show that $Ax = b$ contains a redundant equality one typically transforms the augmented matrix $\bar{A} = [A \mid b]$ into row echelon form. This can be quite an expensive operation, especially when performed with rational arithmetic, since it is as expensive as solving a system of linear equations, i.e., as certifying the feasibility of a basis with rational arithmetic. Therefore we compute the row

echelon form using as few equations as possible.

Consider a linear program of the form

$$\begin{array}{lrcl} \max & \delta & & \\ \text{s.t.} & A'x + \mathbb{1}\delta & \leqslant & b' \\ & A''x & = & b'' \end{array}$$

and assume that its objective function value is zero. Solving this linear program, we get an approximate dual solution (p^*, q^*) with objective function value about 0. The (in)equalities with nonzero dual value are transformed into equalities and the resulting set of equalities contains at least one redundant equality, as explained in Section 3.3.1. Deciding whether the objective function value is 0 and which components of p^* and q^* are 0 is done based on the floating-point results with some suitable ε, since the correctness of the method does not depend on the correctness of these tests. Indeed, a wrong set can only lead to the answer "unknown". Furthermore the set of (in)equalities with nonzero dual values is expected to be very small and hence we can afford to compute their row echelon form with rational arithmetic. Here we have to use rational arithmetic, as we are only allowed to remove equalities that are truly redundant.

Now assume that the objective function value of the above linear program is different from zero, hence we are in the last iteration of our algorithm. If the LP solver does not indicate that an equality is redundant (by having the equality in the basis), we are fine. Otherwise, we have to transform the set of all equations into row echelon form. If there are many equalities in the last LP, this can take a while. We remove all equalities that are proved to be redundant and use the method outlined in Section 3.3.2 to prove that the solution of the remaining basis satisfies all basic inequalities. Notice that we have to recompute the LU decomposition.

3.3.4 Decreasing the Number of Iterations

The initial experiments with our approach have shown that δ remains zero for many iterations (see Section 3.7), resulting in a long running time of our approach. Yet we realized that in many iterations, a single inequality implies all bounds of the occurring variables. In this case, we simplify the linear program, i.e., we set all occurring variables to the respective bounds and remove the inequality.

More exactly, let $\ell_i \leqslant x_i \leqslant u_i$ be the bound on x_i (with $\ell_i, u_i \neq \pm\infty$). Then a constraint $a_1x_1 + a_2x_2 + \cdots + a_nx_n \sim b$ can be simplified:

- when $\sim$ is $\leqslant$, if $\min(a_1\ell_1, a_1u_1) + \cdots + \min(a_n\ell_n, a_nu_n) = b$; we have

$$\min(a_i\ell_i, a_iu_i) = \begin{cases} a_i\ell_i, & \text{if } a_i \geqslant 0, \\ a_iu_i, & \text{if } a_i < 0 \end{cases}$$

- when $\sim$ is $\geqslant$, if $\max(a_1\ell_1, a_1u_1) + \cdots + \max(a_n\ell_n, a_nu_n) = b$; we have

$$\max(a_i\ell_i, a_iu_i) = \begin{cases} a_iu_i, & \text{if } a_i \geqslant 0, \\ a_i\ell_i, & \text{if } a_i < 0 \end{cases}$$

- when $\sim$ is $=$, if either one of the implied $\leqslant$ or $\geqslant$ constraints can be simplified as above.

These verifications are first performed using floating-point arithmetic; in case the answer is affirmative, a double-check using rational arithmetic takes place, and only if it also ends up positive, the constraint is deemed as simplifiable.

For any constraint that could be simplified after carrying out verifications as described before, the variables contained are set to their respective bound used in the simplification check. In this way the constraint becomes redundant and is finally removed.

3.4 Certifying Infeasibility and Computing Bounds

A linear system $\{Ax \leqslant b\}$ is infeasible if and only if the system

$$\begin{array}{rcl} y^TA & = & 0 \\ y & \geqslant & 0 \\ y^Tb & = & -1 \end{array}$$

is feasible (by Farkas' Lemma). Hence, we can use the approach described above to certify infeasibility of an linear program.

To prove an upper bound for the LP we do the following: At the end of our algorithm for certifying the feasibility, we have transformed some inequalities to equalities. More precisely, if all floating-point computations are almost correct we compute the equality space, otherwise a (hopefully small) superset of it. Let

$$\begin{array}{lrcl} \max & \delta & & \\ \text{s.t.} & A'x + \mathbb{1}\delta & \leqslant & b' \\ & A''x & = & b'' \end{array}$$

be the final LP and $\bar{x}^{[]}_\delta$ be the certified safe interval for the solution. We solve the LP

$$\begin{array}{lrcl} \min & c^Tx & & \\ \text{s.t.} & A'x & \leqslant & b' \\ & A''x & = & b'' \end{array}$$

and compute a certified safe solution $\bar{x}^{[]}_s$. We consider the segments from $\bar{x}^{[]}_s$ to $\bar{x}^{[]}_\delta$, i.e., $(1-\lambda)\bar{x}^{[]}_s + \lambda\bar{x}^{[]}_\delta$ for $0 \leqslant \lambda \leqslant 1$. We would like to have an objective value bound as good as possible (i.e., as close to the solution of the original LP as possible), or equivalently, to have a small λ. We compute the smallest λ^* so that each possible

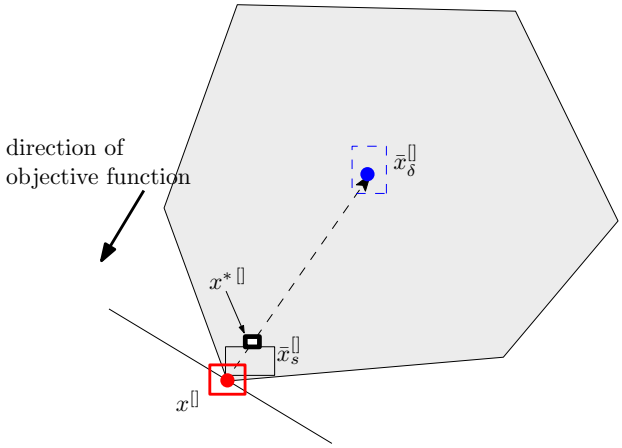


Figure 3.1: Computation of a safe bound on the objective function

resulting point $x^* \in (1-\lambda^*)\bar{x}_s^{[]} + \lambda^*\bar{x}_\delta^{[]}$ on one of the segments satisfies $A'x^* \leqslant b'$. Finally, we compute the best objective function value corresponding to a point in $(1-\lambda^*)\bar{x}_s^{[]} + \lambda^*\bar{x}_\delta^{[]}$.

3.5 Warm Start Using a Previous LP Solution

In this section we sketch the possibility of a warm start, i.e., we want to certify the infeasibility or compute a bound of a linear program and assume that we have already done so for a linear program which has one or only a few constraints less. In this case it makes sense to start with the old basis as the simplex algorithm typically only needs a few further steps to find an optimal basis.

In principle, the same can be done for our approach. Instead of storing the basis for a single linear program, we have to store two bases: one for the computation of the point in the (computed) relative interior and one for computing the optimal point within the (computed) equality space. Notice that in this way it is possible for the equality space to become larger. In this case, it can happen that we have more than one iteration in our approach.

We did not implement the possibility for a warm start and cannot report on results.

3.6 Implementation

In order to evaluate our method, we implemented it in C++. As linear program solver we used SoPlex [78, 86], slightly modified for our purposes as we will show in the following. For computations involving multiple precision we used the GMP library [38]. For interval arithmetic and basic linear algebra operations we chose the Boost library [15].

The source code amounts to more than 31,000 lines of code implementing some twenty classes and interfaces.[1] In the next sections we give more details on specific classes and design decisions.

3.6.1 Main Classes

To make our implementation modular and avoid possible naming collisions when using them as part of other approaches (see, e.g., Chapter 4), we wrote all our classes inside the namespace `flp`.

Class `LPProblem`

This class is the working horse of our implementation. It models a linear problem and the most usual methods associated with it, and additionally implements special routines.

The class stores its data using a member of type `LPObjective`, a map of `LPConstraints` indexed by constraint names, and a set of `LPBounds`. It can read linear programs in both MPS and LP file formats; each coefficient and free term is stored both as C++ `double` and as GMP `mpq_class` variable.

The most important methods of the class `LPProblem` are:

`readMPS()`
: Reads a linear program in the standard MPS format (used, e.g., by the LP solver CPLEX).

`readLPF()`
: Reads a linear program in LP format. Since in this format it is possible not to specify any name for constraints, the method also assigns them an unique name; this procedure is implemented in the `getUniqueConstraintName_()` method.

`modifyWithDelta()`
: The first step of our approach to certify the feasibility of a linear program is to transform the input problem to a so-called δ-modified problem (see Section 3.3.1). The method takes care of this task and executes the appropriate modifications by updating the relevant data structures. For more details see Section 3.6.5.

`getPermutedBasisMatrix()`
: We use our `LPProblem` object in connection with an instance of the LP solver SoPlex instance. That instance does not necessarily keep the same order of rows (i.e., constraints) and columns (i.e., variables) as our instance. After solving the linear program with SoPlex, we might want to interrogate

[1] The source code is available online at `http://www.informatik.uni-mainz.de/~dumitriu/work/flp`.

the solver in order to obtain its basis matrix. To this end, we have to take care and reorder the rows and columns returned by SoPlex such that they correspond to our own internal representation.

`getPermutedRhs()`
Serves a goal similar to that of `getPermutedBasisMatrix()`, and returns the correctly ordered right-hand side corresponding to the constraints in a SoPlex basis.

`isConstraintSafelySatisfied()`
After we compute our unscaled, interval safe solution by means of the `computeSafeBound()` method inserted in SoPlex's class `CLUFactor` (see Section 3.6.4), and the method `EquilibriumScaler::unscaleSolutionVector()`, we can check if some constraint is safely satisfied by the interval solution. This amounts to computing an interval for the left-hand side of the constraint, and checking if the (in)equality described by the constraint holds. We recall that in interval arithmetic, $[a, b] \leqslant [c, d]$ if and only if for all $x \in [a, b]$, and for all $z \in [c, d]$, it holds $x \leqslant y$. The method receives an iterator to one of the constraints, a list of the fixed variables, and the interval solution. Besides a yes/no answer, it returns a floating-point value showing by how much is the constraint violated.

We want to underline shortly the importance of choosing the appropriate data structures, with effects that can be best observed in the running time. An early implementation of `LPProblem` used a vector of `LPConstraint`s and a vector of `LPBound`s, and `LPConstraint` used a vector of `LPMonomial`s. Among the operations mostly used is finding a constraint by its name, and a bound by the name of the variable to which it refers. Consequently, the proxy methods `findConstraintByName()`, `findBoundByVariable()`, `findVariable()`, and the utility method `getUniqueConstraintName_()` executed linear searches through vectors. We replaced as mentioned these vectors by a map and two sets, respectively, thus causing searches to be executed in logarithmic time. As a result we could experience a significant improvement in the running time, especially in those parts of our application that make heavy use of these methods.

Class `Timekeeper`

Timing an application is seemingly a task that is taking for granted in most applications. However, implementing a sensible approach is not obvious, as system calls differ from platform to platform and even from compiler to compiler, and they have different accuracies and granularities.

This utility class provides methods that are used to measure the time elapsed in various stages of our application. It tries to be as portable as possible by detecting the environment it runs in, and using the appropriate system calls. For example, under Unix it uses the `times()` call in `sys/times.h`, and under

Windows it uses the `clock()` function. Such a timekeeper can be started, stopped, paused and resumed again, and is able to return the elapsed user, system, and real time.

Class `Logger`

To record the progress of an application, it is customary to print log messages to the console or to a file. These messages are especially important for debugging and turning them on and off usually creates a nontrivial maintenance problem. It is important to be able to switch them off programmatically without excessive overhead, to have several logging levels, and to select the desired one quickly, especially when developing a large application like ours.

The class `Logger` implements an eight-level logging mechanism whose granularity can be set (or it can be completely turned off) at compile time. Moreover, the logger is implemented by making heavy use of preprocessor macros, which means that there is no extra overhead created by the logging code that pertains to inferior logging levels, since disabling them at compile time removes the code completely. The logger can write to the screen or to a file.

Class `Utilities`

This class implements routines that are used in most of the other classes. Besides the usual methods for safer comparisons between floating-point numbers, it includes conversion functions from a string to `mpq_class`, and from `mpq_class` to `boost::numeric::interval`. It also provides linear algebra functions and numerical algorithms.

The most important methods of the class `Utilities` is `solveGJ()`:

`solveGJ()`
: This function implements the Gauss–Jordan method for solving a linear system (see Section 2.1.2) in the form $Ax = b$. It receives as parameters the matrix A as a `SparseMatrix`, the right-hand side b as a `SparseVector`, and the size of b.

  The method returns either 0, 1, or 2, corresponding to no solution, one solution, and infinitely many solutions, respectively. If there is a unique solution, it returns additionally this solution. Moreover, it returns a vector containing the indices of redundant rows in case $\operatorname{rank}(A) < m$, where m is the number of rows of A; we use these indices subsequently to remove the redundant rows, an important step in our certification approach (see Section 3.3.3).

Class `StringUtilities`

This is another utility class for interacting with strings. Given the textual nature of the input, the methods of this class are used rather intensively, especially when

reading the input files and storing them as linear programs. It fulfills some basic needs like converting a string to lowercase or uppercase, trimming a string, and splitting a string into tokens upon the occurrence of specific characters. Another use is to ensure that a string (representing a coefficient) starts with a sign, possible by adding a leading +, and that there is a coefficient before every variable in the input, possible by adding a leading 1.

Class `SoPlexInterface`

This is a helper class that translates an `LPProblem` into an instance of class `SPxLP` in SoPlex. During the execution of our algorithm, at each step we transform a linear program as described in our approach; these transformations are made on our internal `LPProblem` instance as it possesses specifically designed methods that allow for programmer-friendlier, more efficient manipulation of its internals. As we use the linear solver SoPlex to effectively solve the linear programs resulting in each iteration, this class helps us to pass the problem to the solver. To this end, it provides methods for transforming the objective, a constraint, or a bound. Finally there exists a wrapper method `L2P_transformLPProblem ToSoplexProblem()` that takes an `LPProblem` including constraint and variable names, and builds a complete SoPlex linear problem.

Class `FlpChecker`

This is the most important class of those that specifically implement our approach. It provides methods for reading an input linear program, setting the parameters (e.g., maximum allowed running time, maximum allowed number of iterations), checking the feasibility or infeasibility of a linear program, and obtaining a bound on its objective bound (should the LP be proven feasible). Moreover, there are utility functions for printing various sensible information, i.e., the solution, the Farkas proof, or the values of the slack variables.

The most important methods of the class `FlpChecker` are:

`checkFeasibility()`
: This method implements the steps needed for verifying feasibility according to our approach. We give more details in Section 3.6.5.

`checkInfeasibility()`
: This method tests the infeasibility of the given linear program. To this end the Farkas system is constructed by calling the method `compute FarkasLP_()`, and we try to certify its feasibility as before. We give more details in Section 3.4.

`computeFarkasLP_()`
: This method constructs the Farkas system corresponding to the given linear program using the Farkas' lemma for general form LPs (see Sec-

tion 2.3.3). Additionally, the variables in the Farkas system corresponding to infinite bounds are set to zero.

`computeSecondLP_()`
: In order to compute a safe bound on the objective value of the linear program, we use the approach described in Section 3.4, for which we need to compute a second LP based on the original one; this method constructs the second LP.

`tightenConstraints_()`
: In each iteration of our approach, an objective value of $\delta = 0$ triggers the tightening of the constraints, i.e., transforming those inequalities whose corresponding nonzero dual variable have a nonzero value to equalities. Subsequently the redundant rows (computed by the Gauss–Jordan eliminator implemented in `Utilities::solveGJ()`) are removed.

`computeObjectiveValueBound_()`
: To compute the safe objective value, this method calls the function that constructs the necessary second linear program, then computes the value according to the algorithm described in Section 3.6.6.

`getComputedSafeBound_()`
: This function returns the bound computed using the LU decomposition of SoPlex. To this end, first the linear program is scaled with our `EquilibriumScaler`, then we form the basis linear system corresponding to our stored LP by permuting the rows and columns according to the SoPlex basis. This basis system is passed to `computeSafeBound()` and is used together with SoPlex's own LU decomposition to compute a safe interval solution.

3.6.2 Sparse Structures

Very often in our application we need to work with vectors and matrices that are sparse. In these cases using general structures and algorithms written for dense instances will incur a severe performance penalty. A possible alternative is to use one of the special libraries written for sparse linear algebra, e.g., Netlib's LAPACK [5] or Boost's uBLAS. However, these libraries are too heavyweight for our typical usage scenarios. For these reasons, we decided to implement our own classes, namely `SparseVector` and `SparseMatrix`.

We want to use these classes in different contexts, i.e., with different entry types, e.g., plain C++ `double`, Boost `boost::numeric::interval`, or GMP `mpq_f` and `mpq_class`. The most natural way of achieving this is to transform both class to templates. At compile time these templates will be instantiated to classes for the specific underlying type, thus avoiding code and effort duplication while preserving the same running time performance as for specifically written classes.

Class `SparseVector<T>`

This class uses for storage a C++ array of entries. An entry is a C `struct` containing the logical index of the stored vector element, and the element value itself, which is of the same type `T` as the template argument. The class manages the memory directly by allocating it dynamically depending on the number of elements in the vector, i.e., it allocates new memory when the vector expands. The usual manipulation functions are available, e.g., add, remove, get, set, and subscript operator. Additionally, there exists a method for computing the dot product with another vector, and a sorting function that rearranges the elements internally according to the logical indices.

Class `SparseMatrix<T>`

The data is stored internally as a C++ array of `SparseVector`s. Beside the usual manipulation and querying functions, the class provides a method for computing the dot product of a matrix row to another vector, and a method to sort each row as explained for `SparseVector`. Furthermore there are two linear algebra methods:

`solveLeft()`
: Solves by forward substitution the linear system $x^T L = v^T$, where L is a lower triangular matrix.

`solveRight()`
: Solves by forward substitution the linear system $Lx = v$, where L is a lower triangular matrix.

Both methods are used in the function `computeSafeBound()` that we added to SoPlex's class `CLUFactor`, namely for the computation of the inverse of the basis matrix using its LU decomposition (see Section 3.6.4). Since SoPlex stores both decomposition matrices as lower triangular (with U being transposed), we do not need similar methods for solving upper triangular linear systems.

3.6.3 Speeding up Interval Computations

The IEEE 754 standard does not require the rounding mode to be set for each operation, consequently most of the floating-point units only provide instructions to set the rounding mode for *all* subsequent operation; for example, the C compiler provides two calls `fegetround()` and `fesetround()`.

When doing interval computations, the Boost Interval library executes a number of operations on numbers of the underlying type. For example, since $[a, b] + [c, d] = [a + c, b + d]$, computing the interval sum amounts to computing two sums with scalars. Such an interval addition can be done by setting the rounding direction only once. If exact operations are required, between every operation (e.g., *two* interval additions) the rounding mode has to be stored, then

set, only to be restored after the operation; this is called *protected rounding* mode. Unfortunately, compilers usually do not optimize away this code; the large amount of function calls for getting and setting the rounding mode causes a very significant penalty in the running time.

For this reason, the Boost Interval library allows the user to explicitly tell the library that the interval operations do not need to be protected, by creating a rounding object of type `I::traits_type::rounding` and using intervals of type `interval_lib::unprotect<I>::type`, where `I` is the initial, unprotected interval type. Of course, then the user alone is responsible for setting the rounding mode, especially for the floating-point operations coming up in-between.

We noticed that in our application the calls to `fegetround()` and `fesetround()` amounted to about 35% of the running time. Therefore, we decided to implement templates that unprotect the rounding mode and work both with scalar types, and with Boost intervals:

```
// non−interval case
template<typename T>
struct fast_numbers {
    struct context {};
    typedef T type;
};

// interval case
template<typename T>
struct fast_numbers<boost::numeric::interval<T> > {
#ifndef FLP_USE_PROTECTED_INTERVALS
    typedef typename boost::numeric::interval<T>::
        traits_type::rounding context;
    typedef typename boost::numeric::interval_lib::
        unprotect<boost::numeric::interval<T> >::type type;
#else
    struct context {};
    typedef boost::numeric::interval<T> type;
#endif
};
```

Afterwards, the defined templates can be used to execute rounding-unprotected, fast operations, for example

```
typedef boost::numeric::interval<double> boost_interval;
fast_numbers<boost_interval>::context context;
typedef fast_numbers<boost_interval>::type U_boost_interval;
U_boost_interval unprotIntv;
// ...
```

or as template argument in other classes, e.g., `SparseMatrix<T>`:

```
typename fast_numbers<T>::context context;
typedef typename fast_numbers<T>::type U;
U unprotVar;
// ...
```

The rounding mode is automatically saved at the creation of the rounding object, and restored at the rounding object destruction.

Because testing whether a variable is zero is a routine called quite often in our code, we also implemented a template `IsZero<T>` with an explicit specialization for `mpq_class`, and a partial specialization for `boost::numeric::interval<T>` that uses that `fast_numbers` template. This also benefits from the fast, unprotected interval operations.

With these changes, the calls related to getting and setting the rounding mode amount to less than 0.1% of the running time.

3.6.4 Implementation of Safe Bounds

In order to implement our approach for computing a safe bound on the objective value, we had to modify the SoPlex source by adding code to two of its files, so that we could access its LU decomposition.

SoPlex implements in class `CLUFactor` a sparse LU decomposition using dynamic Markowitz pivoting. To ensure encapsulation, most of the structures used in the computation have a `protected` visibility modifier, which means that these structures are accessible only in derived classes. SoPlex contains a preconfigured LP solver in the shape of the `SoPlex` class which accommodates a member derived from `CLUFactor`, but this makes `CLUFactor`'s protected members unavailable. We need to access these structures to compute our bound, specifically the L and U matrices, the row and column permutation matrices, and the array of pivot elements.

To make our method completely external and not modify SoPlex's source code, we would need getters for the structures storing the decomposition that we need to access; these functions are at the moment not provided by the SoPlex API.

For this reason we modified the class `CLUFactor` by adding a method called `computeSafeBounds()`. The method takes four arguments, namely the basis matrix, the right-hand side vector, the destination array for the solution and a parameter that mentions which method shall be used for computing the norm of the inverse matrix, i.e., exact computation or the Higham estimation (see Section 2.1.3). The elements of these containers are of type `boost::numeric::interval` with base type `double`.

All matrices and vectors used in this function, with the exception of those that are intrinsically dense, are sparse structures obtained by instantiating the templated classes `SparseVector<T>` and `SparseMatrix<T>` (see Table 3.1), where `T` is the Boost `boost::numeric::interval` using `double` as underlying type.

Variable	Text	Note	Relation
`MAT`	$A^{[]}$	sparse	$A = LU$ (or $PAQ = LU$)
`RHS`	$b^{[]}$	sparse	
`x`	$x^{[]}$	sparse	$A^{[]}x^{[]} = b^{[]}$
`Lmat`	L	sparse, lower	
`Umat`	U^T	sparse, lower (transposed)	
`LUSmat`		sparse	$LUS_{mat} = L_{mat}\,U_{mat}$
`E`	E	not explicitely computed	$\|E\| = \|LUS_{mat} - L_{mat}\,U_{mat}\|$

Table 3.1: Main variables used in `computeSafeBounds()`

Wherever it made sense, we used for performance reasons the version of the intervals with the rounding protection turned off (see Section 3.6.3).

Since the preconfigured solver contains in fact a member derived from the class for the LU decomposition, namely `SLUFactor`, and this is actually the variable we can access in our code, we also had to modify this class. The changes were minimal and consisted in defining a function with the same name `computeSafeBounds()` and with an identical signature, which simply calls its namesake in `CLUFactor`.

The basis matrix and the right-hand side vector passed to the method need to be scaled. To this end, we implemented a scaler class called `EquilibriumScaler` that mimics the behavior of SoPlex's default scaler in class `SPxEquiliSC` and performs equilibrium scaling of the rows and columns [21]. In this procedure the rows are first scaled to have unit infinity norm, then the columns, or the other way around; the first direction to be scaled is that for which the maximum ratio between the absolute largest value and the absolute smallest value is smaller.

Code details for `computeSafeBounds()`

In the following, all vectors are column vectors (in $\mathbb{R}^{n\times 1}$), therefore all transposed vectors are row vectors (in $\mathbb{R}^{1\times n}$).

In Tables 3.1 and 3.2 we describe some of the most important variables and function used in this method.

Computation of $\|A^{-1}\|$

We have $A^{-1}A = I$. Denote $X = A^{-1}$ and let x_i be the ith column of X, and $\mathbf{e}_i$ the ith unit vector:

$$\begin{aligned} XA &= I \\ x_i^T A &= \mathbf{e}_i^T \\ x_i^T LU &= \mathbf{e}_i^T \qquad\qquad \text{let } y_i^T := x_i^T L \end{aligned}$$

Function name	Description
`solveRight()`	solves $Lx = b$ (L lower)
`solveLeft()`	solves $x^T L = b^T$ (L lower)

Table 3.2: Main functions used in `computeSafeBounds()`

$$
\begin{aligned}
y_i^T U &= \mathbf{e}_i^T \\
U^T y_i &= \mathbf{e}_i && \rightarrow \text{call } \texttt{Umat.solveRight}(\mathbf{e}_i, y_i) \\
x_i^T L &= y_i^T && \rightarrow \text{call } \texttt{Lmat.solveLeft}(y_i^T, x_i^T)
\end{aligned}
$$

We obtain the rows of A^{-1} as the vectors x_i^T, and we can compute easily $\|A^{-1}\| = \max_{1 \leqslant i \leqslant n} \sum_{j=1}^{n} |x_{ij}|$.

SoPlex uses the decomposition $PAQ = LU$ (P row permutation matrix, Q column permutation matrix). In this case we have

$$
\begin{aligned}
A &= P^{-1} L U Q^{-1} \\
&= P^T L U Q^T && (P, Q \text{ orthogonal} \rightarrow P^{-1} = P^T, Q^{-1} = Q^T) \\
\|A\| &= \|P^T L U Q^T\|
\end{aligned}
$$

Multiplying (on the left or right) with a permutation matrix does not change the value of the ∞-norm (since the rows are merely computed in the permuted order, and their entries remain the same). Since P^T and Q^T are permutation matrices, it follows that $\|A^{-1}\| = \|LU\| = \|A\|$, so we can continue to compute $\|A^{-1}\|$ as in Section 3.6.4.

The full computation of $\|A^{-1}\|$ would otherwise be:

$$
\begin{aligned}
X A &= I \\
x_i^T P^T L U Q^T &= \mathbf{e}_i^T \\
x_i^T P^T L U &= \mathbf{e}_i^T Q && \text{let } y_i^T := x_i^T P^T L \\
y_i^T U &= \mathbf{e}_i^T Q \\
U^T y_i &= Q^T \mathbf{e}_i && (\text{row permutation; call } \texttt{Umat.solveRight}(Q^T \mathbf{e}_i, y_i)) \\
x_i^T P^T L &= y_i^T && \text{let } z_i^T := x_i^T P^T \\
z_i^T L &= y_i^T && \rightarrow \text{call } \texttt{Lmat.solveLeft}(y_i^T, z_i^T) \\
x_i^T P^T &= z_i^T \\
x_i^T &= z_i^T P
\end{aligned}
$$

$$x_i = P^T z_i \quad \text{(row permutation)}$$

Computation of x

We have $Ax = b$, or equivalently:

$$\begin{aligned} LUx &= b && \text{let } y := Ux \\ Ly &= b && \rightarrow \text{call } \texttt{Lmat.solveRight}(b, y) \\ Ux &= y \\ x^T U^T &= y^T && \rightarrow \text{call } \texttt{Umat.solveLeft}(y^T, x^T) \end{aligned}$$

If $PAQ = LU$ (P row permutation matrix, Q column permutation matrix), we solve the system $Ax = b$ as follows:

$$\begin{aligned} Ax &= b \\ P^T LUQ^T x &= b \\ LUQ^T x &= Pb && \text{(row permutation of } b\text{; let } y := UQ^T x) \\ Ly &= Pb && \rightarrow \text{call } \texttt{Lmat.solveRight}(Pb, y) \\ UQ^T x &= y && \text{let } z := Q^T x \\ Uz &= y \\ z^T U^T &= y^T && \rightarrow \text{call } \texttt{Umat.solveLeft}(y^T, z^T) \\ Q^T x &= z \\ x &= Qz && \text{(row permutation of } z\text{)} \end{aligned}$$

3.6.5 Certifying Feasibility

Our implementation consists of several phases corresponding to the steps of our approach as described in the previous sections. In the following we provide more details for each phase.

Preprocessing phase

The linear program (in MPS or LP format) is read from a file on disk, the coefficients and bounds in the file are processed and stored both as regular `doubles` and as GMP `mpq_class` rationals. The LP problem is first transformed into a δ-modified problem as follows:

1. The objective function is changed to max δ.
2. As we introduced a new variable δ, we add new bounds for it, namely $\delta \geqslant 0$ and $\delta \leqslant 1$.
3. The inequality constraints are transformed as follows: $\mathcal{C}_i \leqslant b_i$ becomes $\mathcal{C}_i + \delta \leqslant b_i$, and $\mathcal{C}_i \geqslant b_i$ becomes $\mathcal{C}_i - \delta \geqslant b_i$.
4. Bounds referring to a variable that only appeared in the old objective function are removed. The SoPlex solver is instructed to raise an error when it encounters bounds for a variable that had not been declared previously.
5. The remaining bounds are transformed to so-called "monomial constraints", i.e., $x_i \leqslant u_i$ becomes $x_i + \delta \leqslant u_i$, and $x_i \geqslant \ell_i$ becomes $x_i - \delta \geqslant \ell_i$. This is done because we need to allow some 'looseness', at most δ, for the bounds, too. Of course, the bound for δ and the fixed bounds are not affected by this process.
6. For any variable x_i for which no explicit bound is given, a new bound $x_i - \delta \geqslant 0$ is introduced. This is done because in both MPS and LP formats, a variable x_i for which no explicit bound is given is considered to have default bounds $0 \leqslant x_i < +\infty$.

Iteration phase

After the preprocessing phase is finished, we proceed in successive iterations. For the first iteration, the δ-modified version of the original problem as given by the preprocessing phase is used. The program stops either when one of the intermediate linear programs could not be solved to optimality, or when we obtain a nonzero solution for δ, or when the preset maximum number of iterations is reached.

An iteration of the program consists of the following steps:

Transforming constraints to bounds

Monomial constraints as defined above are transformed to bounds. A constraint of form $ax_i \pm \delta \sim b_i$ or $ax_i \sim b_i$ (with $a \neq 0$) is replaced with the bound $x_i \sim \frac{b_i}{a}$; depending on the sign of a, the inequality sense could be reversed. We do this in order to be able to apply our approach and simplify the problem fixing the variables, see next step.

Problem simplification

We check whether the problem can be simplified by setting the variables to their respective bounds. In some cases, in order to be satisfied, a constraint forces all the variable that it contains to be fixed to one of their bounds. Such simplifications take place only after they are checked with exact arithmetic, as described in Section 3.3.4.

Transforming bounds to constraints
Bounds are again transformed to monomial constraints as in step 5 of the preprocessing phase, to be able to pass to SoPlex the correct linear program, where the nonfixed columns are allowed some deviation of at most δ.

Solving with SoPlex
We use the `SoplexInterface` class to transform the linear program as stored in the `LPProblem` instance into an instance of SoPlex's `SPxLP` class, then we load the problem transformed through the previous operations in SoPlex and solve it.

Tightening constraints
If the objective value (i.e., δ) is zero, the constraints corresponding to dual variables with nonzero value are made tight and δ is removed from them. We then form a linear systems consisting of those constraints, and solve it with the Gauss–Jordan eliminator implemented in `Utilities::solveGJ()`. This solver also returns the redundant equalities, i.e., constraints, which are subsequently removed from the linear program (see Section 3.3.3).

Final phase

In case the iteration phase ended up with a nonzero δ, we use the linear program from the last iteration and the LU decomposition of its basis matrix as given by SoPlex to compute safe intervals on the solution of the linear program.

If the condition needed in the computation of the perturbation bound (see Section 3.3.2) does not hold, we report that our approach failed, otherwise we can certify that the linear program is infeasible. At this point we can attempt to compute a safe bound on the objective value.

3.6.6 Computing a Safe Bound for the Objective Function

We build the second LP as we described in Section 3.4 and compute its certified solution $\bar{x}_s^{[]}$. Again, if the condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold, we cannot go on and stop without reporting any safe bound. Otherwise, in order to compute λ^*, we need the solutions of the two LPs to contain the same variables. Due to the construction of the linear program, some variables might be present just in the second LP if in the original LP they had appeared only in the objective; we set these variables in $\bar{x}_\delta^{[]}$ to the values they have in $\bar{x}_s^{[]}$.

We know that the solution for the last delta-LP safely satisfies each constraint, i.e., $a_i^T \bar{x}_\delta^{[]} \leqslant b_i$. This is not necessarily true for the solution $\bar{x}_s^{[]}$ of the second LP; should this be the case, we are done, i.e., the constraint is satisfied for all points on the segment from $\bar{x}_s^{[]}$ to $\bar{x}_\delta^{[]}$. Otherwise, we compute λ_i as follows:

$$a_i^T((1-\lambda_i)\bar{x}_s^{\square} + \bar{x}_\delta^{\square}) \leqslant b_i$$
$$a_i^T\lambda_i\bar{x}_\delta^{\square} - a_i^T\lambda_i\bar{x}_s^{\square} \leqslant b_i - a_i^T\bar{x}_s^{\square}.$$

If $\bar{x}_s^{\square}$ violates the constraint, the only point within it that certainly violates it is its upper end, so we can use this point safely in computing λ_i. On the other hand, the worst case would be a large λ_i, which happens for the upper end of $\bar{x}_\delta^{\square}$. It follows that for less-than constraints we can compute λ_i as

$$\lambda_i \geqslant \frac{b_i - upper(a_i^T\bar{x}_s^{\square})}{upper(a_i^T\bar{x}_\delta^{\square}) - upper(a_i^T\bar{x}_s^{\square})}.$$

For a greater-than constraint, the same reasoning applies by replacing upper ends with lower ends, thus we can compute

$$\lambda_i \geqslant \frac{b_i - lower(a_i^T\bar{x}_s^{\square})}{lower(a_i^T\bar{x}_\delta^{\square}) - lower(a_i^T\bar{x}_s^{\square})}.$$

Using these formulas we compute for each constraint the corresponding λ_i and denote $\lambda^* = \max_i \lambda_i$, then we use λ^* to compute a "safe overall solution" $x^{*\square}$ for the original LP as

$$x^{*\square} = (1-\lambda^*)\bar{x}_s^{\square} + \lambda^*\bar{x}_\delta^{\square}.$$

We return the safe bound on the objective function of the original LP as the lower end of $c^T x^{*\square}$ for a maximization problem, respectively its upper end for a minimization problem.

3.7 Experiments

We tested our approach on three benchmark sets. The detailed numerical results of our experiments can be found in Appendix A. In the remainder of this section we present shortly each benchmark set and comment on the results obtained.

The experiments were conducted on a computer with two 6-core 64-bit Intel Core i7-970 processors at 3.2 GHz (only one core was used) and 12 GB of RAM, running Linux with kernel version 2.6.38. We used SoPlex version 1.5.0, the GNU `g++` 4.5.2 compiler with the optimization flag `-O2` turned on, and the libraries GMP 4.3.2 and Boost 1.42.0.

The Netlib benchmark set

The Netlib LP collection [68] has been introduced in 1985 [37] and is one of the most well-known benchmark sets for linear programming. It consists currently of

95 instances whose origins can be traced back to problems appearing in practice in various application areas, collected over a long period.

In their study Ordóñez and Freund [72] show that most of the Netlib problems are ill-posed, i.e., they have an infinite condition measure. An infinite condition measure means that the distances to primal and dual infeasibility are zero. The distance to infeasibility is the norm of the smallest perturbation of the LP that results in an empty feasible set. Specifically, the number of instances that are ill-posed amounted to 71% without any preprocessing, and to 19% after preprocessing was performed using the LP solver CPLEX.

In Table A.1 we summarize our main results. For each of the Netlib instances (for which we give the name and some statistics on its size), we report on the number of iterations of our approach, the number of iterations needed without the simplification of the LPs, the safe bound on the objective function value and the running time for both settings, i.e., using the exact computation of $\|A^{-1}\|$ compared to using an upper bound for its value.

Our simplification described in Section 3.3.4 causes a big reduction in the number of calls to the LP solver.

Our running times are still larger than those of QSopt_ex, but we believe they can be significantly improved as our implementation is still preliminary; one argument supporting our claim is the very low time we use in the rational arithmetic.

With our approach, using the exact norm computation, we can prove feasibility and give a safe bound on the objective function value for 89 of the 95 problems. Of the remaining six instances, four fail because the norm condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold – either for the last iteration delta-LP, or for the second LP needed for computing the safe bound.

For the version where we use the estimate of the norm of the inverse matrix, we are able to give a bound on the objective value for 76 of the 95 problems. Of the remaining 19 instances, 18 fail because the norm condition does not hold.

In Figure 3.2 we show the distribution of the relative error (in percent) of our safe bound on the objective value compared to the real objective value as computed by Koch [56], for the instances where we were able to provide such a safe bound. Note that for the exact norm case 88% of those instances have a relative error less than 1%. For the case using the norm estimate, 75% of the solved instances have a relative error to the exact solution of less than 1%.

As expected, the version using the norm estimate is faster than the one using the exact norm computation. Considering only those 76 instances for which both versions were able to compute an objective value bound, the geometric mean of the running times is 0.53 seconds for the former version, and 0.77 seconds for the latter.

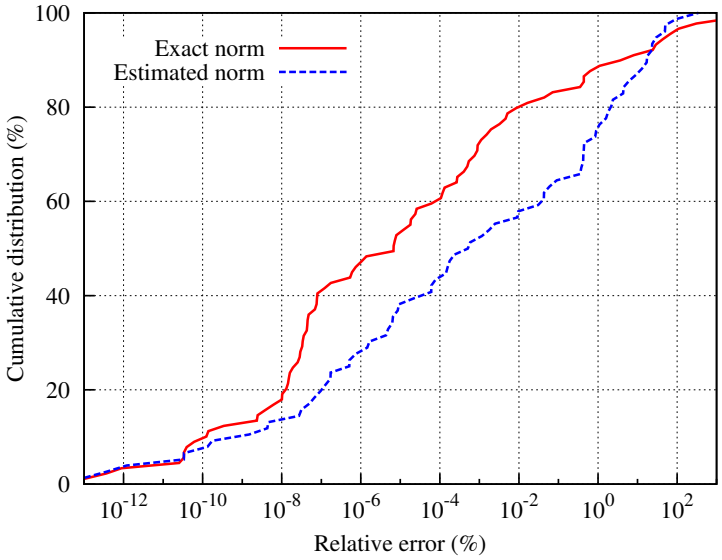


Figure 3.2: Relative error distribution
for the safe bound on the objective value for the Netlib benchmarks

The Mittelmann benchmark set

The second set of benchmarks are the linear programs collected by Hans D. Mittelmann at Arizona State University.[2] It contains 80 instances, out of which we kept only those with a size smaller than 2 MB. Thus the subset on which we tested our approach contains 32 problems, most of them representing fixed-charge transportation problems.

Our results are shown in Table A.2. We were able to solve all the 32 instances in the subset we considered, both with the version that computes the exact norm of the inverse of the basis matrix, and with that using the estimate on the norm.

As it can be noticed in Figure 3.3, for the exact computation case, the relative error of objective value bound returned relative to the objective value computed by SoPlex is less than 0.002 percent. For the version using an estimate on the norm of the inverse matrix, the objective value bound relative error to SoPlex value is less than 0.006 percent.

We can then conclude that for this benchmark set, the quality of the results is very good for both versions. The marginally better accuracy of the exact version comes at the cost of the running time: the geometric mean of the running times for the exact case is 4.31 seconds, while for the estimated norm case, the geometric mean is 2.65 seconds.

[2] Available at `http://plato.asu.edu/ftp/lptestset/`.

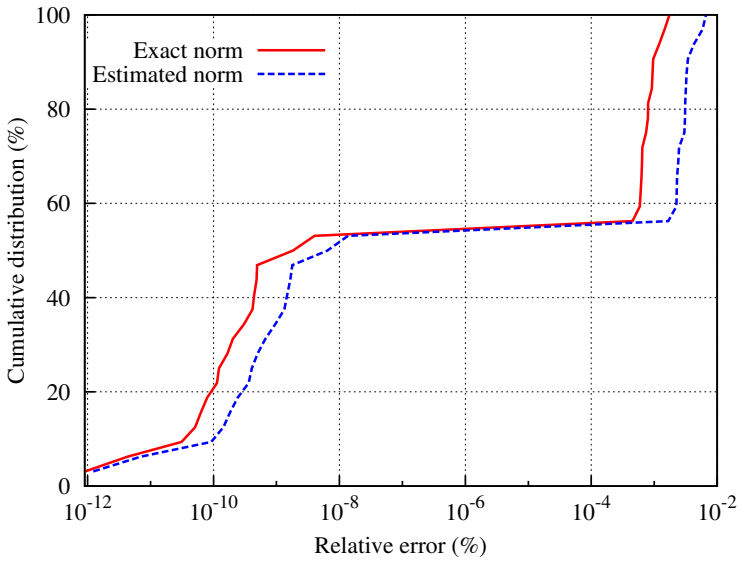


Figure 3.3: Relative error distribution
for the safe bound on the objective value for the Mittelmann benchmarks

The Sztaki benchmark set

The third set of benchmarks on which we tested our approach are the linear programs collected by Csaba Mészáros at MTA SZTAKI, the Computer and Automation Research Institute of the Hungarian Academy of Sciences.[3]

The available set contains in total 405 instances, out of which we kept only those with a size smaller than 2 MB, thus leaving us with 269 problems. After dismissing those instances that caused syntax errors because they used constructions not allowed in the standard MPS format, the working benchmark set contains 263 problems.

In Table A.3 we summarize our results on this benchmark set. Using the version that computes the exact norm of the inverse basis matrix, we were able to solve 255 instances of this test set, corresponding to a success rate of 97%. In six of the remaining eight instances where we fail this is due to the norm condition that does not hold. The computed bounds on the objective value are of good quality, as depicted in Figure 3.4. For 170 instances (69% of those where we could provide such a bound) their relative error to the SoPlex objective value is less than 1%.

For the case of the version that uses an estimate for the norm of the basis inverse, we could solve 172 of the benchmark instances (65%). As expected, the most failures in this setting are caused by not being able to satisfy the condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$; this happens for 85 of the 91 instances where we fail, or 93%.

[3] Available at `http://www.sztaki.hu/~meszaros/public_ftp/lptestset/`.

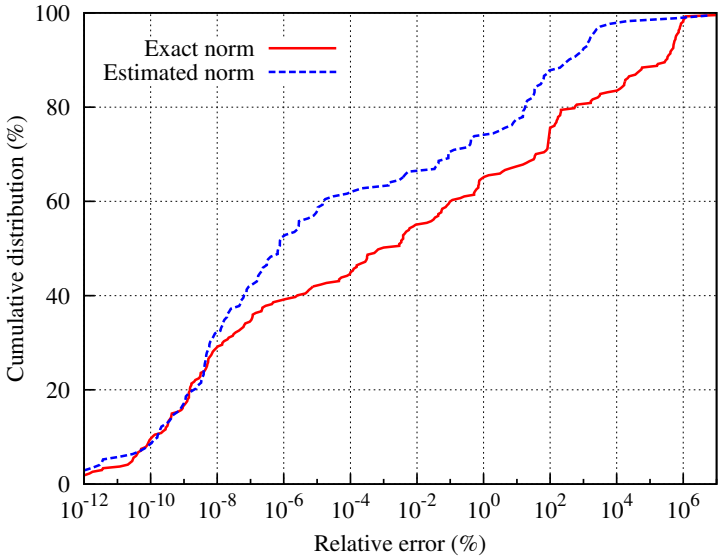


Figure 3.4: Relative error distribution
for the safe bound on the objective value for the Sztaki benchmarks

The quality of the computed bounds on the objective value is slightly better than in the exact case: in 77% of the cases where we could produce such a bound (133 out of 172 instances), the relative error to the SoPlex objective value is less than 1%.

Concerning the speed of the two approaches, the estimated norm approach is as expected significantly faster. Considering only the 172 instances for which both versions were able to compute an objective value bound, the geometric mean of the running times for the exact norm version is 2.57 seconds, while for the estimated norm version the mean is 1.58 seconds.

Summary

We performed experiments on three benchmark sets containing a total of 490 linear programs. Both our versions had good results; using the exact norm computation we could solve 96% of all instances, while using the estimate of the norm resulted in 72% of the instances being solved.

The computed bounds on the objective value typically deviated only slightly from the optimal floating-point solutions. Since the main purpose of our implementation was to prove that our approach is useful, we do believe that by carefully refining it the running times can be significantly improved; the large instances would especially benefit from this.

3.8 Conclusion

We presented an algorithm that certifies the feasibility of a linear program and computes a safe bound on its objective function value while using rational arithmetic as little as possible. For most of the instances of the benchmark set Netlib, our algorithm certifies the feasibility by solving with floating-point arithmetic only a few linear programs of the same size as the given ones; the time spent in the rational arithmetic is typically very small compared to the time needed for solving the linear programs. The objective value bound computed by our algorithms is in most of the cases very close to the known exact objective values.

For the other two benchmark sets on which we tested our approach, we could certify the feasibility for most of the instances. The obtained bounds on the objective value have a small relative error to the objective value given by an LP solver, for most of the cases less than 1%. For the cases where it could provide a bound, the method using an estimate for the norm of the inverse of the basis matrix runs significantly faster than the one using the exact computation of the norm.

For linear programs that are numerically more complicated, it could be beneficial to improve our approach to find a safe vector of intervals for the solution of a system of linear equations. There are several methods proposed in the literature and we have to check which one gives the best compromise between accuracy and running time.

Chapter 4

Integration of an LP Solver into Interval Constraint Propagation

In this chapter we report on the practical employment of a method for certifying the feasibility or infeasibility of a linear program to enhance an existing approach, namely to improve the performance of a solver for satisfiability modulo theories problems.

We describe the integration of an LP solver into iSAT, a Satisfiability Modulo Theories solver that can solve Boolean combinations of linear and nonlinear constraints. The solver iSAT is developed within the AVACS project[1] involving the universities of Oldenburg, Freiburg and Saarbrücken and is a tight integration of the well-known DPLL algorithm and interval constraint propagation that allows it to reason about linear and nonlinear constraints. Since interval arithmetic is known to be less efficient on solving linear programs, we will demonstrate how the integration of an LP solver can improve the overall solving performance of iSAT.

4.1 Introduction

We are considering the satisfiability modulo theory (SMT) problem, which can be shortly described as follows. We are given a set of variables $\{x_1, \dots x_m\}$ and a set $C = \{c_1, \dots, c_n\}$ of constraints belonging to some class of constraints over these variables, e.g., linear inequalities. Furthermore, we are given a Boolean structure over C, i.e., a Boolean formula φ whose variables are the elements of C. The task is to decide the feasibility of φ, i.e., a Boolean assignment $b : C \mapsto \{\text{true}, \text{false}\}$ for the constraints, such that φ is satisfied for b and such that there exists $x \in \mathbb{R}^m$ satisfying the constraints $\{c_i \mid b(c_i) = \text{true}\}$ and $\{\neg c_i \mid b(c_i) = \text{false}\}$. Hence we

[1] Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems", `www.avacs.org`.

assume that the class of constraints is closed under negation.

SMT is an extension of the classical satisfiability problem. Such formulas arise in many applications dealing with the verification of *Hybrid Systems* [45]. A hybrid system combines a finite automaton with a dynamical system and constitute a mathematical model for digital systems interacting with analog environments in real time.

The chapter is organized as follows. We first review in Section 4.2 the interval constraint programming approach of our underlying solver and then describe how we integrate an LP solver in Section 4.3. In Section 4.4, we show how a floating-point based LP solver can be used to solve the linear programs including strict inequalities, thereby certifying the correctness of its result. Before giving a conclusion, we report on some experiments in Section 4.6.

4.2 Background

The problem under consideration can be decided if the feasibility problem of a conjunction of constraints of the class of constraints can be solved. The most prominent algorithm is the DPLL algorithm [23] in which the Boolean values for the constraints are determined via a search with backtracking, like in ordinary SAT algorithms. Furthermore, for each partial assignment of Boolean values, the corresponding set of constraints is checked for feasibility. There are many tricks to improve the running time, like the so-called *unit propagation* – if all but one literal of a clause are assigned to false, the remaining literal has to be true. In particular for the class of linear inequalities, there are many implementations to tackle the problem; we mention here only Yices [29], MathSAT [18], and OpenSMT [19].

We are interested in more complicated classes of constraints, i.e., we allow arbitrary equalities and inequalities (strict and nonstrict) composed of addition, multiplication, exponentiation, and (co)sine. For this reason the problem under consideration becomes undecidable in general; it is well known that nonlinear arithmetic over the real numbers involving transcendental functions like *sin* is undecidable [80].

This large class of constraints allows us to model Hybrid Systems in a more precise manner since many physical systems exhibit a quadratic or exponential behavior that we want to analyze.

As we are dealing with transcendental functions the problem is in general undecidable, therefore we do not aim to solve it, but we either report the infeasibility or report a point $x \in \mathbb{R}^m$ such that all equalities and inequalities are satisfied up to some predefined accuracy in the arithmetic. In particular, strict inequalities are sometimes only satisfied non-strictly. In order to do so, we have to assume that all variables x_i and all subformulas[2] have a bounded domain. We do so by an extension of the DPLL procedure in which we additionally allow the splitting of the domain of a variable. To become efficient, we use *interval constraint*

[2]We do not give a formal definition here and refer to the paper of Fränzle et al. [32] for details.

propagation (ICP), i.e., given a constraint c_i with fixed assignment $b(c_i)$ and the current domains of the variables, we try to narrow the domains without removing any point satisfying the constraint. For example, if $x + y \leqslant 0$ is a constraint and $x \in [0, 1], y \in [-2, 2]$ are the current domains, we can narrow the domain of y to $[-2, 0]$ as there is no point in the original domain that satisfies $x + y \leqslant 0$ with $y > 0$. For more details we refer to the extensive survey by Benhamou and Granvilliers [10].

This algorithm does not require a specialized solver for a particular class of constraints as long as the narrowing of the domains can be done efficiently. Nevertheless, the work of Gao et al. [34] indicates that an additional check for feasibility for all linear constraints can improve upon the running time. They employ an LP solver to characterize the feasible region described by the linear constraints and try to hand over this information to an ICP solver (they use the predecessor of iSAT). Our work differs in that the overall search process is guided by the ICP solver and not by an LP solver as implemented in the paper of Gao et al. As ICP is known to be less efficient when solving linear systems, we perform additional LP solver calls to detect unsatisfiable search parts earlier, and thus prevent the solver from making unnecessary ICP calls. Furthermore, our results demonstrate that the combination of an ICP solver and an LP solver increases the number of problems that we can solve, especially for unsatisfiable problem instances.

We call a specialized LP solver once the domain propagation reached a fixed point. The solver first tries to decide the feasibility or infeasibility using a solution of a previous LP or its Farkas proof, respectively; if this is not successful, there are some heuristics to decide whether the LP should be solved or not. Once an LP has been shown to be infeasible, a small subset of the constraints causing the infeasibility based on the Farkas proof is given to the solver to speed up the search. This technique is called *conflict learning*.

The typical application of our algorithm is for model checking, in which the infeasibility of the formula means that the system is safe. If we report infeasible, this answer must be certain and not corrupted due to rounding errors. Hence the domain narrowing is done in a conservative way, i.e., we always adapt the rounding mode such that the domain overapproximates the real domain. When using an LP solver, we have to make sure that we do not declare a feasible LP to be infeasible or report an infeasible subsystem that is in fact feasible. In previous approaches, this was guaranteed using an LP solver based on rational arithmetic. In this chapter, we show how a state-of-the-art floating-point based LP solver can be used.

We compare an approach that goes along the lines proposed by Dhiflaoui et al. [27] to an approach that goes along the lines proposed by Neumaier and Shcherbina [69]. For a discussion of those approaches we refer to Section 3.2. In Chapter 3 we described our method that extends the latter approach. In this case, we did not need our extension since all linear systems are bounded, but rather we produced a separate implementation on top of the foundation provided by the

basic classes described in Section 3.6.

Notice that in this context, some linear systems are infeasible due to the strictness of some bounds; as a result, an arbitrary small change of the bounds can make an infeasible system feasible, hence rounding errors are critical.

4.3 Integration of an LP Solver into iSAT

In this section we present our approach that combines iSAT, a DPLL based interval constraint solver, and an LP solver. In order to do this we first provide a short introduction to iSAT; for a more detailed account we refer to the paper of Fränzle et al. [32].

4.3.1 Introducing iSAT

In the following let φ be a Boolean combination of linear and nonlinear constraint formula. The front-end of iSAT computes normalized constraints and the *Conjunctive Normal Form* (CNF). After that we end up with a formula having the following syntax:

$$\begin{array}{rcl} \mathit{formula} & ::= & \{\mathit{clause} \wedge\}^* \mathit{clause} \\ \mathit{clause} & ::= & (\{\mathit{atom} \vee\}^* \mathit{atom}) \\ \mathit{atom} & ::= & \mathit{simple\_bound} \mid \mathit{arithmetic\_predicate} \\ \mathit{simple\_bound} & ::= & \mathit{variable} \sim \mathit{rational\_const} \\ \mathit{arithmetic\_predicate} & ::= & \mathit{variable} \sim \mathit{uop\ variable} \mid \\ & & \mathit{variable} \sim \mathit{variable\ bop\ variable} \\ & & \mathit{variable} \sim \mathit{variable\ bop\ rational\_const} \end{array}$$

In the above syntax, *uop* and *bop* are unary and binary operation symbols respectively, including $+, -, \times, \sin(\cdot)$, etc., *rational_ const* ranges over the rational constants, and $\sim \in \{<, \leqslant, =, \geqslant, >\}$. To illustrate this phase, consider the following formula:

$$(x \geqslant 0) \wedge (x \leqslant 10) \wedge ((\sin(1/3x) + \sqrt{x} \geqslant y) \implies (y \geqslant 1/4x + 3)). \tag{4.1}$$

First we eliminate the Boolean operator $\Rightarrow$ by applying a Tseitin transformation [81]. To this end, the implication will be replaced by a new auxiliary Boolean variable, b. The remaining formula is then normalized by introducing additional *real* variables r_1, r_2, and r_3 and the following constraints $r_1 = 1/3x$, $r_2 = \sin(r_1)$, and $r_3 = \sqrt{x}$. Sometimes we call the additional introduced variables (r_1, r_2, r_3)

auxiliary variables. Finally, the normalized CNF problem looks as follows:

$$\begin{aligned}&(x \geqslant 0) \wedge (x \leqslant 10) \wedge (\bar{b} \vee r_2 + r_3 < y \vee y \geqslant 3 + r_4) \wedge \\ &(r_2 + r_3 \geqslant y \vee b) \wedge (y \geqslant 3 + r_4 \vee b) \wedge \\ &(r_1 = 1/3x) \wedge (r_2 = \sin(r_1)) \wedge (r_3 = \sqrt{x}) \wedge (r_4 = 1/4x).\end{aligned} \tag{4.2}$$

All clauses have now the syntax described above and can be transferred to the solver. Before describing the solving process in detail, we informally define the underlying semantics. A constraint formula φ is satisfied by a valuation of its variables if all its clauses are satisfied, that is, if at least one atom is satisfied in any clause. An atom is satisfied with respect to the standard interpretation of the arithmetic operators and the ordering relations over the reals. A constraint formula φ is *satisfiable* if there exists a satisfying valuation, referred to as a *solution* of φ. Otherwise, φ is *unsatisfiable*. We remark that by definition of satisfiability, a formula φ including or implying the empty clause, denoted by $\bot$, cannot be satisfied at all, i.e., if $\bot \in \varphi$ or $\varphi \Rightarrow \bot$ then φ is unsatisfiable.

Instead of real-valued valuations of variables, iSAT manipulates interval ranges. By using the function $\rho : \mathit{Var} \to \mathbb{I}_{\mathbb{R}}$, where *Var* is a set of variables and $\mathbb{I}_{\mathbb{R}}$ is the set of interval ranges of $\mathbb{R}$, we define a range for each variable. Note that we also support discrete variable domains (integer and Boolean). To this end, it suffices to clip the interval of integer variables accordingly, such that $[-3.4, 6.0)$ becomes $[-3, 5]$, for example. The Boolean domain is represented by $\mathbb{B} = [0, 1] \subset \mathbb{Z}$.

If both ρ' and ρ are interval valuations, then ρ' is called a *refinement* of ρ if $\rho'(v) \subseteq \rho(v)$ for each variable $v \in \mathit{Var}$. The lower and upper endpoints of an interval $\rho(x)$ for a variable x can be encoded as simple bounds. We denote the lower and upper interval end of the interval $\rho(x)$ by $\mathit{lower}(\rho(x))$ and $\mathit{upper}(\rho(x))$, respectively. For example, for the interval $\rho(x) = (-4, 9]$ we have $\mathit{lower}(\rho(x)) = (x > -4)$ and $\mathit{upper}(\rho(x)) = (x \leqslant 9)$.

Let x and y be variables, ρ be an interval valuation, and $\circ$ be a binary operation. Then $\rho(x \circ y)$ denotes the *interval hull* of $\rho(x) \hat{\circ} \rho(y)$ (i.e., the smallest enclosing interval which is representable by machine arithmetic), where the operator $\hat{\circ}$ corresponds to $\circ$ but is canonically lifted to sets. This is done analogously for unary operators.

In order to compute the interval hull $\rho(x \circ y)$ we use ICP. This allows us to narrow intervals of the variables, i.e., given a formula $x \circ y \sim z$ and interval ranges for x, y, and z, we look for intervals $\rho(x)$, $\rho(y)$, and $\rho(z)$ that are as small as possible without deleting any possible solution. By doing so iSAT can prune away definitive non-solutions and thus reduce the search space.

Figure 4.1 illustrates the ICP for the constraint $y = x^2$ and the interval ranges $x \in [3, 7]$ and $y \in [-2, 25]$. The left picture illustrates how ICP can derive a new lower bound for variable y. To do this we compute $\mathit{lower}(\rho(y)) = 3^2$, in other words the current interval of y is updated to $y \in [9, 25]$. In an analogous

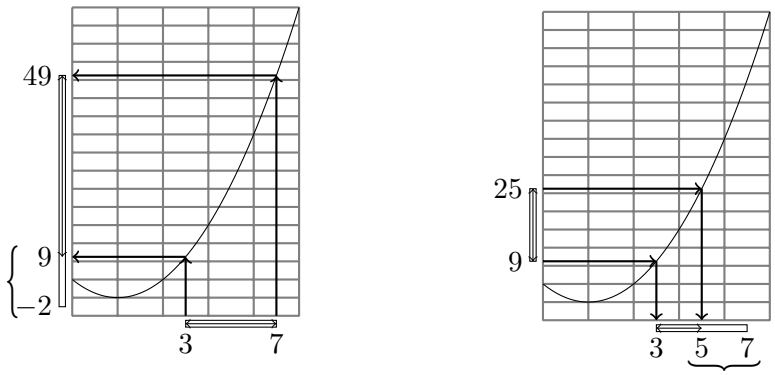


Figure 4.1: Interval constraint propagation

way the upper bound of x can be updated (right picture of Figure 4.1).

We say that an atom a is *inconsistent* under an interval valuation ρ, referred to as $\rho \sharp a$, if no values in the intervals $\rho(x)$ of the variables x in a satisfy the atom a, i.e.,

$$\begin{array}{lcll} \neg\exists v \in \rho(x) & : & v \sim c & \text{if } a = (x \sim c), \\ \neg\exists v \in \rho(x), \neg\exists v' \in \rho(\circ y) & : & v \sim v' & \text{if } a = (x \sim \circ y), \\ \neg\exists v \in \rho(x), \neg\exists v' \in \rho(y \circ z) & : & v \sim v' & \text{if } a = (x \sim y \circ z) \end{array}$$

where $\sim \in \{<, \leqslant, =, \geqslant, >\}$. Otherwise a is *consistent* under ρ. For our purpose we do not need the definition of interval satisfaction. It is sufficient to talk about atoms which are still consistent. We remark that proving the satisfiability of an iSAT formula is not trivial. For more details we refer to Section 4.5 of the paper of Fränzle et al. [32].

In Algorithm 4.1 we give the pseudocode of iSAT. Before the main iSAT routine starts, we assume that all the unit clause information contained in the original formula has already been propagated, which can sometimes allow us to derive tighter bounds. Once this is ensured, Algorithm 4.2 begins by making a decision and splitting the interval range of a variable, e.g., it splits a variable's range in half (line 3). This decision will be propagated in line 4. If a conflict is detected (e.g., a clause evaluates to *false* during propagation) it will be analyzed in line 5. The conflict analysis routine uses the implication graph of the solver to compute the reasons for the conflict. In this way a conflict clause is learned, allowing iSAT to prune off unsatisfiable parts of the search space. iSAT terminates in either line 5 or 13 with either *unsat*, *sat*, or *unknown*. If iSAT has managed it to split every problem variable up to a so-called *minimum splitting width* (msw) and no conflict is detected, the main DPLL loop is terminated and a satisfiability check for every problem clause is triggered in line 13.

```
1 Data: CNF F
2 Result: sat, unsat or unknown
  /* Main DPLL loop. DecideVar returns false once the msw for all
     */
  /* variables is reached, and no further decisions are possible.
     */
3 while decideVar() do
    /* Propagates current decision and unit constraints.        */
4   if propagateICP() = Conflict then
      /* Function tries to resolve the conflict by backtracking.
         */
      /* If conflict is unresolvable, problem is unsatisfiable.
         */
5     if analyseBacktrack() = Unresolvable then return unsat
6   end
7 end
  /* Final test to see if all the constraints are satisfied.     */
8 if allClausesSat() then return sat; else return unknown
```

Algorithm 4.1: DPLL + ICP

4.3.2 Integration of an LP Solver

The integration of an LP solver affects the *normalization* and the *solving* part of iSAT. In the normalization part iSAT detects every linear constraint that is contained in the input formula, as we explain in the following example.

In Equation (4.1) of the previous subsection there are three linear constraints ($x \geqslant 0$, $x \leqslant 10$, and $y \geqslant 1/4x + 3$). Every linear constraint that does not have the syntax of a *simple_ bound* will be given to the LP solver. In this case the only linear constraint that is not a simple bound is $y \geqslant 1/4x+3$; here the normalization routine would transform the linear constraint into $-1/4x + y - s = 0$. In other words, we introduce for every linear constraint a so-called slack variable s; in this way we produce an initially feasible LP tableau that can be transmitted to the LP solver, hence the normalization part produces the input for the iSAT solver and for the LP solver. The input for iSAT looks like:

$$\begin{aligned}
&(x \geqslant 0) \wedge (x \leqslant 10) \wedge (\overline{b} \vee r_2 + r_3 < y \vee y \geqslant r_4 + 3) \wedge \\
&(r_2 + r_3 \geqslant y \vee b) \wedge (s \geqslant 3 \vee b) \wedge \\
&(r_1 = 1/3x) \wedge (r_2 = \sin(r_1)) \wedge (r_3 = \sqrt{x}) \wedge \\
&(r_4 = 1/4x) \wedge (s = y - r_4),
\end{aligned} \tag{4.3}$$

```
1  Data: CNF F
2  Result: sat, unsat or unknown
   /* Main DPLL loop. DecideVar returns 0 once the msw for       */
   /* all variables is reached, and no further decisions         */
   /* are possible.                                              */
3  while decideVar() do
   |  /* Propagates current decision and unit constraints.       */
4  |  if propagateICP() = Conflict then
   |  |  /* Function tries to resolve the conflict by backtracking.
   |  |     */
   |  |  /* If conflict is unresolvable, problem is unsatisfiable.
   |  |     */
5  |  |  if analyseBacktrack() = Unresolvable then return unsat
6  |  end
   |  /* ICP did not find a conflict. Try to find a conflict     */
   |  /* among the linear constraints using an LP solver         */
7  |  else if checkLPFeasibility() = Infeasible then
8  |  |  insertCert() if analyseBacktrack() = Unresolvable then
9  |  |  |  return unsat
10 |  |  end
11 |  end
12 end
   /* Final test: Are all constraints satisfied?                 */
13 if allClausesSat() then return sat; else return unknown
```

Algorithm 4.2: DPLL + ICP + LP

and the input for the LP solver is:

$$-1/4x + y - s = 0. \tag{4.4}$$

The solver iSAT is able to solve the formula given in Equation (4.3) without the help of the LP solver. But as interval arithmetic is known to be less efficient when solving linear programs, our aim is to improve the effectiveness of the solver by integrating an LP solver.

To do this we modify the iSAT procedure to Algorithm 4.2 by adding additional feasibility checks for the linear constraints under the current interval valuation (line 7). These checks are performed after no conflict has been detected by the propagation phase (line 4). If the LP solver reports infeasible, a clause containing the negations of those column bounds that are responsible for the infeasibility is inserted into iSAT's learned clause database (line 8), thus preventing

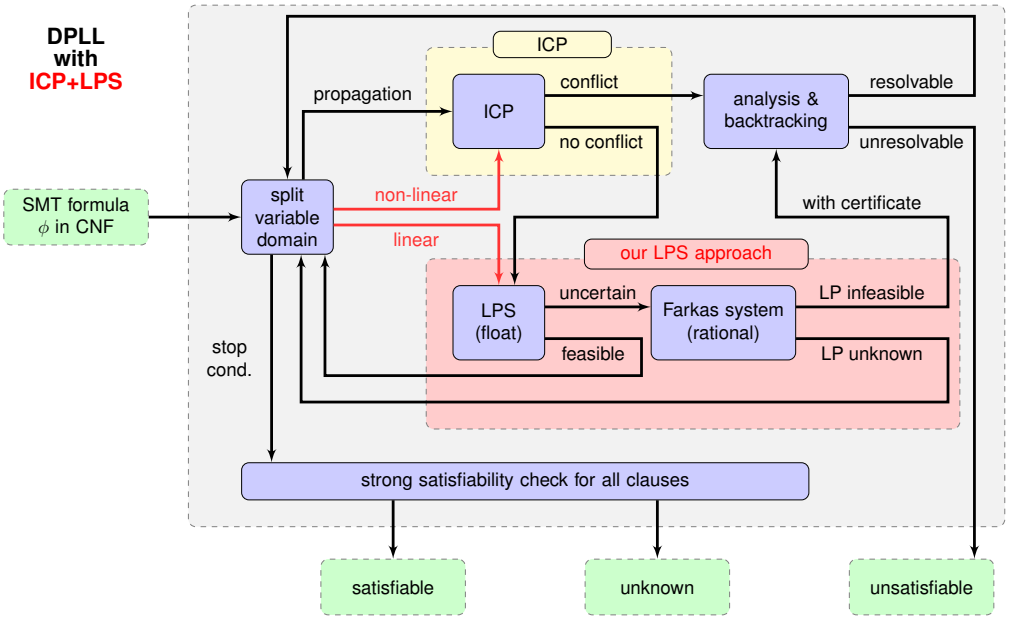


Figure 4.2: Interval constraint propagation with integrated LP solver used with the DPLL framework

iSAT from entering this particular part of the search space again. To keep the number of literals small, we compute these bounds from the Farkas' Lemma as explained in the next section.

We further implemented other options: adding the linear constraints only to the LP solver and nonlinear constraints to iSAT, or adding the linear constraints to both solvers.

4.4 Solving the Linear Programs and Computation of Small Infeasible Subsets

We start our description assuming real arithmetic of the computer and describe necessary changes due to the floating-point arithmetic afterwards. Besson [13] proposes an approach similar to ours, in that he computes infeasible subsystems with floating point arithmetic and certifies a correct result by solving a system of linear equations with rational arithmetic. His approach to compute infeasible subsystems for a system of linear inequalities including strict ones seems to be more complicated than ours.

Given a system of linear inequalities with some strict bounds as outlined in the previous section, deciding the feasibility can be achieved simply by maximizing the minimal distance to one of the strict inequalities, i.e., by solving the

following linear program:

$$\begin{array}{lrcl} \max & \delta & & \\ \text{s.t.} & Ax & = & 0 \\ & x + u^s\delta & \leqslant & u \\ & x - \ell^s\delta & \geqslant & \ell \\ & \delta & \geqslant & 0, \end{array}$$

where we define $\ell_i^s = 1$ if the lower bound ℓ_i on x_i is strict and $\ell_i^s = 0$ otherwise, $u_i^s = 1$ if the upper bound u_i on x_i is strict and $u_i^s = 0$ otherwise.

If the linear program is infeasible or has an objective function value of 0, the system of linear inequalities is infeasible, otherwise the system is feasible. A small but infeasible set of inequalities can be obtained then either from the Farkas certificate or from the dual solution. As the linear equations are globally valid, we do not include them in the certificate, but rather only a set of bounds.

More precisely, if the LP is infeasible, it has no solution with $\delta = 0$. Basically this means that the system has no solution, even if all strict bounds would be nonstrict. Therefore we can drop δ and we know that the simplified linear system $\{Ax = 0,\ \ell \leqslant x \leqslant u\}$ is infeasible. Hence its Farkas system (a linear system which is feasible if and only if the given system is infeasible) is feasible and reads as follows:

$$\begin{array}{rcl} p^T A + q^T - r^T & = & 0 \\ q^T u - r^T \ell & = & -1 \\ q, r & \geqslant & 0 \end{array} \tag{4.5}$$

Notice that if we drop all variables with value 0, the linear system (4.5) is still feasible and hence the original linear program without the corresponding constraints is still infeasible. Therefore the certificate consists of all lower bounds ℓ_i for which the corresponding Farkas system variables r_i are strictly positive and all upper bounds u_i for which the corresponding system variables q_i are strictly positive. The Farkas certificate for the (nonsimplified) linear program can be obtained from all LP solvers after the original program has been solved and can be directly used for the simplified program.

If the LP is feasible with objective function value 0, its dual has a solution with objective function value 0, i.e., the following system of linear equations is feasible:

$$\begin{array}{rcl} q^T u - r^T \ell & = & 0 \\ p^T A + q^T - r^T & = & 0 \\ q^T u^s + r^T \ell^s & = & 1 \\ q, r & \geqslant & 0 \end{array} \tag{4.6}$$

Furthermore, if the system is feasible, the LP has objective function value 0 or is infeasible. Again the certificate consists of all lower bounds ℓ_i for which $r_i > 0$ and all upper bounds u_i for which $q_i > 0$ and can be obtained from the LP solution.

Assume now that the LP is solved with a state-of-the-art floating-point solver.

We discuss three alternative methods to certify the infeasibility and the computed Farkas certificate; we implemented the latter two of them. In case any of these methods fails, it returns that the feasibility status of the LP is unknown.

The first method was proposed by Dhiflaoui et al. [27]. They take the LP basis and try to verify its correctness using rational arithmetic. Basically, the basis gives a subset of the variables such that the system (4.5), respectively (4.6), has a solution with all nonbasic variables having value 0, and the system of equations reduced to the basic variables has a unique solution which is nonnegative. Given the basis, we can solve a system of linear equations and check that the solution is nonnegative, in order to certify that the system (4.5), respectively (4.6), has a solution. This is done using rational arithmetic. The experiments of Dhiflaoui et al. show that for some instances, the time to solve these systems of linear equations is much higher than the time to solve the LP. This observation has been confirmed by a more efficient implementation by Koch [56].

In the second method, we solve a smaller system of equations in order to become more efficient. More precisely, we select the variables with floating-point value larger than some $\varepsilon > 0$ and solve the corresponding subsystem of equations. Notice that these variables are a subset of the basic variables; this subset is strict if the basis is degenerate, which is often the case in practical applications. In this situation, we get a linear system that is overdetermined. In Section 4.6, we show that these systems are often significantly smaller and can be solved very efficiently.

The third method was proposed by Neumaier and Shcherbina [69] and can be applied as all variables have finite bounds. We discuss the case of the infeasible LP first. Given the floating-point solution (p, q, r), they compute $p^T A$ using interval arithmetic. From this, they compute intervals for q and r such that the system $p^T A + q^T - r^T = 0$ definitely has a solution by setting

$$q_i = \big[\max\{0, -\mathit{upper}((p^T A)_i)\},\ \max\{(0, -\mathit{lower}((p^T A)_i)\}\big],$$

and

$$r_i = \big[\max\{0, \mathit{lower}((p^T A)_i)\},\ \max\{0, \mathit{upper}((p^T A)_i)\}\big].$$

Then we certify, again using interval arithmetic, that $q^T u - r^T \ell < 0$. Notice that it suffices that the value is strictly negative, as a solution of the linear system can then be obtained by scaling. The certificate consists of all upper bounds u_i such that the interval q_i contains a nonzero and all lower bounds ℓ_i such that the interval r_i contains a nonzero. Notice that in contrast to the two methods above, it is possible that the certificate contains both the lower and upper bound of a variable.

If the LP has objective function value 0, we have to certify this with interval arithmetic. This is only possible if p can be represented exactly by floating-point numbers and if during computation of $p^T A$ the intervals remain point intervals. Still, the certificates are often trivial enough such that this is achieved. Notice that we cannot expect to obtain a method that can handle non-point intervals,

since a slight change of the bounds can change the objective function value and hence the feasibility status.

4.5 Implementation

We have implemented our approach into iSAT using C++ and SoPlex as LP solver. Our choice to use SoPlex as LP solver and not Gurobi or CPLEX is motivated by our intention to certify the feasibility of LPs using the approach we described in Chapter 3 in a future version, where we will need access to the LU decomposition of the matrix. SoPlex gives us the possibility to access its LU decomposition, although for this it may be necessary to modify its source code, as we had to do for our certifying approach (see Section 3.6.4). The proprietary nature of Gurobi and CPLEX does not allow us access to their internals.

To conveniently communicate with the linear solver, we wrote an interface to SoPlex that allows basic manipulation of LPs. For storing and manipulating linear programs, as well as for utilities and sparse vectors and matrices, we used the main classes from our approach for certifying feasibility and computing safe bounds on the objective value as described in Section 3.6.

Given the special form

$$\begin{array}{rrcl} \max & 0 & & \\ \text{s.t.} & Ax & = & 0 \\ \ell \leqslant & x & \leqslant & u \\ \ell^s < & x & < & u^s \end{array}$$

of the linear programs occurring in iSAT, the interface allows to create and reset (i.e., clear) the linear program, give names to variables, add constraints, and set bounds on variables. There can be both strict and nonstrict bounds. The latter involve special care; since normal linear program solvers can deal only with nonstrict bounds or constraints, the strict bounds are modeled as nonstrict constraints containing a supplementary positive variable δ (see Section 4.4). This solution requires extra bookkeeping: when a variable with a nonstrict bound is modified in order to have a strict bound, the extra δ-constraint has to be added to the LP, and when the strict bound becomes nonstrict, the corresponding δ-constraint has to be removed. At the same time, as long as there exists at least one strict bound, the LP objective is max δ; when all bounds become nonstrict, the objective is simply reduced to max 0.

Furthermore, the interface provides a `solve()` method that effectively calls the SoPlex solver and, based on its outcome, takes the necessary steps described in Section 4.4 and returns *feasible*, *infeasible*, or *unknown*.

In case the answer is *infeasible*, a set of bounds is returned, itself a subset of the infeasible set that proves that the LP is indeed infeasible; this is done by calling the method `get_explanation_bounds()`. In the following we explain shortly the rationale behind the infeasible set.

For LPs that it declares infeasible, SoPlex provides access to a so-called Farkas proof, basically a coefficient for every constraint such that the corresponding linear combination generates a contradiction, proving the (floating-point) infeasibility. In this way one can get an infeasible set composed of constraints and bounds that is a witness for the infeasibility. The constraints are those with a nonzero Farkas vector entry. Let y be the Farkas vector returned by SoPlex. The bounds in the infeasible set are the upper bounds u_i for those variables x_i for which $(y^T A)_i > 0$, and the lower bounds ℓ_i for those variables x_i for which $(y^T A)_i < 0$. Notice that only a set of bounds will never yield infeasibility (unless they are trivially inconsistent, i.e., the lower bound is greater than the upper bound). Normally, neither the constraints alone (then the root LP is already infeasible which is uninteresting) nor the bounds alone yield infeasibility. An infeasible set can never contain both bounds: consider that there are two feasible solutions x^u, x^ℓ when removing upper and lower bound respectively. Then a suitable convex combination of x^u and x^ℓ will be feasible with respect to both bounds (and all others).

Unfortunately, the current implementation of the SoPlex solver is not able to return the Farkas vector when there is a simplifier and it is turned on. For this reason, we had to use the solver without a simplifier. This can have a negative impact on the performance that we have to accept, since our approach depends on having a Farkas vector at hand.

The SoPlex API provides a method `isConsistent()` that checks that the internals of a linear program are correct and is supposed to be used when the linear program is programmatically manipulated, e.g., constraints and bounds are added, removed, or changed. Since these are the main operations executed during the iSAT procedure, the method ought to be called often. Nevertheless, the overhead of the method is significant. We were able to achieve a considerable improvement in the running time by 'aggregating' the calls, basically ensuring that the check takes place at least before every call to SoPlex's `solve()`.

Given the 'incremental' nature of the linear programs occurring during the execution of iSAT, i.e., in every iteration the LP to be solved has only a limited number of modifications compared to the previous one, and that we use the simplex algorithm, it makes sense not to restart from scratch every time (by calling SoPlex's method `reLoad()`) but make use of the previous solution and optimal base, except of course when the LP is completely cleared. We can thus reduce the number of iterations the algorithm needs for finding a new optimum and the running time. However in a few cases SoPlex throws exceptions and we do try to recover by starting the solver anew; in most of the cases this solves the issue.

We also implemented a Gauss–Jordan solver for sparse linear systems using our templated classes `SparseVector<T>` and `SparseMatrix<T>`; for the rational arithmetic approach, the system coefficients and free terms are `mpq_class` rationals. The solver uses full Markowitz pivoting to exploit the sparsity of the system matrix and minimize the fill-in generated during the execution (see Section 2.1.2)

and returns the number of solutions, i.e., none, one, or infinitely many, an information that is further used to decide on the answer to be returned to the iSAT system.

To become efficient, we further try to detect implications among the linear constraints before starting the search, as presented by Pigorsch and Scholl [73]. In this way the number of LP solver calls is decreased. Other methods, like the elimination of variables by exploiting equalities appearing as top-level conjuncts of the formula or the simplification of the Boolean part of the input formula, both suggested by Bruttomesso et al. [19], have not yet been implemented; they would probably speed up our algorithm considerably.

4.6 Experiments

We tested the performance of our implementation on the QF-LRA benchmarks (quantifier-free linear real arithmetic) from SMT-LIB [77], which contains SMT problems having only linear constraints. In order to be able to use them within our ICP solver, we artificially make all variables bounded between -10^5 and 10^5. These instances can be solved by specialized solvers, which have much better running times. Nevertheless, we have chosen these benchmarks because they come from a standard benchmark library.

The iSAT solver is tuned to prove the unsatisfiability of a formula and returns *unknown* whenever it finds a candidate solution. Hence, for feasible solutions it often returns *unknown* and thus we cannot compare running times for those instances. Therefore, we restrict to unsatisfiable instances.

All experiments are performed on a 2.3 GHz Quad-Core AMD Opteron machine with 4 GB of physical memory running Linux with kernel version 2.6.32. We compiled our program with GNU `g++` 4.4.3 with the optimization flag `-O2` turned on.

4.6.1 Comparison of Different LP Solving Techniques

We compared four different methods to solve the linear systems, all of them giving certified correct results, i.e., results that are not corrupted by possible errors in the floating-point arithmetic. In addition to the three methods described in Section 4.4, we use an LP solver based on rational arithmetic. For this purpose, we use the LP solver Yices [29], one of the fastest SMT solvers available, which is specialized to handle strict inequalities.

Due to internals of iSAT, it terminates sometimes with *unknown* before the time limit. These instances cannot be considered as correctly solved.

The results of our experiments can be found in Appendix A. In Table A.4, we show for each method the number of nodes in the search tree and the total running time for a typical subset of the instances, the *clock_synchro* instances. With our approach for certifying the correctness of the floating-point LP solver, 197 out of the 300 infeasible instances are solved, whereas only 193 are solved with the

second best approach, i.e., using a rational LP solver. The small difference in the number of solved problem instances is explained by the fact that both approaches are very similar. However, the main advantage of our approach is that it needs approximately half of the solving time, especially when focusing on instances with larger running times. Using the approach of Neumaier and Shcherbina to certify the infeasibility of the LPs, we can solve 174 of the instances.

We implemented the option to additionally handle linear constraints with the ICP approach and show for this setting the same numbers. This option turns out to produce worse results, e.g., it solves only 146 instead of 193 of the instances. It is not surprising that the combination of ICP and LP solving is less efficient: in this case ICP is redundant and less accurate compared to an LP solver. Furthermore, deriving that two linear constraints cannot hold at the same time by applying ICP steps can lead to a huge amount of arithmetic computations, causing the slowdown of the overall search process.

4.6.2 Detailed Evaluation of our Certifying Approach

In Table A.5 we show the details of our approach for certifying the correctness of the floating-point based LP solver. For a subclass of the infeasible instances, we show the average size of the basis, the average size of the infeasible subsystem, the running time to solve the linear program, the running time to certify its correctness, the number of times our certifying is not successful, the number of nodes in the search tree, and the total running time.

If our approach is successful, it returns exactly the same infeasible subsystem as the approach of Dhiflaoui et al., but solves a much smaller system of equations with rational arithmetic. More precisely, we solve a system with as many variables as the size of the infeasible subsystem, whereas they solve a system in the size of the basis. We show the differences in the size of the solved systems in Table A.5 and do not report on the approach of Dhiflaoui et al. in our comparison on different LP certifying methods.

4.6.3 Summary

The integration of a floating-point based LP solver into ICP can greatly reduce the running time needed to solve the benchmark instances.

Moreover, the size of the infeasible subsets and hence the size of the system of linear equations we have to solve with rational arithmetic is very small on average, as one can see in column *ifs* (compare to column *basis*). This is helpful in two aspects. On one hand, it gives small conflict clauses that can reduce the search space. On the other hand, the running time for the Gaussian elimination is not the bottleneck of the approach – as opposed to solving with rational arithmetic the equation system given by the LP basis as, for example, in the work of Dhiflaoui et al.

We believe that we can become even more efficient, if we implement some further tricks to speed up the solution process; among those we mention stor-

ing LP solutions, carefully deciding which linear programs should be solved, or implementing further preprocessing techniques.

Furthermore, we plan to improve our approach by using the information from the LP solver to propagate bounds and to add linear relaxations of the nonlinear constraints to the LP solver.

4.7 Conclusion

We presented a tight integration of an LP solver into interval constraint propagation and experimented with our initial implementation. Already in this preliminary scenario offering multiple possibilities for further optimization, the benefit of this integration is obvious.

The experiments we performed revealed that our approach needs only approximately half of the solving time, especially when focusing on instances with larger running times. We were able to show that an LP solver can help to improve solvers based on interval constraint propagation.

As future work, we want to improve our approach, for instance by finding new methods to propagate bounds based on the linear constraints, as well as appropriate linear relaxations of nonlinear constraints.

Appendix A

Experimental Results

In this appendix we list the detailed numerical results of our experiments.

We presented in Chapter 3 our approach that certifies the feasibility of a linear program and computes a safe bound on its objective value. We described our implementation in Section 3.6, and the technical settings and the used benchmark sets in Section 3.7. In Tables A.1, A.2, and A.3 we give the outcome of our experiments. For a discussion of our results we refer to Section 3.7.

In Chapter 4 we presented the integration of an LP solver in interval constraint propagation. The implementation details can be found in Section 4.5; an explanation of the hardware environment and the benchmark sets is given in Section 4.6. In Tables A.4 and A.5 we list the results of the experiments we performed for our approach. These results are discussed in Section 4.6.

Table A.1: Experimental results on the Netlib benchmark set.

A subscript X means using $\|A^{-1}\|$ computed exactly, E using the upper bound for $\|A^{-1}\|$. *SPx objv* is the value of objective function for the original LP as computed by SoPlex, $Time_S$ is the time needed by SoPlex, *It* shows the number of iterations in our approach (this is the same for X and E as the iteration part does not involve norm computation), It_0 shows iterations without simplification of LP, *Safe objv* is the computed safe bound on the objective function value, *Time* is the total time in seconds needed to certify feasibility and compute the safe bound on the objective function value, $Time_{Qex}$ is the time needed by QSopt_ex. An explicit value for *It* simply means that we were able to stop with a positive δ, subject to a later (non-)successful certification.

Name	Rows	Cols	Nzos	SPx objv	Time_S	It	It_0	Safe objv_X	Time_X	Safe objv_E	Time_E	Time_{Qex}
25FV47	821	1571	10400	5.5018e+03	0.59	1	1	—[b]	—	—[b]	—	1.21
80BAU3B	2262	9799	21002	9.8722e+05	0.71	4	2	5.5987e+05	80.46	1.4724e+06	50.89	0.61
ADLITTLE	56	97	383	2.2549e+05	0.00	1	2	2.2549e+05	0.03	2.2549e+05	0.03	0.00
AFIRO	27	32	83	–4.6475e+02	0.00	1	1	–4.6475e+02	0.02	–4.6475e+02	0.01	0.00
AGG	488	163	2410	–3.5992e+07	0.01	1	71	–3.5992e+07	0.19	–3.5992e+07	0.15	0.02
AGG2	516	302	4284	–2.0239e+07	0.01	1	3	–2.0239e+07	0.39	–2.0239e+07	0.26	0.03
AGG3	516	302	4300	1.0312e+07	0.01	1	3	1.0312e+07	0.40	1.0312e+07	0.27	0.03
BANDM	305	472	2494	–1.5863e+02	0.05	1	22	–1.5863e+02	0.34	–7.8901e+01	0.25	0.10
BEACONFD	173	262	3375	3.3592e+04	0.00	1	79	3.3592e+04	0.14	3.3592e+04	0.12	0.01
BLEND	74	83	491	–3.0812e+01	0.00	1	1	–3.0812e+01	0.04	–3.0812e+01	0.03	0.00
BNL1	643	1175	5121	1.9776e+03	0.09	1	81	1.9777e+03	1.82	1.9859e+03	1.19	0.17
BNL2	2324	3489	13999	1.8112e+03	0.22	1	112	1.8112e+03	14.29	2.1194e+03	8.15	0.66
BOEING1	351	384	3485	–3.3521e+02	0.04	2	2	–3.3521e+02	0.62	–3.3518e+02	0.43	0.04
BOEING2	166	143	1196	–3.1502e+02	0.00	5	2	–3.1502e+02	0.14	–3.1488e+02	0.12	0.01
BORE3D	233	315	1429	1.3731e+03	0.00	2	140	1.3791e+03	0.12	1.3791e+03	0.10	0.03
BRANDY	220	249	2148	1.5185e+03	0.01	1	44	1.5185e+03	0.17	1.5242e+03	0.13	0.06
CAPRI	271	353	1767	2.6900e+03	0.01	1	1	2.6900e+03	0.27	2.6900e+03	0.19	0.02
CYCLE	1903	2857	20720	–5.2264e+00	0.27	1	333	–5.0358e+00	9.81	1.3008e+01	5.44	0.31
CZPROB	929	3523	10669	2.1852e+06	0.19	1	348	2.1852e+06	5.89	2.1852e+06	3.91	0.15

[a] condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold, [b] numerical difficulties.

(*continued on next page*)

Table A.1: Experimental results on the Netlib benchmark set. (*continued*)

Name	Rows	Cols	Nzos	SPx objv	Time_S	It	It_0	Safe objv_X	Time_X	Safe objv_E	Time_E	Time_Qex
D2Q06C	2171	5167	32417	1.2278e+05	2.34	4	2	5.0314e+05	37.01	—[a]	—	93.86
D6CUBE	415	6184	37704	3.1549e+02	0.13	1	1	3.1748e+02	34.33	—[a]	—	0.99
DEGEN2	444	534	3978	–1.4352e+03	0.08	55	7	–1.4194e+03	4.23	–1.4194e+03	4.07	0.08
DEGEN3	1503	1818	24646	–9.8729e+02	0.70	126	8	–9.8381e+02	88.11	–9.8381e+02	85.66	0.87
DFL001	6071	12230	35632	1.1266e+07	20.86	6	13	5.3500e+17	388.78	—[a]	—	114.25
E226	223	282	2578	–1.8752e+01	0.03	1	31	–1.8749e+01	0.23	–1.6337e+01	0.18	0.05
ETAMACRO	400	688	2409	–7.5572e+02	0.03	1	48	–7.5572e+02	0.58	–7.5571e+02	0.39	0.40
FFFFF800	524	854	6227	5.5568e+05	0.07	1	20	5.5568e+05	1.15	5.5618e+05	0.76	0.05
FINNIS	497	614	2310	1.7279e+05	0.02	1	29	1.7280e+05	0.52	1.7434e+05	0.35	0.03
FIT1D	24	1026	13404	–9.1464e+03	0.02	1	1	–9.1464e+03	3.20	–8.5947e+03	1.97	0.08
FIT1P	627	1677	9868	9.1464e+03	0.07	1	1	9.1464e+03	3.15	9.1464e+03	1.89	0.09
FIT2D	25	10500	129018	–6.8464e+04	1.04	1	1	–6.8434e+04	296.60	—[a]	—	5.76
FIT2P	3000	13525	50284	6.8464e+04	2.33	26	—	—[b]	—	—[a]	—	6.13
FORPLAN	161	421	4563	–6.6422e+02	0.01	2	—	–6.6422e+02	0.29	–6.3540e+02	0.25	0.03
GANGES	1309	1681	6912	–1.0959e+05	0.04	1	101	–1.0959e+05	3.15	–1.0959e+05	1.86	0.10
GFRD-PNC	616	1092	2377	6.9022e+06	0.03	1	27	6.9022e+06	1.36	6.9022e+06	0.90	0.05
GREENBEA	2392	5405	30877	–7.2555e+07	6.25	1	2	–7.2504e+07	28.74	—[a]	—	4.01
GREENBEB	2392	5405	30877	–4.3023e+06	2.88	1	1180	–3.9577e+06	25.75	—[a]	—	5.00
GROW15	300	645	5620	–1.0687e+08	0.12	1	1	–1.0687e+08	1.30	–1.0495e+08	0.79	4.12
GROW22	440	946	8252	–1.6083e+08	0.22	1	1	–1.6083e+08	2.64	–1.5699e+08	1.53	1.79
GROW7	140	301	2612	–4.7788e+07	0.01	1	1	–4.7788e+07	0.33	–4.7787e+07	0.22	0.31
ISRAEL	174	142	2269	–8.9664e+05	0.01	1	1	–8.9664e+05	0.12	–8.1473e+05	0.09	0.01
KB2	43	41	286	–1.7499e+03	0.00	1	1	–1.7499e+03	0.02	–1.7499e+03	0.02	0.00
LOTFI	153	308	1078	–2.5265e+01	0.00	1	1	–2.5265e+01	0.13	–2.5265e+01	0.10	0.00
MAROS	846	1443	9614	–5.8064e+04	0.41	14	525	–1.9277e+04	5.68	—[a]	—	0.28
MAROS-R7	3136	9408	144848	1.4972e+06	6.29	1	—	—[a]	—	—[a]	—	1009.90
MODSZK1	687	1620	3168	3.2062e+02	0.03	1	—	3.2062e+02	2.02	3.2081e+02	1.37	0.17
NESM	662	2923	13288	1.4076e+07	0.34	1	1	1.4076e+07	10.62	1.4704e+07	5.99	0.45
PEROLD	625	1376	6018	–9.3808e+03	0.43	1	40	–6.6659e+03	3.05	—[a]	—	6.82

[a] condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold, [b] numerical difficulties.

(*continued on next page*)

Table A.1: Experimental results on the Netlib benchmark set. (*continued*)

Name	Rows	Cols	Nzos	SPx objv	Time_S	It	It_0	Safe objv_X	Time_X	Safe objv_E	Time_E	Time_{Qex}
PILOT	1441	3652	43167	–5.5749e+02	4.21	1	79	–4.2247e+02	40.49	—[a]	—	527.30
PILOT4	410	1000	5141	–2.5811e+03	0.18	1	7	–6.4165e+04	1.82	—[a]	—	2.64
PILOT87	2030	4883	73152	3.0171e+02	10.60	1	—	—[a]	—	—[a]	—	6383.73
PILOTJA	940	1988	14698	–6.1131e+03	0.87	—	—	—[a]	—	—[a]	—	34.16
PILOTNOV	975	2172	13057	–4.4973e+03	0.44	1	58	—[a]	—	—[a]	—	8.72
PILOTWE	722	2789	9126	–2.7201e+06	0.50	1	32	2.4139e+05	7.44	—[a]	—	4.31
RECIPE	91	180	663	–2.6662e+02	0.00	1	18	–2.6662e+02	0.05	–2.6662e+02	0.05	0.00
SC105	105	103	280	–5.2202e+01	0.00	1	2	–5.2202e+01	0.05	–5.2202e+01	0.03	0.01
SC205	205	203	551	–5.2202e+01	0.00	1	3	–5.2202e+01	0.11	–5.2202e+01	0.06	0.00
SC50A	50	48	130	–6.4575e+01	0.00	1	2	–6.4575e+01	0.01	–6.4575e+01	0.01	0.00
SC50B	50	48	118	–7.0000e+01	0.00	1	3	–7.0000e+01	0.01	–7.0000e+01	0.01	0.00
SCAGR25	471	500	1554	–1.4753e+07	0.06	1	1	–1.4753e+07	0.44	–1.4753e+07	0.30	0.04
SCAGR7	129	140	420	–2.3314e+06	0.01	1	1	–2.3314e+06	0.06	–2.3314e+06	0.04	0.00
SCFXM1	330	457	2589	1.8417e+04	0.02	1	15	1.8417e+04	0.37	1.8573e+04	0.25	0.01
SCFXM2	660	914	5183	3.6660e+04	0.05	1	29	3.6660e+04	1.25	3.6816e+04	0.78	0.08
SCFXM3	990	1371	7777	5.4901e+04	0.09	1	43	5.4902e+04	2.59	—[a]	—	0.12
SCORPION	388	358	1426	1.8781e+03	0.00	13	67	1.8863e+03	0.35	1.8865e+03	0.30	0.02
SCRS8	490	1169	3182	9.0430e+02	0.02	1	42	9.0430e+02	1.10	9.0469e+02	0.73	0.03
SCSD1	77	760	2388	8.6667e+00	0.00	1	1	8.6667e+00	0.41	8.6667e+00	0.29	0.03
SCSD6	147	1350	4316	5.0500e+01	0.01	1	1	5.0500e+01	1.15	5.0500e+01	0.77	0.07
SCSD8	397	2750	8584	9.0500e+02	0.05	1	1	9.0500e+02	4.31	9.0509e+02	2.62	0.20
SCTAP1	300	480	1692	1.4123e+03	0.01	1	1	1.4123e+03	0.30	1.4123e+03	0.21	0.01
SCTAP2	1090	1880	6714	1.7248e+03	0.02	1	1	1.7248e+03	3.24	1.7248e+03	1.90	0.03
SCTAP3	1480	2480	8874	1.4240e+03	0.02	1	1	1.4240e+03	5.58	1.4240e+03	3.22	0.02
SEBA	515	1028	4352	1.5712e+04	0.01	1	133	1.5712e+04	1.33	1.5712e+04	0.82	0.01
SHARE1B	117	225	1151	–7.6589e+04	0.01	1	1	–7.6589e+04	0.11	–5.4185e+04	0.09	0.01
SHARE2B	96	79	694	–4.1573e+02	0.01	1	1	–4.1573e+02	0.05	–3.2002e+02	0.03	0.00
SHELL	536	1775	3556	1.2088e+09	0.02	1	1	1.2088e+09	1.66	1.2088e+09	1.09	0.04
SHIP04L	402	2118	6332	1.7933e+06	0.01	1	202	1.7933e+06	2.22	1.7933e+06	1.43	0.04

[a] condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold, [b] numerical difficulties.

(*continued on next page*)

Table A.1: Experimental results on the Netlib benchmark set. (*continued*)

Name	Rows	Cols	Nzos	SPx objv	Time_S	It	It_0	Safe objv_X	Time_X	Safe objv_E	Time_E	Time_Qex
SHIP04S	402	1458	4352	1.7987e+06	0.00	1	90	1.7987e+06	1.14	1.7987e+06	0.76	0.02
SHIP08L	778	4283	12802	1.9091e+06	0.01	3	1126	1.9091e+06	7.53	1.9091e+06	5.28	0.08
SHIP08S	778	2387	7114	1.9201e+06	0.02	3	552	1.9201e+06	2.37	1.9201e+06	1.76	0.05
SHIP12L	1151	5427	16170	1.4702e+06	0.03	1	1067	1.4702e+06	9.88	1.4702e+06	5.91	0.14
SHIP12S	1151	2763	8178	1.4892e+06	0.02	1	361	1.4892e+06	2.67	1.4892e+06	1.69	0.08
SIERRA	1227	2036	7302	1.5394e+07	0.00	1	1	1.5394e+07	8.66	1.5394e+07	4.71	0.04
STAIR	356	467	3856	–2.5127e+02	0.06	1	1	–2.5127e+02	0.60	–2.5119e+02	0.36	6.19
STANDATA	359	1075	3031	1.2577e+03	0.01	1	55	1.2577e+03	1.04	1.2577e+03	0.72	0.00
STANDGUB	361	1184	3139	1.2577e+03	0.00	1	—	1.2577e+03	1.04	1.2577e+03	0.73	0.01
STANDMPS	467	1075	3679	1.4060e+03	0.01	1	55	1.4060e+03	1.27	1.4060e+03	0.87	0.01
STOCFOR1	117	111	447	–4.1132e+04	0.00	1	1	–4.1132e+04	0.04	–4.1131e+04	0.03	0.00
STOCFOR2	2157	2031	8343	–3.9024e+04	0.24	1	1	–3.9024e+04	6.12	–2.9794e+04	3.51	0.29
STOCFOR3	16675	15695	64875	–3.9977e+04	10.46	1	—	–3.9976e+04	425.76	–3.3002e+04	217.95	11.21
TRUSS	1000	8806	27836	4.5882e+05	5.55	1	1	4.5882e+05	41.09	4.6882e+05	23.64	1.96
TUFF	333	587	4520	2.9215e–01	0.01	1	30	2.9216e–01	0.65	5.6884e–01	0.46	0.03
VTPBASE	198	203	908	1.2983e+05	0.00	1	172	1.2983e+05	0.07	1.2983e+05	0.06	0.00
WOOD1P	244	2594	70215	1.4429e+00	0.14	1	793	1.4430e+00	8.81	—[a]	—	1.31
WOODW	1098	8405	37474	1.3045e+00	0.58	1	—	1.3045e+00	22.68	1.3253e+00	15.38	0.29

[a] condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold, [b] numerical difficulties.

Table A.2: Experimental results on the Mittelmann/Arizona State University benchmark set.

A subscript X means using $\|A^{-1}\|$ computed exactly, E using the upper bound for $\|A^{-1}\|$. *SPx objv* is the value of objective function for the original LP as computed by SoPlex, $Time_{\mathrm{S}}$ is the time needed by SoPlex, *It* shows the number of iterations in our approach (this is the same for X and E as the iteration part does not involve norm computation), *Safe objv* is the computed safe bound on the objective function value, *Time* is the total time in seconds needed to certify feasibility and compute the safe bound on the objective function value, $Time_{\mathrm{Qex}}$ is the time needed by QSopt_ex. An explicit value for *It* simply means that we were able to stop with a positive δ, subject to a later (non-)successful certification.

Name	Rows	Cols	Nzos	SPx objv	$\mathrm{Time_S}$	It	Safe $\mathrm{objv_X}$	$\mathrm{Time_X}$	Safe $\mathrm{objv_E}$	$\mathrm{Time_E}$	$\mathrm{Time_{Qex}}$
BAL8X12	116	192	384	4.511881e+02	0.01	1	4.511881e+02	0.09	4.511881e+02	0.06	0.00
BK4X3	19	24	48	3.216667e+02	0.00	1	3.216667e+02	0.01	3.216667e+02	0.01	0.00
GR4X6	34	48	96	1.855500e+02	0.00	1	1.855500e+02	0.02	1.855500e+02	0.01	0.00
N3700	5150	10000	20000	9.723057e+05	1.64	1	9.723120e+05	166.56	9.723299e+05	89.05	1.27
N3701	5150	10000	20000	9.617640e+05	1.57	1	9.617697e+05	163.67	9.617857e+05	90.21	1.35
N3702	5150	10000	20000	9.611842e+05	1.61	1	9.611914e+05	170.26	9.612139e+05	90.51	1.17
N3703	5150	10000	20000	9.342122e+05	1.49	1	9.342197e+05	159.68	9.342420e+05	90.38	1.44
N3704	5150	10000	20000	9.698002e+05	1.55	1	9.698096e+05	164.64	9.698335e+05	88.56	1.38
N3705	5150	10000	20000	9.733610e+05	1.79	1	9.733727e+05	169.71	9.734024e+05	88.17	1.32
N3706	5150	10000	20000	9.608821e+05	1.50	1	9.608989e+05	169.20	9.609460e+05	87.60	1.26
N3707	5150	10000	20000	9.351369e+05	1.54	1	9.351429e+05	164.95	9.351593e+05	91.31	1.34
N3708	5150	10000	20000	9.675228e+05	1.73	1	9.675305e+05	165.39	9.675518e+05	91.47	1.23
N3709	5150	10000	20000	9.593142e+05	1.59	1	9.593282e+05	166.69	9.593703e+05	86.84	1.40
N370A	5150	10000	20000	9.792195e+05	1.43	1	9.792285e+05	167.27	9.792515e+05	89.64	1.37
N370B	5150	10000	20000	9.883081e+05	1.59	1	9.883126e+05	162.16	9.883246e+05	91.20	1.35
N370C	5150	10000	20000	9.654309e+05	1.38	1	9.654370e+05	167.48	9.654532e+05	89.68	1.32
N370D	5150	10000	20000	9.654309e+05	1.33	1	9.654370e+05	167.00	9.654532e+05	88.85	1.31
N370E	5150	10000	20000	9.616513e+05	1.43	1	9.616603e+05	161.88	9.616811e+05	90.69	1.33
RAN10X10A	120	200	400	1.252742e+03	0.00	1	1.252742e+03	0.09	1.252742e+03	0.07	0.00
RAN10X10B	120	200	400	2.613469e+03	0.01	1	2.613469e+03	0.09	2.613469e+03	0.07	0.00

(*continued on next page*)

Table A.2: Experimental results on the Mittelmann/ASU benchmark set. (*continued*)

Name	Rows	Cols	Nzos	SPx objv	Time_S	It	Safe objv_X	Time_X	Safe objv_E	Time_E	Time_Qex
RAN10X10C	120	200	400	1.120309e+04	0.00	1	1.120309e+04	0.10	1.120309e+04	0.07	0.00
RAN10X12	142	240	480	2.426225e+03	0.01	1	2.426225e+03	0.12	2.426225e+03	0.09	0.00
RAN10X26	296	520	1040	3.857023e+03	0.00	1	3.857023e+03	0.44	3.857023e+03	0.29	0.00
RAN12X12	168	288	576	1.826550e+03	0.00	1	1.826550e+03	0.17	1.826550e+03	0.12	0.00
RAN12X21	285	504	1008	3.157377e+03	0.01	1	3.157377e+03	0.42	3.157377e+03	0.28	0.01
RAN13X13	195	338	676	2.691439e+03	0.01	1	2.691439e+03	0.21	2.691439e+03	0.15	0.00
RAN14X18	284	504	1008	3.016944e+03	0.01	1	3.016944e+03	0.42	3.016944e+03	0.28	0.01
RAN16X16	288	512	1024	3.116430e+03	0.01	1	3.116430e+03	0.44	3.116430e+03	0.29	0.02
RAN17X17	323	578	1156	1.215246e+03	0.00	1	1.215246e+03	0.53	1.215246e+03	0.34	0.02
RAN4X64	324	512	1024	9.637933e+03	0.01	1	9.637933e+03	0.46	9.637933e+03	0.30	0.01
RAN6X43	307	516	1032	6.244707e+03	0.01	1	6.244707e+03	0.45	6.244707e+03	0.29	0.00
RAN8X32	296	512	1024	4.937585e+03	0.01	1	4.937585e+03	0.44	4.937585e+03	0.28	0.01

Table A.3: Experimental results on the Mészáros/SZTAKI benchmark set.

A subscript X means using $\|A^{-1}\|$ computed exactly, E using the upper bound for $\|A^{-1}\|$. *SPx objv* is the value of objective function for the original LP as computed by SoPlex, $Time_S$ is the time needed by SoPlex, *It* shows the number of iterations in our approach (this is the same for X and E as the iteration part does not involve norm computation), *Safe objv* is the computed safe bound on the objective function value, *Time* is the total time in seconds needed to certify feasibility and compute the safe bound on the objective function value, $Time_{Qex}$ is the time needed by QSopt_ex. An explicit value for *It* simply means that we were able to stop with a positive δ, subject to a later (non-)successful certification.

Name	Rows	Cols	Nzos	SPx objv	Time_S	It	Safe objv_X	Time_X	Safe objv_E	Time_E	Time_{Qex}
AA4	426	7195	52121	2.587761e+04	0.24	92	4.292163e+04	608.92	4.292163e+04	570.46	1.18
AA6	646	7292	51728	2.697719e+04	0.29	199	3.613250e+04	833.88	3.615093e+04	774.15	2.23
AIRCRAFT	3754	7517	20267	1.567042e+03	0.24	1	1.567042e+03	49.76	1.567042e+03	25.18	0.62
CEP1	1521	3248	6712	3.551601e+05	0.03	1	3.551601e+05	7.08	3.551601e+05	4.34	0.14
COMPLEX	1023	1408	46463	–9.966667e+01	1.48	—	—[a]	—	—[a]	—	8.63
CR42	905	1513	6614	2.801850e+01	0.03	1	2.801850e+01	2.48	2.801850e+01	1.45	0.16
CREW1	135	6469	46950	2.055556e+01	0.07	1	2.055556e+01	121.03	—[a]	—	0.20
DE080285	936	1488	4662	1.392475e+01	0.04	1	4.586495e+02	1.73	—[a]	—	1.93
DELF000	3128	5464	12606	3.076374e+00	0.23	1	3.078052e+00	16.68	—[a]	—	2.79
DELF001	3098	5462	13214	2.358609e+03	0.21	1	2.358611e+03	18.31	2.525189e+03	9.20	3.27
DELF002	3135	5460	13287	2.832814e+00	0.20	1	2.838535e+00	18.68	4.100342e+01	9.41	3.07
DELF003	3065	5460	13269	9.115545e+03	0.25	1	9.120413e+03	18.02	—[a]	—	3.15
DELF004	3142	5464	13546	1.586937e+02	0.31	1	1.645238e+02	16.65	—[a]	—	3.49
DELF005	3103	5464	13494	2.287304e+03	0.34	1	4.543354e+04	16.83	—[a]	—	3.25
DELF006	3147	5469	13604	2.284311e+02	0.30	1	5.498678e+02	17.34	—[a]	—	6.17
DELF007	3137	5471	13758	3.799558e+02	0.29	1	4.851428e+04	17.26	—[a]	—	4.40
DELF008	3148	5472	13821	2.412355e+02	0.45	1	1.522895e+03	17.48	—[a]	—	4.78
DELF009	3135	5472	13750	4.841248e+02	0.33	1	2.320441e+05	17.70	—[a]	—	6.13
DELF010	3147	5472	13802	2.291067e+02	0.37	1	5.817167e+02	17.60	—[a]	—	4.95

[a] condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold, [b] numerical difficulties.

(*continued on next page*)

Table A.3: Experimental results on the Mészáros/SZTAKI benchmark set. (*continued*)

Name	Rows	Cols	Nzos	SPx objv	Time_S	It	Safe objv_X	Time_X	Safe objv_E	Time_E	Time_Qex
DELF011	3134	5471	13777	4.734942e+02	0.36	1	1.109515e+03	17.12	—[a]	—	3.48
DELF012	3151	5471	13793	1.786801e+02	0.39	1	5.571493e+02	19.11	—[a]	—	3.92
DELF013	3116	5472	13809	2.616677e+03	0.29	1	2.643700e+03	17.18	—[a]	—	4.87
DELF014	3170	5472	13866	1.534986e+02	0.39	1	8.282318e+03	17.88	—[a]	—	3.70
DELF015	3161	5471	13793	7.370368e+02	0.45	1	1.312764e+04	17.51	—[a]	—	3.89
DELF017	3176	5471	13732	4.615184e+02	0.32	1	1.383766e+03	19.15	—[a]	—	3.74
DELF018	3196	5471	13774	1.421339e+02	0.51	1	3.828498e+02	19.51	—[a]	—	2.81
DELF019	3185	5471	13762	2.342129e+03	0.34	1	2.354309e+03	19.16	4.832079e+05	9.80	2.53
DELF020	3213	5472	14070	3.516813e+02	0.49	1	6.156548e+02	19.87	—[a]	—	3.61
DELF021	3208	5471	14068	3.947044e+02	0.34	1	6.382237e+05	17.90	—[a]	—	2.73
DELF022	3214	5472	14060	3.649929e+02	0.38	1	9.950891e+02	17.83	—[a]	—	2.87
DELF023	3214	5472	14098	3.541171e+02	0.38	1	1.477083e+05	17.97	—[a]	—	2.83
DELF024	3207	5466	14456	3.514168e+02	0.58	1	4.823323e+02	17.85	—[a]	—	4.93
DELF025	3197	5464	14447	3.502968e+02	0.32	1	1.107624e+04	17.41	—[a]	—	3.38
DELF026	3190	5462	14220	3.516134e+02	0.36	1	1.944794e+05	17.52	—[a]	—	3.16
DELF027	3187	5457	14200	3.194556e+02	0.40	1	8.224572e+02	17.41	—[a]	—	2.66
DELF028	3177	5452	14402	2.972457e+02	0.27	1	8.302804e+03	17.54	—[a]	—	2.80
DELF029	3179	5454	14402	2.752257e+02	0.32	1	4.015515e+04	17.46	—[a]	—	2.79
DELF030	3199	5469	14262	2.538402e+02	0.32	1	1.682195e+03	17.79	—[a]	—	3.32
DELF031	3176	5455	14205	2.342985e+02	0.30	1	4.157724e+04	17.33	—[a]	—	2.78
DELF032	3196	5467	14251	2.160392e+02	0.31	1	3.624510e+04	19.55	—[a]	—	3.15
DELF033	3173	5456	14205	2.002717e+02	0.33	1	3.636626e+04	19.11	—[a]	—	2.90
DELF034	3175	5455	14208	1.894375e+02	0.33	1	4.059815e+04	17.15	—[a]	—	3.22
DELF035	3193	5468	14284	1.769488e+02	0.32	1	4.172587e+04	17.82	—[a]	—	3.34
DELF036	3170	5459	14202	1.644593e+02	0.28	1	2.794988e+04	17.35	—[a]	—	3.00
DETER0	1923	5468	11173	–2.045920e+00	0.10	1	–2.045920e+00	63.25	–2.045920e+00	29.58	0.41
DETER1	5527	15737	32187	–2.557564e+00	0.31	1	–2.557564e+00	743.80	–2.557564e+00	445.54	2.10
DETER2	6095	17313	35731	–1.829869e+00	0.33	1	–1.829868e+00	961.71	–1.829868e+00	558.00	2.92
DETER4	3235	9133	19231	–1.428035e+00	0.10	1	–1.428035e+00	231.81	–1.428035e+00	108.05	1.62

[a] condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold, [b] numerical difficulties.

(*continued on next page*)

Table A.3: Experimental results on the Mészáros/SZTAKI benchmark set. (*continued*)

Name	Rows	Cols	Nzos	SPx objv	Time_S	It	Safe objv_X	Time_X	Safe objv_E	Time_E	Time_{Qex}
DETER5	5103	14529	29715	–2.261224e+00	0.32	1	–2.261224e+00	659.49	–2.261224e+00	364.42	1.93
DETER6	4255	12113	24771	–2.307987e+00	0.20	1	–2.307987e+00	464.48	–2.307987e+00	244.72	1.58
DETER7	6375	18153	37131	–2.228568e+00	0.39	1	–2.228568e+00	913.35	–2.228568e+00	542.63	3.14
DETER8	3831	10905	22299	–2.794295e+00	0.18	1	–2.794295e+00	341.04	–2.794295e+00	173.13	1.20
DISP3	2182	1856	6407	2.020795e+05	0.01	25	2.020795e+05	13.91	2.020795e+05	9.27	0.05
FARM	7	12	36	1.750000e+04	0.00	1	1.750000e+04	0.00	1.750000e+04	0.01	0.00
FXM2–16	3900	5602	31239	1.841676e+04	0.52	1	1.841680e+04	48.39	1.844039e+04	18.42	0.70
FXM2–6	1520	2172	12139	1.841707e+04	0.11	1	1.841709e+04	9.13	1.842533e+04	3.23	0.19
FXM3_6	6200	9492	54589	1.861604e+04	1.04	1	1.861610e+04	144.59	1.862296e+04	48.94	1.27
GAMS10A	114	61	297	1.000000e+00	0.00	1	1.000000e+00	0.03	1.000000e+00	0.03	0.02
GAMS30A	354	181	937	1.000000e+00	0.00	1	1.000000e+00	0.19	1.000000e+00	0.14	0.02
IIASA	669	2970	6648	2.634861e+08	0.02	1	2.634862e+08	3.51	2.634898e+08	2.28	0.07
KLEEMIN3	3	3	6	–1.000000e+04	0.00	1	–1.000000e+04	0.02	–1.000000e+04	0.00	0.00
KLEEMIN4	4	4	10	–1.000000e+06	0.00	1	–1.000000e+06	0.01	–1.000000e+06	0.00	0.00
KLEEMIN5	5	5	15	–1.000000e+08	0.00	1	–1.000000e+08	0.00	–1.000000e+08	0.01	0.00
KLEEMIN6	6	6	21	–1.000000e+10	0.00	1	–1.000000e+10	0.02	–1.000000e+10	0.00	0.00
KLEEMIN7	7	7	28	–1.000000e+12	0.00	1	–1.000000e+12	0.01	–1.000000e+12	0.00	0.00
KLEEMIN8	8	8	36	–1.000000e+14	0.00	1	–1.000000e+14	0.00	–1.000000e+14	0.00	0.00
L9	244	1401	4577	9.982357e–01	0.08	1	—[b]	—	—[a]	—	0.25
LARGE000	4239	6833	16573	7.260856e+01	0.29	1	1.324479e+02	30.87	—[a]	—	5.16
LARGE001	4162	6834	17225	3.318142e+04	1.52	1	3.365058e+04	30.11	—[a]	—	3.59
LARGE002	4249	6835	18330	2.881523e+00	0.65	1	1.042166e+03	28.86	—[a]	—	8.55
LARGE003	4200	6835	18016	2.459549e+04	0.52	1	1.968270e+05	0.85	—[a]	—	7.21
LARGE004	4250	6836	17739	1.747063e+02	0.69	1	1.963186e+06	28.43	—[a]	—	7.08
LARGE005	4237	6837	17575	4.118529e+02	0.55	1	2.076443e+06	34.38	—[a]	—	4.83
LARGE006	4249	6837	17887	2.474492e+02	0.55	1	1.512326e+06	35.59	—[a]	—	6.33
LARGE007	4236	6836	17856	5.026115e+02	0.70	1	1.658453e+06	28.68	—[a]	—	6.17
LARGE008	4248	6837	17898	2.681140e+02	0.54	1	2.241937e+06	35.91	—[a]	—	6.80
LARGE009	4237	6837	17878	5.028242e+02	0.66	1	2.075390e+06	35.26	—[a]	—	8.90

[a] condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold, [b] numerical difficulties.

(*continued on next page*)

Table A.3: Experimental results on the Mészáros/SZTAKI benchmark set. (*continued*)

Name	Rows	Cols	Nzos	SPx objv	Time_S	It	Safe objv_X	Time_X	Safe objv_E	Time_E	Time_{Qex}
LARGE010	4247	6837	17887	2.562503e+02	0.56	1	1.849972e+08	29.26	—[a]	—	6.29
LARGE011	4236	6837	17878	4.935848e+02	0.53	1	4.240448e+06	37.64	—[a]	—	6.19
LARGE012	4253	6838	17919	1.997012e+02	0.60	1	8.184743e+05	36.72	—[a]	—	6.31
LARGE013	4248	6838	17941	5.710062e+02	0.60	1	5.751937e+06	35.52	—[a]	—	5.31
LARGE014	4271	6838	17979	1.715761e+02	0.63	1	1.710034e+06	34.80	—[a]	—	5.45
LARGE015	4265	6838	17957	6.725306e+02	0.75	1	1.991157e+06	36.29	—[a]	—	5.96
LARGE016	4287	6838	18029	1.634869e+02	0.52	1	1.895979e+06	38.41	—[a]	—	6.22
LARGE017	4277	6837	17983	6.256828e+02	0.45	1	1.716226e+06	29.20	—[a]	—	4.75
LARGE018	4297	6837	17791	2.146165e+02	0.52	1	1.717140e+06	36.99	—[a]	—	4.11
LARGE019	4300	6836	17786	5.255277e+02	0.52	1	4.106102e+06	37.73	—[a]	—	3.80
LARGE020	4315	6837	18136	3.775187e+02	0.75	1	2.483043e+06	36.66	—[a]	—	4.05
LARGE021	4311	6838	18157	4.221310e+02	0.63	1	2.450336e+06	37.79	—[a]	—	5.48
LARGE022	4312	6834	18104	3.774854e+02	0.59	1	2.054623e+06	36.47	—[a]	—	4.86
LARGE023	4302	6835	18123	3.624467e+02	0.64	1	8.176672e+07	29.49	—[a]	—	4.71
LARGE024	4292	6831	18599	3.560315e+02	0.97	—	—[a]	—	—[a]	—	6.84
LARGE025	4297	6832	18743	3.554460e+02	0.95	1	1.599421e+06	30.28	—[a]	—	9.00
LARGE026	4284	6824	18631	3.563755e+02	0.61	1	1.811919e+06	29.79	—[a]	—	6.36
LARGE027	4275	6821	18562	3.337057e+02	0.80	1	1.759159e+06	35.04	—[a]	—	4.48
LARGE028	4302	6833	18886	3.113651e+02	0.88	1	1.548677e+06	30.55	—[a]	—	5.68
LARGE029	4301	6832	18952	2.874776e+02	1.06	1	5.078356e+03	30.23	—[a]	—	6.45
LARGE030	4285	6823	18843	2.651260e+02	0.82	1	1.550175e+06	36.14	—[a]	—	5.34
LARGE031	4294	6826	18867	2.450277e+02	0.70	1	8.227692e+05	40.81	—[a]	—	6.97
LARGE032	4292	6827	18850	2.261456e+02	0.81	1	1.643177e+06	29.99	—[a]	—	11.78
LARGE033	4273	6817	18791	2.098381e+02	0.60	1	1.515376e+06	40.39	—[a]	—	5.00
LARGE034	4294	6831	18855	1.987676e+02	0.87	1	3.423576e+05	29.57	—[a]	—	10.61
LARGE035	4293	6829	18881	1.856066e+02	0.79	1	6.739348e+05	29.79	—[a]	—	11.22
LARGE036	4282	6822	18840	1.710654e+02	0.67	1	9.878149e+04	30.06	—[a]	—	9.65
LPL2	3294	10755	32106	4.465000e+03	0.04	1	4.465013e+03	55.42	4.466487e+03	28.74	0.13
MODEL1	362	798	3028	0.000000e+00	0.00	1	0.000000e+00	1.08	0.000000e+00	0.69	0.07

[a] condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold, [b] numerical difficulties.

(*continued on next page*)

Table A.3: Experimental results on the Mészáros/SZTAKI benchmark set. (*continued*)

Name	Rows	Cols	Nzos	SPx objv	Time$_S$	It	Safe objv$_X$	Time$_X$	Safe objv$_E$	Time$_E$	Time$_{Qex}$
MODEL2	379	1212	7498	–7.400489e+03	0.08	1	3.897465e+04	2.26	—[a]	—	0.25
MODEL3	1609	3840	23236	1.748906e+04	1.20	2	1.762019e+04	16.95	—[a]	—	3.15
MODEL4	1337	4549	45340	1.112702e+06	1.91	1	1.379682e+06	28.45	—[a]	—	89.62
MODEL6	2096	5001	27340	1.175077e+05	1.98	1	1.175077e+05	29.32	—[a]	—	42.90
MODEL7	3358	8007	49452	4.938009e+04	4.40	1	1.536549e+05	66.10	—[a]	—	44.23
MODEL8	2896	6464	25277	0.000000e+00	0.33	1	1.305683e+01	66.88	1.306319e+01	49.70	23.97
MODEL9	2879	10257	55274	–1.411231e+05	1.96	2	—[b]	—	—[b]	—	9.68
MULTI	61	102	961	4.440468e+04	0.00	1	4.440468e+04	0.07	4.440468e+04	0.05	0.00
NEMSAFM	334	2252	2730	–6.792374e+03	0.00	1	–6.792374e+03	3.66	–6.792374e+03	1.38	0.03
NEMSCEM	651	1570	3698	8.977233e+04	0.03	1	8.977238e+04	3.23	8.977498e+04	1.97	0.06
NEMSPMM1	2372	8622	55586	–3.274158e+05	3.95	1	3.423783e+05	83.26	—[a]	—	10.26
NSIC1	451	463	2853	–9.168554e+06	0.01	1	–9.168554e+06	0.43	–9.168554e+06	0.29	0.02
NSIC2	465	463	3015	–8.203512e+06	0.02	1	–8.203486e+06	0.57	–7.851668e+06	0.32	0.02
NUG05	210	225	1050	5.000000e+01	0.01	1	5.000000e+01	0.14	5.000069e+01	0.10	0.00
NUG06	372	486	2232	8.600000e+01	0.04	1	8.600059e+01	0.57	—[a]	—	0.11
NUG07	602	931	4214	1.480000e+02	0.20	1	1.525989e+04	2.05	—[a]	—	0.50
NUG08	912	1632	7296	2.035000e+02	1.07	—	—[a]	—	—[a]	—	1.50
NUG12	3192	8856	38304	5.228944e+02	705.41	—	—[a]	—	—[a]	—	412.87
ORNA1	882	882	3108	–5.044297e+08	0.11	1	–3.291465e+08	4.74	—[a]	—	2.96
ORNA2	882	882	3108	–5.864624e+08	0.12	1	–4.976368e+08	4.89	–4.973973e+08	2.61	2.99
ORNA3	882	882	3108	–5.840362e+08	0.10	1	–5.275500e+08	4.96	–4.943411e+08	2.71	3.15
ORNA4	882	882	3108	6.721839e+08	0.14	1	6.722066e+08	4.94	6.721841e+08	2.68	4.89
ORNA7	882	882	3108	–5.837442e+08	0.11	1	–5.655058e+08	5.16	—[a]	—	3.09
ORSWQ2	80	80	264	4.847429e–01	0.00	1	4.847434e–01	0.10	4.863877e–01	0.05	0.01
P0033	15	33	98	2.520572e+03	0.00	1	2.520572e+03	0.01	2.520572e+03	0.01	0.00
P0040	23	40	110	6.179655e+04	0.00	11	6.179655e+04	0.03	6.179655e+04	0.03	0.00
P0201	133	201	1923	6.875000e+03	0.00	23	6.875000e+03	0.43	6.875000e+03	0.41	0.00
P0282	241	282	1966	1.768675e+05	0.00	1	1.768675e+05	0.31	1.768675e+05	0.21	0.01
P0291	252	291	2031	1.705129e+03	0.00	1	1.705129e+03	0.33	1.705129e+03	0.21	0.01

[a] condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold, [b] numerical difficulties.

(*continued on next page*)

Table A.3: Experimental results on the Mészáros/SZTAKI benchmark set. (*continued*)

Name	Rows	Cols	Nzos	SPx objv	Time_S	It	Safe objv_X	Time_X	Safe objv_E	Time_E	Time_Qex
P0548	176	548	1711	3.152549e+02	0.00	1	3.152641e+02	0.63	3.152720e+02	0.42	0.01
P19	284	586	5305	2.539644e+05	0.01	1	2.539644e+05	0.80	2.541855e+05	0.50	0.06
P2756	755	2756	8937	2.688750e+03	0.00	1	2.688750e+03	12.99	2.688750e+03	7.11	0.03
P6000	2095	5872	17731	–2.351871e+06	0.68	1	–2.350838e+06	69.48	–2.350838e+06	43.52	0.11
PCB1000	1565	2428	20071	5.680946e+04	0.14	18	5.685072e+04	25.34	—[a]	—	0.57
PF2177	9728	900	21706	9.000000e+01	0.87	—	—[a]	—	—[a]	—	1.80
PGP2	4034	9220	18440	4.473243e+02	0.13	1	4.473243e+02	71.95	4.473243e+02	38.37	6.54
PLDD000B	3069	3267	8980	2.743281e+00	0.15	1	2.745950e+00	10.21	3.738538e+00	5.63	1.46
PLDD001B	3069	3267	8981	3.811372e+00	0.16	1	3.844215e+00	10.39	4.274387e+01	5.85	1.61
PLDD002B	3069	3267	8982	4.065131e+00	0.16	1	4.094832e+00	10.43	1.363560e+02	5.56	1.59
PLDD003B	3069	3267	8983	4.034888e+00	0.20	1	4.064322e+00	10.51	6.500126e+01	5.76	1.52
PLDD004B	3069	3267	8984	4.182203e+00	0.15	1	4.211323e+00	13.37	7.170367e+01	5.76	1.37
PLDD005B	3069	3267	8985	4.153993e+00	0.21	1	4.182748e+00	10.46	1.996196e+01	5.77	1.29
PLDD006B	3069	3267	8986	4.182429e+00	0.15	1	4.208129e+00	11.02	6.006743e+01	5.76	2.12
PLDD007B	3069	3267	8987	4.043838e+00	0.16	1	4.054474e+00	10.55	1.198175e+01	5.81	1.53
PLDD008B	3069	3267	9047	4.172146e+00	0.16	1	4.447175e+00	11.57	2.315719e+05	6.01	2.34
PLDD009B	3069	3267	9050	4.543196e+00	0.15	1	5.427677e+00	11.13	2.552913e+05	6.10	1.45
PLDD010B	3069	3267	9053	4.827592e+00	0.16	1	4.831475e+00	10.71	7.630175e+00	5.99	2.28
PLDD011B	3069	3267	9055	4.968126e+00	0.14	1	4.993870e+00	10.80	2.924784e+04	6.03	2.30
PLDD012B	3069	3267	9057	4.613329e+00	0.17	1	4.617527e+00	11.13	6.188799e+00	5.91	2.11
PLTEXPA2–16	1726	4540	9233	–9.663308e+00	0.04	1	–9.663308e+00	12.21	–9.663308e+00	7.20	0.06
PLTEXPA2–6	686	1820	3703	–9.479354e+00	0.01	1	–9.479354e+00	2.08	–9.479354e+00	1.39	0.03
PLTEXPA3_6	4430	11612	23611	–1.396937e+01	0.39	1	–1.396935e+01	148.21	–1.396935e+01	86.23	0.35
PRIMAGAZ	1554	10836	21665	1.071903e+09	0.25	1	1.071903e+09	76.95	1.071903e+09	38.28	0.51
PROBLEM	12	46	86	0.000000e+00	0.00	1	0.000000e+00	0.02	0.000000e+00	0.01	0.00
PROGAS	1650	1425	8422	7.607732e+05	0.47	1	—[a]	—	—[a]	—	24.78
QIULP	1192	840	3432	–9.316389e+02	0.16	1	–9.316384e+02	1.91	–9.279609e+02	1.21	0.72
REFINE	29	33	124	–3.926918e+05	0.01	1	–3.926918e+05	0.01	–3.926918e+05	0.00	0.00
ROSEN1	520	1024	23274	–2.761274e+04	0.16	1	–2.761266e+04	3.45	–1.232230e+03	1.81	0.22

[a] condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold, [b] numerical difficulties.

(*continued on next page*)

Table A.3: Experimental results on the Mészáros/SZTAKI benchmark set. (*continued*)

Name	Rows	Cols	Nzos	SPx objv	Time$_S$	It	Safe objv$_X$	Time$_X$	Safe objv$_E$	Time$_E$	Time$_{Qex}$
ROSEN2	1032	2048	46504	–5.441751e+04	0.60	1	–5.441722e+04	12.30	–7.717359e+03	5.66	0.59
ROSEN7	264	512	7770	–2.032970e+04	0.03	1	–2.032970e+04	0.75	–2.032966e+04	0.47	0.05
ROSEN8	520	1024	15538	–4.212268e+04	0.13	1	–4.212268e+04	2.53	–3.566208e+04	1.35	0.14
SC205–2R–100	2213	2214	6030	–1.007049e+01	0.15	1	–1.007049e+01	7.52	–1.007049e+01	3.53	0.13
SC205–2R–16	365	366	990	–5.538771e+01	0.00	1	–5.538771e+01	0.23	–5.538771e+01	0.16	0.01
SC205–2R–200	4413	4414	12030	–1.007049e+01	0.41	1	–1.007049e+01	23.66	–1.007049e+01	13.39	0.49
SC205–2R–27	607	608	1650	–1.510574e+01	0.01	1	–1.510574e+01	0.52	–1.510574e+01	0.35	0.01
SC205–2R–32	717	718	1950	–5.538771e+01	0.02	1	–5.538771e+01	0.73	–5.538771e+01	0.48	0.00
SC205–2R–4	101	102	270	–6.042296e+01	0.00	1	–6.042296e+01	0.03	–6.042296e+01	0.03	0.00
SC205–2R–400	8813	8814	24030	–1.007049e+01	1.68	1	–1.007049e+01	132.07	–1.007049e+01	60.71	1.81
SC205–2R–50	1113	1114	3030	–3.076411e+01	0.06	1	–3.076411e+01	1.62	–3.076411e+01	1.06	0.04
SC205–2R–64	1421	1422	3870	–5.538771e+01	0.08	1	–5.538771e+01	2.57	–5.538771e+01	1.65	0.04
SC205–2R–8	189	190	510	–6.042296e+01	0.00	1	–6.042296e+01	0.08	–6.042296e+01	0.07	0.00
SCAGR7–2B–16	623	660	2058	–8.329022e+05	0.05	1	–8.329022e+05	0.60	–8.329022e+05	0.40	0.03
SCAGR7–2B–4	167	180	546	–8.329413e+05	0.00	1	–8.329413e+05	0.07	–8.329413e+05	0.05	0.01
SCAGR7–2B–64	9743	10260	32298	–8.329015e+05	2.78	1	–5.647991e+05	211.31	–5.647991e+05	115.87	2.36
SCAGR7–2C–16	623	660	2058	–8.322600e+05	0.05	1	–8.322600e+05	0.59	–8.322600e+05	0.40	0.03
SCAGR7–2C–4	167	180	546	–8.322600e+05	0.00	1	–8.322600e+05	0.07	–8.322600e+05	0.05	0.01
SCAGR7–2C–64	2447	2580	8106	–8.187196e+05	0.21	1	–8.187196e+05	7.79	–8.187196e+05	4.61	0.24
SCAGR7–2R–108	4119	4340	13542	–8.340388e+05	0.58	1	–8.340388e+05	23.75	–8.340388e+05	12.20	0.42
SCAGR7–2R–16	623	660	2058	–8.329022e+05	0.04	1	–8.329022e+05	0.59	–8.329022e+05	0.40	0.04
SCAGR7–2R–216	8223	8660	27042	–8.340388e+05	1.99	1	–8.340388e+05	126.96	–8.340388e+05	70.65	1.65
SCAGR7–2R–27	1041	1100	3444	–8.335438e+05	0.05	1	–8.335438e+05	1.49	–8.335438e+05	0.96	0.07
SCAGR7–2R–32	1231	1300	4074	–8.329021e+05	0.08	1	–8.329021e+05	2.06	–8.329021e+05	1.31	0.08
SCAGR7–2R–4	167	180	546	–8.329022e+05	0.00	1	–8.329022e+05	0.10	–8.329022e+05	0.06	0.00
SCAGR7–2R–54	2067	2180	6846	–8.338341e+05	0.14	1	–8.338341e+05	5.49	–8.338341e+05	3.33	0.18
SCAGR7–2R–64	2447	2580	8106	–8.329020e+05	0.24	1	–8.329020e+05	7.66	–8.329020e+05	4.63	0.23
SCAGR7–2R–8	319	340	1050	–8.329022e+05	0.01	1	–8.329022e+05	0.19	–8.329022e+05	0.13	0.02
SCFXM1–2B–16	2460	3714	13959	2.877564e+03	0.27	1	2.877737e+03	14.51	3.475363e+03	7.93	0.43

[a] condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold, [b] numerical difficulties.

(*continued on next page*)

Table A.3: Experimental results on the Mészáros/SZTAKI benchmark set. (*continued*)

Name	Rows	Cols	Nzos	SPx objv	Time_S	It	Safe objv_X	Time_X	Safe objv_E	Time_E	Time_Qex
SCFXM1–2B–4	684	1014	3999	2.875983e+03	0.04	1	2.875983e+03	1.26	2.876092e+03	0.78	0.09
SCFXM1–2C–4	684	1014	3999	2.875983e+03	0.04	1	2.875984e+03	1.26	2.931384e+03	0.78	0.08
SCFXM1–2R–16	2460	3714	13959	2.877564e+03	0.27	1	2.877661e+03	15.03	3.395804e+03	7.88	0.47
SCFXM1–2R–27	4088	6189	23089	2.886965e+03	0.61	1	2.902998e+03	42.17	1.168577e+04	20.94	0.80
SCFXM1–2R–32	4828	7314	27239	2.877564e+03	0.75	1	2.877568e+03	73.34	2.880080e+03	32.85	1.20
SCFXM1–2R–4	684	1014	3999	2.877564e+03	0.04	1	2.877573e+03	1.25	2.964588e+03	0.78	0.10
SCFXM1–2R–64	9564	14514	53799	2.877564e+03	2.96	1	3.005138e+03	220.06	—[a]	—	5.46
SCFXM1–2R–8	1276	1914	7319	2.877564e+03	0.10	1	2.877581e+03	4.04	2.889496e+03	2.33	0.19
SCRS8–2B–16	476	645	1633	1.121043e+02	0.00	1	1.121043e+02	0.34	1.121043e+02	0.25	0.00
SCRS8–2B–4	140	189	457	1.121043e+02	0.00	1	1.121043e+02	0.05	1.121043e+02	0.04	0.00
SCRS8–2B–64	1820	2469	6337	1.121451e+03	0.01	1	1.121451e+03	4.12	1.121451e+03	2.68	0.04
SCRS8–2C–16	476	645	1633	1.124317e+02	0.00	1	1.124317e+02	0.35	1.124317e+02	0.25	0.00
SCRS8–2C–32	924	1253	3201	1.121562e+02	0.01	1	1.121562e+02	1.13	1.121562e+02	0.77	0.03
SCRS8–2C–4	140	189	457	1.121043e+02	0.00	1	1.121043e+02	0.05	1.121043e+02	0.05	0.00
SCRS8–2C–64	1820	2469	6337	1.118812e+02	0.02	1	1.118812e+02	4.04	1.118812e+02	2.60	0.05
SCRS8–2C–8	252	341	849	1.124159e+02	0.00	1	1.124159e+02	0.12	1.124159e+02	0.10	0.01
SCRS8–2R–128	3612	4901	12609	1.177345e+03	0.07	1	1.177345e+03	15.75	1.177345e+03	9.77	0.11
SCRS8–2R–16	476	645	1633	1.231766e+02	0.00	1	1.231766e+02	0.35	1.231766e+02	0.25	0.01
SCRS8–2R–256	7196	9765	25153	1.144161e+03	0.18	1	1.144161e+03	96.40	1.144161e+03	48.88	0.32
SCRS8–2R–27	784	1063	2711	5.871664e+02	0.01	1	5.871664e+02	0.92	5.871664e+02	0.59	0.01
SCRS8–2R–32	924	1253	3201	1.231766e+02	0.01	1	1.231766e+02	1.14	1.231766e+02	0.77	0.03
SCRS8–2R–4	140	189	457	1.231766e+02	0.01	1	1.231766e+02	0.05	1.231766e+02	0.04	0.00
SCRS8–2R–64	1820	2469	6337	1.231766e+02	0.02	1	1.231766e+02	4.04	1.231766e+02	2.61	0.05
SCRS8–2R–64B	1820	2469	6337	1.346096e+03	0.01	1	1.346096e+03	4.05	1.346096e+03	2.66	0.06
SCRS8–2R–8	252	341	849	1.122971e+03	0.00	1	1.122971e+03	0.12	1.122971e+03	0.10	0.01
SCSD8–2B–16	330	2310	7170	2.750000e+01	0.00	1	2.750000e+01	2.95	2.750000e+01	1.79	0.04
SCSD8–2B–4	90	630	1890	1.525000e+01	0.00	1	1.525000e+01	0.30	1.525000e+01	0.22	0.00
SCSD8–2C–16	330	2310	7170	1.500000e+01	0.00	1	1.500000e+01	2.94	1.500000e+01	1.74	0.05
SCSD8–2C–4	90	630	1890	1.500000e+01	0.00	1	1.500000e+01	0.30	1.500000e+01	0.22	0.00

[a] condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold, [b] numerical difficulties.

(*continued on next page*)

Table A.3: Experimental results on the Mészáros/SZTAKI benchmark set. (*continued*)

Name	Rows	Cols	Nzos	SPx objv	Time_S	It	Safe objv_X	Time_X	Safe objv_E	Time_E	Time_{Qex}
SCSD8–2R–16	330	2310	7170	1.599950e+01	0.00	1	1.599950e+01	2.92	1.599950e+01	1.72	0.13
SCSD8–2R–27	550	3850	12010	2.400000e+01	0.00	1	2.400000e+01	7.46	2.400000e+01	4.18	0.17
SCSD8–2R–32	650	4550	14210	1.599900e+01	0.00	1	1.599900e+01	11.33	1.599900e+01	6.28	1.20
SCSD8–2R–4	90	630	1890	1.550000e+01	0.00	1	1.550000e+01	0.31	1.550000e+01	0.22	0.00
SCSD8–2R–54	1090	7630	23890	2.385000e+01	0.01	1	2.385000e+01	37.78	2.385000e+01	17.94	0.80
SCSD8–2R–64	1290	9030	28290	1.584275e+01	0.01	1	1.584275e+01	56.24	1.584275e+01	26.67	0.71
SCSD8–2R–8	170	1190	3650	1.600000e+01	0.00	1	1.600000e+01	0.88	1.600000e+01	0.57	0.02
SCSD8–2R–8B	170	1190	3650	1.600000e+01	0.00	1	1.600000e+01	0.88	1.600000e+01	0.57	0.01
SCTAP1–2B–16	990	1584	5740	2.808000e+02	0.00	1	2.808000e+02	2.37	2.808000e+02	1.39	0.02
SCTAP1–2B–4	270	432	1516	2.392500e+02	0.00	1	2.392500e+02	0.24	2.392500e+02	0.17	0.00
SCTAP1–2C–16	990	1584	5740	3.264000e+02	0.00	1	3.264000e+02	2.37	3.264000e+02	1.41	0.01
SCTAP1–2C–4	270	432	1516	2.362500e+02	0.00	1	2.362500e+02	0.24	2.362500e+02	0.17	0.00
SCTAP1–2C–64	3390	5424	19820	2.003906e+02	0.05	1	2.003906e+02	29.42	2.003906e+02	14.23	0.07
SCTAP1–2R–108	6510	10416	38124	2.480000e+02	0.25	1	2.480000e+02	150.49	2.480000e+02	81.78	0.15
SCTAP1–2R–16	990	1584	5740	3.590000e+02	0.01	1	3.590000e+02	2.37	3.590000e+02	1.46	0.02
SCTAP1–2R–27	1650	2640	9612	2.475000e+02	0.02	1	2.475000e+02	6.27	2.475000e+02	3.61	0.02
SCTAP1–2R–32	1950	3120	11372	3.540000e+02	0.02	1	3.540000e+02	8.79	3.540000e+02	4.94	0.04
SCTAP1–2R–4	270	432	1516	2.805000e+02	0.00	1	2.805000e+02	0.24	2.805000e+02	0.17	0.00
SCTAP1–2R–54	3270	5232	19116	2.492500e+02	0.05	1	2.492500e+02	26.31	2.492500e+02	13.23	0.07
SCTAP1–2R–64	3870	6192	22636	3.440000e+02	0.06	1	3.440000e+02	43.48	3.440000e+02	19.06	0.09
SCTAP1–2R–8	510	816	2924	3.605000e+02	0.01	1	3.605000e+02	0.72	3.605000e+02	0.47	0.01
SCTAP1–2R–8B	510	816	2924	2.500000e+02	0.01	1	2.500000e+02	0.71	2.500000e+02	0.46	0.00
SEYMOURL	4944	1372	33549	4.038465e+02	0.71	1	4.038465e+02	35.32	4.360914e+02	15.31	1.75
SMALL000	709	1140	2749	2.128250e+01	0.01	1	2.128351e+01	0.90	2.328258e+01	0.57	0.12
SMALL001	687	1140	2871	2.007632e+03	0.03	1	2.008172e+03	0.94	2.585305e+03	0.59	0.14
SMALL002	713	1140	2946	3.765764e+01	0.02	1	3.766947e+01	0.97	3.005992e+02	0.61	0.26
SMALL003	711	1140	2945	1.854368e+02	0.02	1	1.854439e+02	0.97	2.172601e+02	0.60	0.16
SMALL004	717	1140	2983	5.468441e+01	0.01	1	5.468495e+01	0.97	5.473207e+01	0.60	0.15
SMALL005	717	1140	3017	3.417138e+01	0.01	1	3.417143e+01	0.97	3.431484e+01	0.57	0.14

[a] condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold, [b] numerical difficulties.

(*continued on next page*)

Table A.3: Experimental results on the Mészáros/SZTAKI benchmark set. (*continued*)

Name	Rows	Cols	Nzos	SPx objv	Time_S	It	Safe objv_X	Time_X	Safe objv_E	Time_E	Time_Qex
SMALL006	710	1138	3024	2.443131e+01	0.01	1	2.443236e+01	0.96	4.014895e+01	0.61	0.14
SMALL007	711	1137	3079	1.388243e+01	0.02	1	1.388370e+01	0.97	3.585775e+02	0.63	0.19
SMALL008	712	1134	3042	8.053826e+00	0.02	1	8.055414e+00	0.96	5.286932e+01	0.62	0.16
SMALL009	710	1135	3030	4.908141e+00	0.01	1	4.913804e+00	0.96	4.988745e+01	0.61	0.16
SMALL010	711	1138	3027	2.262912e+00	0.02	1	2.262979e+00	0.95	2.274826e+00	0.60	0.15
SMALL011	705	1133	3005	8.866442e–01	0.01	1	8.871523e–01	0.92	5.605605e+01	0.59	0.16
SMALL012	706	1134	3014	6.230641e–01	0.01	1	6.233326e–01	0.91	1.222169e+01	0.58	0.13
SMALL013	701	1131	2989	9.630487e–01	0.01	1	9.633818e–01	0.86	3.321504e+00	0.54	0.13
SMALL014	687	1130	2927	1.229840e+00	0.01	1	1.229887e+00	0.85	1.871516e+00	0.54	0.13
SMALL015	683	1130	2967	1.680485e+00	0.01	1	1.680616e+00	0.85	3.829229e+01	0.54	0.14
SMALL016	677	1130	2937	2.203081e+00	0.02	1	2.203207e+00	0.82	2.600518e+00	0.51	0.14
STORMG2–8	4409	10193	27424	1.553524e+07	0.24	1	1.553524e+07	119.35	1.553592e+07	56.28	0.37

[a] condition $\text{cond}(A) \cdot \|E\|/\|A\| < 1$ does not hold, [b] numerical difficulties.

Table A.4: Benchmark results for different SMT approaches.

The first line indicates the solver for the LP, where *ICP w/o LPS* means that no LP solver is used, i.e., all linear constraints are only handled with the ICP-approach, *Neumaier/S* that we use the approach proposed by Neumaier and Shcherbina [69] to certify the infeasibility of the linear program and ICP is not used, *ICP+our* that we use our approach and the ICP-approach of iSAT, *rational LPS* that we use the rational LP solver Yices and ICP is not used, and *our* that we use only our approach and ICP is not used. For each version, we provide the size of the search tree (*nodes*), the running time in seconds (*time*) and the result (*rs*), where *U* means that the solver correctly returned unsatisfiable and *?* that it returned *unknown*. The running time of the fastest approach was additionally marked with boldface.

	ICP w/o LPS			Neumaier/S			ICP+our			rational LPS			our approach		
Benchmark	nodes	rs	time	nodes	rs	time	nodes	rs	time	nodes	rs	time	nodes	rs	time
10cl.m-i.b	80	?	0.69	27197	?	249.29	1198	?	146.61	timeout			timeout		
10cl.w-c-s.b	0	U	**0.03**	0	U	**0.03**	0	U	0.04	0	U	0.04	0	U	**0.03**
10cl.w-c-s.in	timeout			timeout			timeout			timeout			timeout		
2cl.m-i.b	110	?	0.14	1120	?	1.55	179	?	4.54	1429	U	2.21	873	U	**1.17**
2cl.m-i.in	memout			562	?	3.81	timeout			2810	U	141.69	3889	U	**51.4**
2cl.w-c-s.b	0	U	0.01	0	U	0.01	0	U	0.02	0	U	0.02	0	U	**0**
2cl.w-c-s.in	timeout			25	U	**0.13**	20	U	16.22	27	U	0.66	25	U	0.56
3cl.m-i.b	119	?	0.19	2952	?	5.23	260	?	8.65	2166	U	7.08	2153	U	**3.52**
3cl.m-i.in	memout			1963	?	28.92	memout			timeout			4962	?	93.46
3cl.w-c-s.b	0	U	**0.01**	0	U	**0.01**	0	U	0.02	0	U	0.03	0	U	0.02
3cl.w-c-s.in	timeout			150	?	0.69	131	U	62.46	159	U	5.17	150	U	**3.57**
4cl.m-i.b	133	?	0.24	2808	?	7.65	272	?	19.61	3369	U	15.79	4045	U	**10.01**
4cl.m-i.in	memout			4265	?	89.62	memout			timeout			timeout		
4cl.w-c-s.b	0	U	**0.01**	0	U	**0.01**	0	U	0.03	0	U	0.02	0	U	0.02
4cl.w-c-s.in	timeout			351	?	2.48	486	U	157.94	394	U	20.51	379	U	**10.00**
5cl.m-i.b	142	?	0.29	5411	?	18.33	387	?	31.41	5724	U	34.9	7391	U	**21.27**
5cl.m-i.in	memout			10126	?	299	memout			timeout			timeout		
5cl.w-c-s.b	0	U	**0.02**	0	?	0	0	U	**0.02**	0	U	**0.02**	0	?	0.01
5cl.w-c-s.in	timeout			954	?	10.68	timeout			1469	U	63.3	964	U	**25.61**

(*continued on next page*)

Table A.4: Benchmark results for different approaches. (*continued*)

Benchmark	ICP w/o LPS			Neumaier/S			ICP+our			rational LPS			our approach		
	nodes	rs	time	nodes	rs	time	nodes	rs	time	nodes	rs	time	nodes	rs	time
6cl.m-i.b	156	?	0.37	16776	?	70.72	269	?	8.44	7358	U	70.69	11952	U	**38.84**
6cl.m-i.in	memout			5704	?	244.22	timeout			timeout			timeout		
6cl.w-c-s.b	0	U	0.03	0	U	**0.01**	0	U	0.02	0	U	0.03	0	U	0.02
6cl.w-c-s.in	timeout			1189	?	15.54	timeout			2227	U	216.17	2172	U	**58.50**
7cl.m-i.b	170	?	0.52	10231	?	65.95	594	?	51.88	11767	U	151.75	15465	U	**78.39**
7cl.m-i.in	memout			timeout			timeout			timeout			timeout		
7cl.w-c-s.b	0	U	0.03	0	U	0.03	0	U	**0.02**	0	U	0.03	0	U	0.03
7cl.w-c-s.in	timeout			2132	?	42.59	timeout			timeout			8866	U	**277.91**
8cl.m-i.b	179	?	0.58	11877	?	84.4	679	?	100.35	15778	U	207.85	21113	U	**115.45**
8cl.m-i.in	memout			timeout			timeout			timeout			timeout		
8cl.w-c-s.b	0	U	0.02	0	U	0.02	0	U	**0.01**	0	U	0.03	0	U	0.03
8cl.w-c-s.in	timeout			2560	?	68.28	timeout			timeout			timeout		
9cl.m-i.b	193	?	0.69	18838	?	162.94	1193	?	119.28	30200	U	286.39	28586	U	**164.25**
9cl.w-c-s.b	0	U	0.01	0	U	0.02	0	U	0.02	0	U	0.02	0	U	**0.00**
9cl.w-c-s.in	timeout			7791	?	281.53	timeout			timeout			timeout		

Table A.5: Additional details for our SMT approach using an LP solver.

We give some additional details for the instances that are solved: beside the size of the search tree (*nodes*) and the running time (*time*) which can already be found in Table A.4, we give the average size of the basis of the LPs (*basis*), the average size of the infeasible subsystem (*ifs*), the time needed by the LP solver (t_{LPS}), and the time needed for the Gaussian elimination (t_{Gauss}). Furthermore, we give the total number of linear systems that are declared to be infeasible by the floating-point LP solver (*#inf*) and the number of times our method to certify infeasibility is not successful (*#?*).

Benchmark	nodes	time	basis	ifs	t_{LPS}	t_{Gauss}	#inf	#?
10cl.w-case-s.b	0	0.03	31	6	0	0	1	0
2cl.m-i.b	873	1.17	131	7	0.52	0.19	65	0
2cl.m-i.in	3889	51.4	249	10.91	24.6	20.49	367	65
2cl.w-case-s.b	0	0	15	6	0	0	1	0
2cl.w-case-s.in	25	0.56	87	15.76	0.03	0.45	17	0
3cl.m-i.b	2153	3.52	195	7.34	2	0.27	99	0
3cl.m-i.in	4962	93.46	388	11.79	58.8	21.45	461	230
3cl.w-case-s.b	0	0.02	17	6	0	0	1	0
3cl.w-case-s.in	150	3.57	128	17.88	0.51	2.72	71	7
4cl.m-i.b	4045	10.01	267	6.81	5.03	0.21	135	0
4cl.w-case-s.b	0	0.02	19	6	0	0	1	0
4cl.w-case-s.in	379	10	173	19.49	2.72	6.36	141	13
5cl.m-i.b	7391	21.27	347	6.97	11.57	0.44	173	0
5cl.w-case-s.b	0	0.01	21	0	0	0	1	1
5cl.w-case-s.in	964	25.61	222	19.79	9.77	12.4	233	14
6cl.m-i.b	11952	38.84	435	6.86	21.9	0.37	217	0
6cl.w-case-s.b	0	0.02	23	6	0	0	1	0
6cl.w-case-s.in	2172	58.5	275	20.35	27.61	22.74	403	61
7cl.m-i.b	15465	78.39	531	7.04	42.77	0.7	264	0
7cl.w-case-s.b	0	0.03	25	6	0	0	1	0
7cl.w-case-s.in	8866	277.91	332	20.91	171.34	70.51	1048	276
8cl.m-i.b	21113	115.45	635	6.8	59.07	0.69	279	0
8cl.w-case-s.b	0	0.03	27	6	0	0	1	0
9cl.m-i.b	28586	164.25	747	7.26	86.45	0.86	327	0
9cl.w-case-s.b	0	0	29	6	0	0	1	0

Bibliography

[1] T. Achterberg. *Constraint integer programming.* PhD thesis, Technische Universität Berlin, 2007.

[2] E. Althaus and D. Dumitriu. Fast and accurate bounds on linear programs. In J. Vahrenhold, editor, *Proc. 8th International Symposium on Experimental Algorithms (SEA '09)*, volume 5526 of *LNCS*, pages 40–50. Springer, 2009. ISBN 978-3-642-02010-0.

[3] E. Althaus and D. Dumitriu. Certifying feasibility and objective value of linear programs. *Operations Research Letters*, 40(4):292–297, 2012.

[4] E. Althaus, B. Becker, D. Dumitriu, and S. Kupferschmid. Integration of an LP solver into Interval Constraint Propagation. In W. Wang, X. Zhu, and D.-Z. Du, editors, *Proc. 5th International Conference on Combinatorial Optimization and Applications (COCOA '11)*, volume 6831 of *LNCS*, pages 343–356. Springer, 2011. ISBN 978-3-642-22615-1.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, 3rd edition, 1999. ISBN 0-89871-447-8.

[6] D. Applegate, W. Cook, S. Dash, and D. Espinoza. Exact solutions to linear programming problems. *Operations Research Letters*, 35(6):693–699, 2007.

[7] D. G. Bailey and J. M. Borwein. Exploratory experimentation and computation. *Notices of the AMS*, 58(10):1410–1419, 2011.

[8] E. R. Barnes. A variation on Karmarkar's algorithm for solving linear programming problems. *Mathematical Programming*, 36(2):174–182, 1986. ISSN 0025-5610.

[9] F. Benford. The law of anomalous numbers. *Proceedings of the American Philosophical Society*, 74(4):551–572, 1938.

[10] F. Benhamou and L. Granvilliers. Continuous and Interval Constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 16, pages 571–603. Elsevier, 2006.

[11] A. Berman and R. J. Plemmons. *Nonnegative Matrices in the Mathematical Sciences*. SIAM, 2nd edition, 1994.

[12] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997.

[13] F. Besson. On using an inexact floating-point LP solver for deciding linear arithmetic in an SMT solver. In *8th International Workshop on Satisfiability Modulo Theories*, 2010.

[14] R. G. Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2(2):103–107, 1977. ISSN 0364765X.

[15] Boost. C++ Libraries. `http://www.boost.org/`.

[16] J. M. Borwein and R. E. Crandall. Closed forms: what they are and why we care, 2010. In press. Available from `http://www.carma.newcastle.edu.au/~jb616/closed-form.pdf`.

[17] R. Brinkmann and R. Drechsler. RTL-Datapath verification using integer linear programming. In *Design Automation Conference (ASP-DAC)*, pages 741–746. IEEE, 2002. ISBN 0-7695-1299-2.

[18] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT Solver. In A. Gupta and S. Malik, editors, *Computer Aided Verification*. Springer, 2008.

[19] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The OpenSMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2010.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[21] A. R. Curtis and J. K. Reid. On the automatic scaling of matrices for Gaussian elimination. *IMA Journal of Applied Mathematics*, 10(1):118–124, 1972.

[22] G. B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In T. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 339–347. Wiley & Chapman-Hall, 1951. Written in 1947.

[23] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[24] S. de Vries and R. V. Vohra. Combinatorial auctions: A survey. *INFORMS Journal on Computing*, 15(3):284–309, 2003.

[25] S. Dellacherie, S. Devulder, and J.-L. Lambert. Software verification based on linear programming. In J. M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1147–1165. Springer, 1999. ISBN 3-540-66588-9.

[26] S. Devulder and J.-L. Lambert. A comparative study between linear programming validation (LPV) and other verification methods. In *Automated Software Engineering (ASE)*, pages 299–302, 1999.

[27] M. Dhiflaoui, S. Funke, C. Kwappik, K. Mehlhorn, M. Seel, E. Schömer, R. Schulte, and D. Weber. Certifying and repairing solutions to large LPs. How good are LP-solvers? In *Symposium of Discrete Algorithms (SODA)*, pages 255–256, 2003.

[28] M. Dowson. The Ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22 (2):84, Mar. 1997. ISSN 0163-5948.

[29] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer-Aided Verification*, LNCS, pages 81–94. Springer, 2006.

[30] G. Farkas. A Fourier-féle mechanikai elv alkamazásai. *Mathematikai és Természettudományi Értesítő*, 12:457–472, 1894.

[31] J. Farkas. Über die Theorie der einfachen Ungleichungen. *Journal für die reine und angewandte Mathematik*, 124(1):1–27, 1902.

[32] M. Fränzle, C. Herde, S. Ratschan, T. Schubert, and T. Teige. Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. *JSAT Special Issue on Constraint Programming and SAT*, 1:209–236, 2007.

[33] R. M. Freund, R. Roundy, and M. J. Todd. Identifying the set of always-active constraints in a system of linear inequalities by a single linear program. Technical Report 1674–1685, Sloan W.P., 1985.

[34] S. Gao, M. Ganai, F. Ivancic, A. Gupta, S. Sankaranarayanan, and E. Clarke. Integrating ICP and LRA solvers for deciding nonlinear real arithmetic. In *FMCAD*, 2010.

[35] B. Gärtner. Exact arithmetic at low cost – a case study in linear programming. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '98, pages 157–166, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[36] C. F. Gauss. *Theory of the Motion of the Heavenly Bodies Moving About the Sun in Conic Sections*. Perthes & Besser, Hamburg, 1809. Translation by C.H. Davis, 1857.

[37] D. M. Gay. Eletronic mail distribution of linear programming test problems. *Mathematical Programming Society COAL Bulletin*, (13):10–12, Dec. 1985.

[38] GMP. The GNU Multiple Precision arithmetic library. `http://gmplib.org/`.

[39] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computer Surveys*, 23(1):5–48, Mar. 1991. ISSN 0360-0300.

[40] D. Goldfarb and J. K. Reid. A practicable steepest-edge simplex algorithm. *Mathematical Programming*, 12:361–371, 1977. ISSN 0025-5610.

[41] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore and London, 3rd edition, 1996.

[42] Gurobi Optimization. Gurobi optimizer. `http://www.gurobi.com/`.

[43] T. C. Hales, J. Harrison, S. McLaughlin, T. Nipkow, S. Obua, and R. Zumkeller. A revision of the proof of the Kepler conjecture. *Discrete & Computational Geometry*, 44(1):1–34, 2010.

[44] P. M. J. Harris. Pivot selection methods of the Devex LP code. *Mathematical Programming*, 5:1–28, 1973. ISSN 0025-5610.

[45] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Proc. of the 27th Annual Symposium on Theory of Computing*. ACM Press, 1995.

[46] N. J. Higham. A survey of condition number estimation for triangular matrices. *SIAM Review*, 29(4):575–596, Dec. 1987. ISSN 0036-1445.

[47] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2nd edition, 2002.

[48] IBM. ILOG CPLEX optimizer. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.

[49] IEEE Computer Society. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*, 1985. Appeared also as standard ISO/IEC/IEEE 60559:1989.

[50] IEEE Computer Society. *ANSI/IEEE 754-2008, Standard for Floating-Point Arithmetic*, 2008. Appeared also as standard ISO/IEC/IEEE 60559:2011.

[51] L. V. Kantorovich. A new method of solving some classes of extremal problems. *Reports of the Academy of Sciences of the USSR*, 28:211–214, 1940.

[52] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–396, 1984.

[53] C. Keil and C. Jansson. Computational experience with rigorous error bounds for the Netlib linear programming library. *Reliable Computing*, 12 (4):303–321, 2006.

[54] L. G. Khachyian. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, (20):191–194, 1979.

[55] V. Klee and G. J. Minty. How good is the simplex algorithm? In *Inequalities–III*, pages 159–175. Academic Press, New York, 1972.

[56] T. Koch. The final NETLIB-LP results. *Oper. Res. Lett.*, 32(2):138–142, 2004.

[57] R. Levi, D. B. Shmoys, and C. Swamy. LP-based approximation algorithms for capacitated facility location. *Mathematical Programming*, 131(1-2):365–379, 2012.

[58] L. Lovász. A new linear programming algorithm – better or worse than the simplex method? *Mathematical Intelligencer*, 2(3):141–146, 1980.

[59] Maple. Waterloo Maple Inc., Waterloo, Ontario, Canada. `http://www.maplesoft.com/`.

[60] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3(3):255–269, 1957.

[61] Mathematica. Wolfram Research Inc., Champaign, IL, USA. `http://www.wolfram.com/`.

[62] MATLAB. The Mathworks Inc., Natick, MA, USA. `http://www.mathworks.com/`.

[63] N. Megiddo. Pathways to the optimal set in linear programming. In *On Progress in Mathematical Programming: Interior-Point and Related Methods*, pages 131–158. Springer, 1989. ISBN 0-387-96847-4.

[64] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems*, 30(3), 2008.

[65] T. S. Motzkin. *Beiträge zur Theorie der linearen Ungleichungen.* PhD thesis, University of Basel, 1934. Azriel, Jerusalem, 1936.

[66] K. G. Murty. *Linear programming.* John Wiley & Sons Inc., New York, 1983. ISBN 0-471-09725-X.

[67] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, 1988.

[68] Netlib. A Linear Programming Library. `http://www.netlib.org/lp/`.

[69] A. Neumaier and O. Shcherbina. Safe bounds in linear and mixed-integer linear programming. *Math. Program.*, 99(2):283–296, 2004.

[70] B. Nuseibeh. Ariane 5: Who dunnit? *IEEE Software*, 14(3):15–16, May 1997.

[71] S. Obua. *Flyspeck II: the basic linear programs*. PhD thesis, Technische Universität München, 2008.

[72] F. Ordóñez and R. M. Freund. Computational experience and the explanatory value of condition measures for linear optimization. *SIAM Journal on Optimization*, 14(2):307–333, 2003.

[73] F. Pigorsch and C. Scholl. Using implications for optimizing state set representations of linear hybrid systems. In *GI/ITG/GMM Workshop MBMV*, 2009.

[74] D. Poole. *Linear Algebra: A Modern Introduction*. Thomson Brooks/Cole, 2nd edition, 2006.

[75] R. A. Raimi. The first digit problem. *American Mathematical Monthly*, 83: 521–538, 1976.

[76] M. G. C. Resende, K. G. Ramakrishnan, and Z. Drezner. Computing lower bounds for the quadratic assignment problem with an interior point algorithm for linear programming. *Operations Research*, 43:781–791, 1995.

[77] SMT-LIB. The Satisfiability Modulo Theories library. `http://goedel.cs.uiowa.edu/smtlib/`.

[78] SoPlex. The Sequential object-oriented simplex. `http://soplex.zib.de/`.

[79] D. E. Steffy. *Topics in exact precision mathematical programming*. PhD thesis, Georgia Institute of Technology, 2011.

[80] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. Univ. of California Press, Berkeley, 1951.

[81] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pages 115–125, 1970.

[82] R. Vanderbei, M. Meketon, and B. Freedman. A modification of Karmarkar's linear programming algorithm. *Algorithmica*, 1:395–407, 1986. ISSN 0178-4617.

[83] J. Varah. A lower bound for the smallest singular value of a matrix. *Linear Algebra and its Applications*, 11(1):3–5, 1975.

[84] R. S. Varga. On diagonal dominance arguments for bounding $\|A^{-1}\|_\infty$. *Linear Algebra and its Applications*, 14(3):211–217, 1976.

[85] D. S. Watkins. *Fundamentals of Matrix Computations*. John Wiley & Sons, Inc., New York, 2nd edition, 2002.

[86] R. Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus*. PhD thesis, Technische Universität Berlin, 1996. `http://www.zib.de/Publications/abstracts/TR-96-09/`.

[87] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Matrix Analysis and Applications*, 2(1):77–79, 1981.

[88] Y. Ye. An $O(n^3L)$ potential reduction algorithm for linear programming. *Mathematical Programming*, 50:239–258, 1991. ISSN 0025-5610.