Towards an Efficient Parallel Parametric Linear Programming Solver

Hang Yu

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Mathématiques et Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Hang YU

Thèse dirigée par **David MONNIAUX**, Communauté Université Grenoble Alpes
et codirigée par **Michaël PERIN**

préparée au sein du **Laboratoire VERIMAG**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Vers un solveur de programmation linéaire paramétrique parallèle efficace

Towards an Efficient Parallel Parametric Linear Programming Solver

Thèse soutenue publiquement le **19 décembre 2019**,
devant le jury composé de :

Monsieur DAVID MONNIAUX
DIRECTEUR DE RECHERCHE, CNRS DELEGATION ALPES, Directeur de thèse
Monsieur MICHAËL PERIN
MAITRE DE CONFERENCES, UNIVERSITE GRENOBLE ALPES, Examinateur
Monsieur PHILIPPE CLAUSS
PROFESSEUR, UNIVERSITE DE STRASBOURG, Rapporteur
Monsieur CHRISTOPHE ALIAS
CHARGE DE RECHERCHE HDR, INRIA CENTRE DE GRENOBLE RHÔNE-ALPES, Rapporteur
Monsieur LIQIAN CHEN
MAITRE DE CONFERENCES, UNIV. NATIONALE TECH. DEFENSE A CHANGSHA, Examinateur
Madame NADIA BRAUNER
PROFESSEUR, UNIVERSITE GRENOBLE ALPES, Examinateur

Président de jury: **Madame NADIA BRAUNER**

Contents

Notations

Capital letters (e.g. A) denote matrices, small bold letters (e.g. $\boldsymbol{x}$) denote vectors, and small letters (e.g. b) denote scalars. The ith row of A is $\boldsymbol{a}_{i\bullet}$, its jth column is $\boldsymbol{a}_{\bullet j}$. For simplification, we use $\boldsymbol{a}_i$ to abridge $\boldsymbol{a}_{i\bullet}$. The element at the ith row and the jth column of the matrix A is denoted by a_{ij}. The superscript T (e.g. A^{T}) represents the transpose of a matrix or a vector. $\mathcal{P}$ denotes a polyhedron and $\mathcal{C}$ a constraint. The ith constraint of $\mathcal{P}$ is $\mathcal{C}_i$. $\mathbb{Q}$ denotes the field of rational numbers, and $\mathbb{F}$ is the set of finite floating-point numbers, which is a subset of $\mathbb{Q}$. In the pseudo-code the floating point (or rational) variables are denoted by name$^{\mathbb{F}}$ (or name$^{\mathbb{Q}}$); name$^{\mathbb{Q}\times\mathbb{F}}$ means that the variables are stored in both rational and floating point numbers.

Contributions

We improved on the approach of the PLP solver represented in the work [1] in several aspects and provided a more efficient implementation.

Floating-point arithmetic We replace most of the exact computations in arbitrary precision rational numbers by floating-point computations, which is performed using an off-the-shelf linear programming solver called GLPK. We can however recover exact solutions and check them exactly, following an approach previously used for SMT-solving [2, 3]. We provide checkers to guarantee the soundness of the results, and thus the floating-point solver will always return a sound solution. When it is necessary, a rational solver is invoked which guarantees the precision of our result.

Degeneracy We resolve some difficulties due to geometric degeneracy, which previously resulted in redundant computations. We aim at obtaining a partition of the space without overlapping areas because of two reasons: i) we want to avoid the redundant computations; ii) when there is no overlaps, we are able to check the precision of our result. But the occurrence of degeneracy leads to overlaps, and thus we need to deal with the degeneracy. We proved that there is no *dual degeneracy* in our PLP solver, and we proposed a method to avoid the *primal degeneracy*. Then we can check the precision of our result. If the result is not precise, i.e., there are missing solutions of PLP, a rational solver will be launched for computing these missing solutions.

Parallelism We implemented two level parallelism: in the first level we parallelized the algorithm of the PLP solver; the minimization algorithm launched by the PLP solver is parallelized as the second level. We used two strategies to parallelize the PLP algorithm: one used Intel's Thread Building Blocks (TBB), and the other used OpenMP tasks [4]. The details will be explained in Section 5.3.

Outline

In Chapter 1 we will present some related work. Some content related to *polyhedra abstract domain* will be explained. We will also introduce the *Verified Polyhedra Library (VPL)*, on which our work is based.

In Chapter 2 we first present preliminary knowledge about convex polyhedra. Then some basic knowledge about floating point arithmetic will be exposed, as most of our computation use floating point numbers. We will show what are *linear programming (LP)* and *parametric linear programming (PLP)*, as the core of our work is solving LP and PLP problems. There are various methods to solve LP problems, but we will only present and use the *Simplex algorithm*.

Chapter 3 shows the sequential algorithm for computing polyhedral *minimization*, *projection* and *convex hull*. A flowchart at the beginning of Chapter 3 illustrates the process of our algorithm, and we then explain the cooperation between rational and floating point arithmetic.

Chapter 4 explains the degeneracy, which is a difficulty in our work. We will explain what are *primal degeneracy* and *dual degeneracy* respectively with examples. We will show that degeneracy of PLP will result in *overlapping regions*, which affects the precision of our approach. An algorithm will be shown for avoiding the overlapping regions. We will also illustrates some checkers at several steps in our floating point PLP solver, which ensure the correctness and precision of our floating point solver with respect to the rational solver. We provide several rational procedures which have been implemented in floating point numbers. Once a floating point procedure is proved to be incorrect or not applicable, the corresponding rational procedure will be launched for computing the exact result.

In Chapter 5 the details of our parallelized algorithm of minimization and PLP solver will be presented. Some basic knowledge about the libraries for parallel programming will be introduced. Then we will detail our implementation.

The experiments of *minimization*, *polyhedral projection* and convex hull will be shown in Chapter 6. We will focus on the performance analysis. The performance is affected by several parameters. We will change one parameter each time and maintain the others to figure out the situation where our algorithm of PLP solver has advantages and drawbacks compared to other polyhedra libraries. Then we will use a static analyzer to extract some randomly selected programs from SV-COMP benchmarks [5], and compare our approach with other state-of-art libraries on these benchmarks.

Chapter 7 shows some possible future work. Our approach contains rational solvers for obtaining rational solutions and guaranteeing the correctness and precision of our algorithm. When the exact solutions are not required, it is possible to use a pure floating point solver. We need to guarantee the soundness in each step of computation.

Chapter 1

Introduction

Abstract interpretation [6] is a theory of sound approximation of the semantics of computer programs, and it is used for obtaining invariant properties of programs, which may be used to verify their correctness. Abstract interpretation searches for invariants within an *abstract domain*. For numerical properties, a common and cheap choice is one interval per variable per location in the program, but it cannot represent relationships between variables. Such imprecision often makes it difficult to prove properties of the program using the interval domain. If we retain linear equalities and inequalities between variables, we will obtain the domain of *convex polyhedra* [7], which is more expensive, but more precise. The search for polyhedral invariants are provided in [8, 9]. The needed operations are projection, convex hull, inclusion test and equality test, together with the operator widening [7, 10].

The abstract interpretation is *sound* if it prevents *false negative*, meaning that a state which could present during execution is missing. In other words, the semantics described by the abstraction should cover all the possible semantics during the execution. For guaranteeing the soundness the abstract interpretation should be over-approximate, which means that if a property is proved to be true in the abstract domain, the property must not be violated at run time. Considering an abstract convex polyhedron $\mathcal{P}$, in a sound system, if the initial state belongs to $\mathcal{P}$, all possible time steps will never end outside $\mathcal{P}$.

Several implementations of the domain of convex polyhedra over the field of rational numbers are available. The most popular ones for abstract interpretation are NewPolka[1] and the Parma Polyhedra Library (PPL) [11]. These libraries, and others [12, 13], use the *double description* of polyhedra: as *generators* (vertices, rays and lines) and constraints (linear equalities and inequalities). Some operators are easier to be computed on one description than on the other, and some, such as removing redundant constraints or generators, are easier if both descriptions are available. To benefit from the efficiency of double description, it is necessary to maintain the equivalence between the two descriptions. One description is computed from the other using Chernikova's algorithm [14, 15], which is expensive in some cases. Furthermore, in some cases, one description is exponentially larger than the other. This in particular occurs in the cases of the generator description of hypercubes, i.e., products of intervals.

[1] Now distributed as part of Apron: `http://apron.cri.ensmp.fr/library/`

In 2012 Verimag started implementing a library using constraint-only description, which is called VPL (Verified Polyhedra Library)[2] [16, 1]. There are several reasons for using only constraints. We have already cited the high complexity of the generator description of some polyhedra commonly found in abstract interpretation. Besides, the high cost of Chernikova's algorithm is able to be avoided if we use the single description. Another reason is to be able to certify the results of the computation, in particular that the obtained polyhedra includes the one that should have been computed, i.e., the soundness of the result.

In the first version of VPL, which was implemented by Alexis Fouilhé in 2015 [16], all main operations boil down to projection, which is performed using *Fourier-Motzkin elimination* [17]. This method generates many *redundant constraints* which must be eliminated at a high cost. Besides, for projecting out multiple variables, it computes all intermediate steps. For each variable λ_i to be eliminated, the inequalities are divided into three groups depending on the sign (positive, negative and zero) of the coefficient of λ_i. Then it computes the combination of each inequality from the positive group and that from the negative group. These intermediate steps may have high description complexity.

Jones et al. explained the relationship between polyhedral projection and *parametric linear programming (PLP)* in [18]. Based on this work, [19] proposed a new approach to compute polyhedral projection using PLP, which can replace Fourier-Motzkin elimination. Inspired by [18] and [19], Alexandre Maréchal implemented a novel algorithm for computing the projection and convex hull, which both boil down to PLP, and released version 2.0 of the VPL.

VPL 2.0 was implemented in arbitrary precision arithmetic in OCaml [20]. The algorithm of the PLP solver is intrinsically parallelable, however it is developed with OCaml, which does not well support parallelism programming. Our work focuses on improving the efficiency of the algorithm of PLP solver, and parallelizing it for obtaining a better performance.

In recent years some state-of-the-art methods are presented that aim at improving the efficiency of polyhedra domain. The expensive cost of polyhedra domain is partially caused by the "dimension curse" of double description. Thus there appears to be different ideas to solve the problem. What we did was getting rid of the double description and developing new algorithms for the operators that are inefficient using constraint-only description. There are other methods which focus on reducing the dimension of the polyhedra [21, 13]. In the polyhedra library ELINA[3] [13] the variables are divided into small groups, where the variables in different groups are not related to each other. The authors compared the performance of ELINA with the NewPolka library in Apron, and PPL library. By decreasing the dimension of polyhedra, i.e, the number of variables, the ELINA library has a speed up of dozens of times to thousands of times.

In other aspects, most of the existing polyhedra libraries use rational arithmetic, and thus some new attempts use floating point arithmetic to improve the efficiency. Apart from our approach, the work in [22] also presented an polyhedra domain in floating point numbers. The authors used constraint-only description, and implemented a sound approach based on Fourier-Motzkin elimination. As the floating

[2] Contributors: Alexis Fouilhé, Alexandre Maréchal, Sylvain Boulmé, Hang Yu, Michaël Périn, David Monniaux.
[3] http://elina.ethz.ch/

point arithmetic is not precise, they obtain a set of linear interval inequalities after eliminating one variable. Then they apply a sound linearization approach to these inequalities. Comparing with their method, our approach has the advantage in several aspects: i) our approach gets rid of the intermediate steps of Fourier-Motzkin elimination, which produce amount of redundant constraints; ii) we can project out multiple variables each time; iii) our results are more precise as we reconstruct the rational solutions.

Chapter 2

Preliminaries

In this chapter, we will present some important preliminaries including convex polyhedra, floating point arithmetic, non-parametric and parametric linear programming, etc.

The *polyhedra abstract domain* is firstly presented by P. Cousot and N. Halbwachs in [8]. A convex polyhedron is composed of a set of linear constraints (equalities or inequalities). In static analysis, a linear constraint represents the relationship between the variables of a program. Compared with other numerical abstract domains, polyhedra domain is the most precise but most expensive one. There are dozens of polyhedral operators in the polyhedra domain. In this paper we will mainly talk about minimization, projection and convex hull.

A floating point number is the representation of a rational number in the computers, and the representation may be approximate because of the *rounding errors*. When we use floating point arithmetic, it is necessary to measure the round errors to guarantee the correctness of the results. The study of the consequence of the rounding errors is a topic of *numerical analysis.* In this paper, we used some basic knowledge of numerical analysis, which is explained in [23].

Linear programming is a method to compute the optimal value of a linear objective function subjecting to a set of linear constraints. In the past decades several methods to solve linear programming problems have been presented [24, 25]. In this paper we will only use Dantzig's Simplex algorithm. The difference between non-parametric and parametric linear programming is that the objective function of the latter contains parameters. By setting the parameters to various values we obtain a set of non-parametric linear programs whose objective functions are different. The parametric linear program can be solved by computing these non-parametric linear programs and generalizing the results. We will also present the method to solve a PLP problem using the Simplex algorithm directly. Both methods will be used in the following chapters.

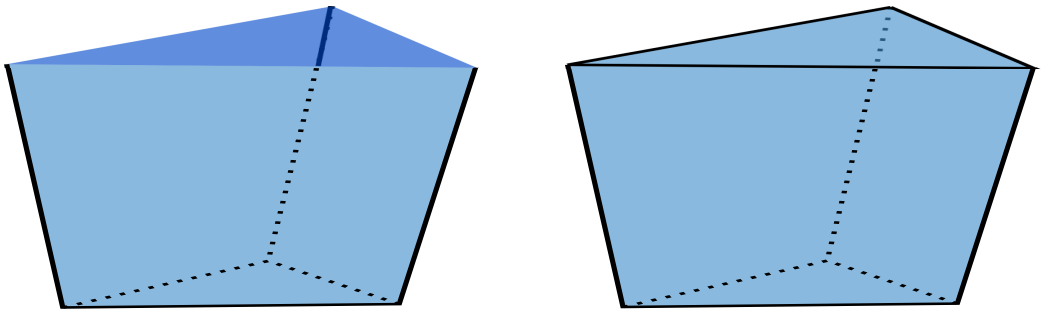
2.1 Convex polyhedra and polyhedra domain

As we aim to implement an algorithm to manipulate the operations on convex polyhedra, in this section we firstly present the definition and the descriptions of a convex polyhedron. We also introduce some operators in the polyhedra domain.

2.1.1 Convex polyhedra

A polyhedron is made of faces, edges and vertices. A polyhedron is *bounded* if it does not contain unbounded faces.

> **Example 2.1.** The Figure 2.1(a) shows an unbounded polyhedron, which has 4 faces and 3 of them are unbounded; the bounded polyhedron which contains 5 bounded faces is shown in Figure 2.1(b).



(a) Unbounded polyhedra. (b) Bounded polyhedra.

Figure 2.1

All the polyhedra we will mention in this paper are *convex* polyhedra, and thus we will omit the word "convex" in the following context. According to [26], a convex set is defined as in Definition 2.1.

> **Definition 2.1** (Convex set)**.** A set $K \subset \mathbb{Q}^n$ is convex if and only if for each pair of distinct points $\boldsymbol{a}$, $\boldsymbol{b} \in K$, the closed segment with endpoints $\boldsymbol{a}$ and $\boldsymbol{b}$ is contained in K.

2.1.2 Double description of polyhedra

A polyhedron can be represented by constraints or generators. A constraint defines a linear relation between variables. It can be an equality, a strict or non-strict inequality. A constraint C_i on variables $\boldsymbol{\lambda}$ in $\mathbb{Q}^n$ is represented by Equation 2.1.

$$\boldsymbol{a}_i^{\mathsf{T}} \boldsymbol{\lambda} \leq b_i \tag{2.1}$$

where $\boldsymbol{a}_i = [a_{i0}, \cdots, a_{in}]^{\mathsf{T}}$ are the coefficients of constraint C_i and b_i is a constant. In the equation $\leq$ can also be $<, >, =$ or $\geq$, and we use $\leq$ as an example here.

Equation 2.2 shows a polyhedron defined by the conjunction of m constraints.

$$\mathcal{P} = \{\boldsymbol{\lambda} \in \mathbb{Q}^n | \bigwedge_{i=1}^{m} \boldsymbol{a}_i^{\mathsf{T}} \boldsymbol{\lambda} \leq b_i\} \tag{2.2}$$

The Equation 2.2 can be represented in the matrix form. Give $A \in \mathbb{Q}^{m \times n}$ and $\boldsymbol{b} \in \mathbb{Q}^m$, the same polyhedron can be represented by Equation 2.3.

$$\mathcal{P} = \{\boldsymbol{\lambda} \in \mathbb{Q}^n | A\boldsymbol{\lambda} \leq \boldsymbol{b}\} \tag{2.3}$$

A polyhedron can also be represented by generators, i.e. vertices, rays, and lines, as it is shown in Equation 2.4. A bounded polyhedron is represented by a set of vertices, and an unbounded polyhedron is represented by the combination of vertices, rays and lines. The rays and lines define the unbounded directions. A line can be considered as two rays which are collinear and in opposite directions.

$$\mathcal{P} = \{\boldsymbol{\lambda} \in \mathbb{Q}^n | \boldsymbol{\lambda} = \sum_{i=1}^{|\mathcal{V}|} \alpha_i \boldsymbol{v}_i + \sum_{i=1}^{|\mathcal{Y}|} \beta_i \boldsymbol{r}_i + \sum_{i=1}^{|\mathcal{L}|} \gamma_i \boldsymbol{l}_i, \alpha_i, \beta_i \geq 0, \sum_{i=1}^{|\mathcal{V}|} \alpha_i = 1\} \tag{2.4}$$

where $\mathcal{V}$ is a set of vertices, $\mathcal{Y}$ is a set of rays, $\mathcal{L}$ is a set of lines, and $\alpha_i,\ \beta_i,\ \gamma_i$ are constants.

Example 2.2. We can represent the unbounded polyhedron in Figure 2.2 by constraints as:

$$\mathcal{P} = \{2\lambda_2 - 1 \geq 0, \lambda_1 - \lambda_2 \geq 0\}$$

The corresponding generator description is:

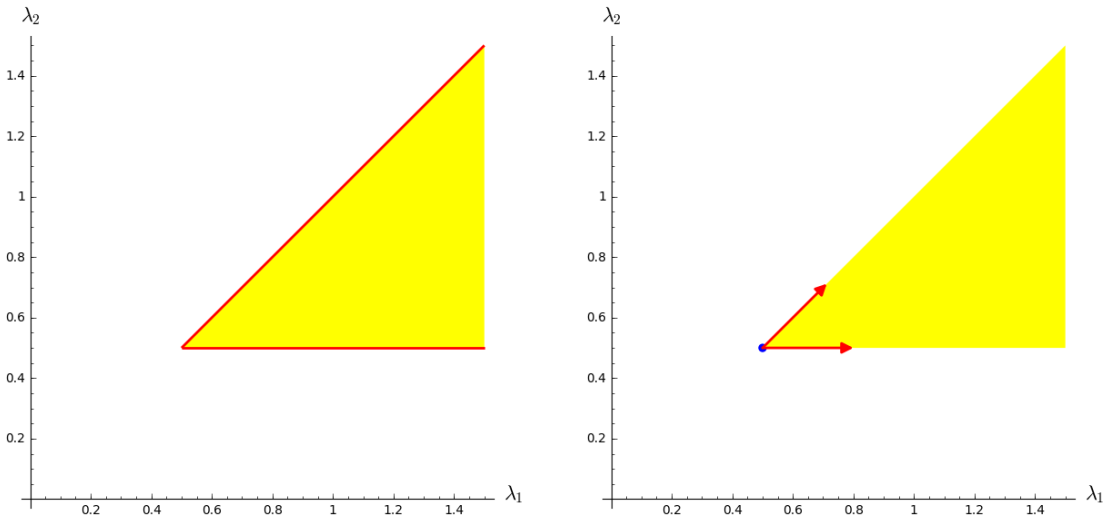
$$\mathcal{V} = \{(\frac{1}{2}, \frac{1}{2})\}$$
$$\mathcal{Y} = \{(1,0), (1,1)\}$$

2.1.3 Operators in polyhedra domain

The polyhedra domain contains dozens of operators. We will present the most frequently used ones.

Elimination of redundancy A polyhedron may contain constraints that are redundant, meaning that removing these constraints will not change the polyhedron. The process of removing redundant constraints is called *minimization*. A polyhedron whose redundant constraints are removed is called a *minimized* polyhedron. We will present this operator in Section 3.2.

Meet The meet of polyhedra $\mathcal{P}_1$ and $\mathcal{P}_2$, which is denoted by $\mathcal{P}_1 \sqcap \mathcal{P}_2$, is the conjunction of the constraints of these two polyhedra, and then the result needs to be minimized.



(a) Constraint description.

(b) Generator description.

Figure 2.2

Join The join operator computes the convex hull of polyhedra. The convex hull of $\mathcal{P}_1$ and $\mathcal{P}_2$, i.e. $\mathcal{P}_1 \sqcup \mathcal{P}_2$, is the smallest convex set that contains $\mathcal{P}_1$ and $\mathcal{P}_2$. This operator will be introduced in Section 3.3.

Inclusion The polyhedron $\mathcal{P}_1$ is included in $\mathcal{P}_2$ if $\forall \boldsymbol{x} \in \mathcal{P}_1$, $\boldsymbol{x} \in \mathcal{P}_2$. The inclusion test can be implemented by testing redundancy: $\mathcal{P}_1 \sqsubseteq \mathcal{P}_2$ if all the constraints of $\mathcal{P}_2$ are redundant with respect to $\mathcal{P}_1$.

Equality The polyhedra $\mathcal{P}_1 = \mathcal{P}_2$ if they define the same convex set, i.e. the two polyhedra contain equivalent constraints. The equality test can be implemented by inclusion test: $\mathcal{P}_1 = \mathcal{P}_2$ if $\mathcal{P}_1 \sqsubseteq \mathcal{P}_2$ and $\mathcal{P}_2 \sqsubseteq \mathcal{P}_1$.

Projection The projection operator is used to eliminate variables and project the polyhedron to lower dimension. It will be presented in Section 3.3.

Widening The widening operator is used for the loops. It is used to enforce the convergence of iteration sequences or accelerate converging steps of the iterations.

2.2 Floating point arithmetic

As floating point arithmetic accounts for a large proportion in our computation, we will illustrate the fundamental of floating point numbers. We start from the representation of a floating point number, and then introduce the *rounding errors*. We also present some basic knowledge of numerical analysis [23] which is used to measure the rounding errors.

Floating point numbers

In computers real numbers are represented approximately by floating-point numbers in the form $dd\cdots d \times \beta^e$, where $dd\cdots d$ is called significand, e is exponent, and β is the base. To unify the form of floating point representation, IEEE (Institute of Electrical and Electronics Engineers) [27] defined a technical standard for floating point numbers in 1990s, which is called IEEE Standard for Floating-Point Arithmetic (IEEE 754). There are three specifications in IEEE 754:

- the base β, which is always 2 ;
- the precision, i.e., the number of digits in $dd\cdots d$
- an exponent range from e_{min} to e_{max}, with $e_{min} = 1 - e_{max}$

According to IEEE 754 standard, the format of a floating point number is shown in Table 2.1. The sign bit is either 0 for positive or 1 for negative. The significand of a normalized number is always in the range of 1 to 2, i.e. in the form $1.d\cdots d$, and thus there is no need to represent the leading digit 1 explicitly. So the significand $1.d\cdots d$ is stored as $d\ldots d$. A bias needs to be added to the exponent for representing negative numbers. The infinity $+\infty$ and $-\infty$ are denoted by setting the exponent bits all ones and the significand all zeros. The numbers in form $0.d\cdots d$, which are called *denormalized numbers*, are represented by setting the exponent all zeros.

	total	sign	exponent	significand
single precision	32 bits	1 bit	8 bits	24 bits (one bit is implicit)
double precision	64 bits	1 bit	11 bits	53 bits (one bit is implicit)

Table 2.1 – floating point numbers format in IEEE 754 standard

Example 2.3. We use the subscript to represent the base, which is either 2 or 10. Let us see how to store the decimal number 0.40625_{10} as single precision binary floating point number. $0.40625_{10} = 0.25_{10} + 0.125_{10} + 0.03125_{10} = 2_{10}^{-2} + 2_{10}^{-3} + 2_{10}^{-5} = 0.01101_2 = 1.101_2 \times 2^{-2}$. The sign bit stores 1, as it is positive. There are 8 bits for exponent, and the range is -127_{10} to 128_{10}, so we need to add a bias 127_{10} to the exponent. Thus we have $-2_{10} + 127_{10} = 125_{10} = 01111101_2$. For the significand, we only store 101. Then we obtain the binary number, which is shown in Table 2.2.

Rounding errors

Floating point numbers are not precise. The most common reason is that some real numbers cannot be represented by limited binary numbers. As floating point numbers are stored as finite binary numbers, the infinite binary numbers or the numbers with too many digits will be rounded. Therefore, there is a difference between the floating point number and the exact value, which is called rounding errors.

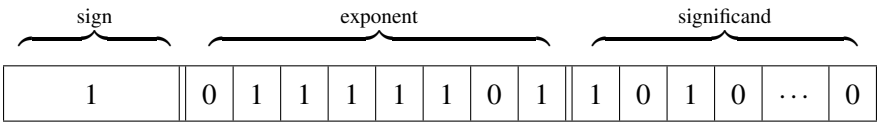


Table 2.2 – floating point numbers format in IEEE 754 standard

Example 2.4. What will happen if we run the piece of code below?

```
double a = 0.1 ;
double b = 0.6 ;
assert(a * 6 == b) ;
```

The assert will fail, because the decimal number 0.1 cannot be finitely represented by a binary number: $0.1_{10} = 0.0\overline{0011}_2$ which means the digits 0011 would repeat forever in an infinite representation. Then the binary number will be truncated to limit bits. The stored value could be less or greater than 0.1 depending on the number of significant bits.

Machine epsilon

Because of the rounding errors, we cannot compare two floating-point numbers directly. Instead, we need to use a threshold. To find out an appropriate threshold, it is necessary to measure the error between the floating-point number and the real number it represents. The measurement of the quantity of rounding errors is a problem in the field of numerical analysis. The rounding errors can be measured by the *machine epsilon*, whose definition is given by Definition 2.2 according to [23].

Definition 2.2 (Machine epsilon)**.** The machine epsilon ε is the difference between 1.0 and the next representable floating point number greater than 1.0.

Given the precision p, according to IEEE 754 standard, we store $p-1$ digits in the significand part, and the smallest representable number is obtained by filling the significand bits with $00\ldots01$, which represents 2_2^{1-p}. Thus we know $\varepsilon = 2_2^{1-p}$. For instance, in double precision $\varepsilon = 2_2^{-52} \approx 2.22_{10}^{-16}$.

Basic arithmetic model

In the basic arithmetic model, the rounding errors between the floating point and the exact computation of an operator op is given by Equation 2.5 [23].

$$fl(x\ op\ y) = \frac{x\ op\ y}{1+\delta} \qquad op \in \{+,\ -, \times,\ \div\} \tag{2.5}$$

where $|\delta| < u$, $u = \frac{1}{2}\varepsilon$ and ε is the machine epsilon. From *Equation* 2.5 we have:

$$|fl(x\ op\ y) - (x\ op\ y)| < \varepsilon |fl(x\ op\ y)| \tag{2.6}$$

where $|\cdot|$ represents the absolute value.

Error bounds of inner product

The only rounding errors we need to measure in our algorithm is that of inner product of floating-point numbers. According to [23], the error between the inner product of two vectors $\boldsymbol{a}^\mathsf{T}\boldsymbol{r}$ and its floating-point value $fl(\boldsymbol{a}^\mathsf{T}\boldsymbol{r})$ is:

$$|fl(\boldsymbol{a}^\mathsf{T}\boldsymbol{r}) - \boldsymbol{a}^\mathsf{T}\boldsymbol{r}| \leq |fl(\boldsymbol{a})|^\mathsf{T}|fl(\boldsymbol{r})|\frac{\alpha u}{1-\alpha u} \tag{2.7}$$

where $\alpha = \lceil \log_2 m \rceil + 1$, m is the dimension of $\boldsymbol{a}$ and $\boldsymbol{r}$, and $u = \frac{1}{2}\varepsilon$.

Furthermore, Equation 2.8 can be derived from Equation 2.7.

$$|fl(\boldsymbol{a}^\mathsf{T}\boldsymbol{r}) - \boldsymbol{a}^\mathsf{T}\boldsymbol{r}| \leq 1.01mu|fl(\boldsymbol{a})|^\mathsf{T}|fl(\boldsymbol{r})| \tag{2.8}$$

where m is the size of the vector. As $u < \varepsilon$, we can use ε to replace u for simplifying implementation. To simplify the explanation, we use $t(\boldsymbol{a},\boldsymbol{r})$ to represent the rounding error of the inner product of $\boldsymbol{a}$ and $\boldsymbol{r}$.

$$t(\boldsymbol{a},\boldsymbol{r}) = 1.01m\varepsilon|fl(\boldsymbol{a})|^\mathsf{T}|fl(\boldsymbol{r})| \tag{2.9}$$

In most part of our procedure we do not consider the rounding errors during computation since we reconstruct the final result in rational numbers and verify that it is a solution of the initial rational problem. In several steps we need to consider and measure the rounding to guarantee the correctness of the floating point arithmetic.

2.3 Linear programming

The *linear programming (LP)* is the core of our algorithm. In order to develop an efficient parametric linear programming solver, we need to solve a set of linear programming problems. Therefore, we will present the form of linear programming in this section.

Definition of an LP problem

Definition 2.3 (Linear function)**.** A linear function $f(\boldsymbol{x})$ is a function that satisfies additivity and homogeneity, i.e., $f(\boldsymbol{x}+\boldsymbol{y}) = f(\boldsymbol{x}) + f(\boldsymbol{y})$ and $f(\boldsymbol{x}) = \alpha f(\boldsymbol{x})$, where $\boldsymbol{x}$ are variables. A linear equation is an equation in the form $\sum_{i=0}^{n} a_i x_i + b = 0$, where a_i and b are constants.

Given a function $f(\boldsymbol{x}) = \sum_{i=0}^{n} a_i x_i + b$, when $b \neq 0$, $f(\boldsymbol{x})$ does not satisfy Definition 2.3, and it is called *linear affine function*. But sometimes, as what we will do in this paper, a linear affine function is simply called linear function.

Linear programming (LP) is a method to find the optimal value of a linear function subject to a set of linear constraints. Given variables $\boldsymbol{\lambda} = [\lambda_1, \ldots, \lambda_n]^\mathsf{T}$, an LP problem in *standard form* is expressed in

Problem 2.10.

$$\begin{aligned} \text{maximize} \quad & Z(\boldsymbol{\lambda}) = \sum_{j=1}^{n} o_j \lambda_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} \lambda_j = b_i \quad \forall i \in \{1, \cdots m\} \\ \text{and} \quad & \boldsymbol{\lambda} \geq 0 \end{aligned} \tag{2.10}$$

where $Z(\boldsymbol{\lambda})$ is the linear objective function, $\sum_{j=1}^{n} a_{ij}\lambda_j = b_i$ $(\forall i \in \{1, \cdots m\})$ is the set of linear constraints, and a_{ij} and o_j are coefficients.

Standard form of an LP problem There are several ways to express an LP problem. In this paper, except in a few cases, we will use the *standard form* to describe the LP problems, as it is shown in Problem 2.10. All the LP problems can be transformed into standard form. For the objective function, minimizing $Z(\boldsymbol{\lambda})$ is equivalent to maximizing $-Z(\boldsymbol{\lambda})$. The constraint $\boldsymbol{a}_i\boldsymbol{\lambda} \leq b_i$ can be transformed into equality by adding a slack variable $s_i \geq 0$ for each constraint $\mathcal{C}_i$: $\boldsymbol{a}_i\boldsymbol{\lambda} + s_i = b_i$. We can also express the constraint $\boldsymbol{a}_i\boldsymbol{\lambda} \leq b_i$ as $r_i = \boldsymbol{a}_i\boldsymbol{\lambda},\ r_i \leq b_i$, where r_i is called *row variable*. In other words, there are two alternative ways to transform the constraint $\mathcal{C}_i$: $\boldsymbol{a}_i\boldsymbol{\lambda} \leq b_i$ into standard form: $\mathcal{C}_i$: $\boldsymbol{a}_i\boldsymbol{\lambda} + s_i = b_i, s_i \geq 0$ and $r_i = \boldsymbol{a}_i\boldsymbol{\lambda},\ r_i \leq b_i$, which are equivalent. If a decision variable λ_i does not have the lower bound 0, we can split it into two variables: $\lambda_i = \lambda_i' - \lambda_i'',\ \lambda_i',\ \lambda_i'' \geq 0$.

Example 2.5. Assuming we have an LP problem in Problem 2.11, and we will transform it into standard form.

$$\begin{aligned} \text{minimize} \quad & Z(\boldsymbol{\lambda}) = 2\lambda_1 + \lambda_2 \\ \text{subject to} \quad & \lambda_1 - 2\lambda_2 \leq -4 \\ & 2\lambda_1 + \lambda_2 \geq 1 \end{aligned} \tag{2.11}$$

First we transform the objective function into:

$$\text{maximize} \quad Z(\boldsymbol{\lambda}) = -2\lambda_1 - \lambda_2$$

Then we unify the constraints with $\leq$ operator and obtain $\{\lambda_1 - 2\lambda_2 \leq -4,\ -2\lambda_1 - \lambda_2 \leq -1\}$. Then to make $\boldsymbol{\lambda} \geq 0$, we split the variables as $\lambda_1 = \lambda_1' - \lambda_1'',\ \lambda_2 = \lambda_2' - \lambda_2''$, where $\lambda_1',\ \lambda_1'',\ \lambda_2',\ \lambda_2'' \geq 0$. The original constraints become $\{\lambda_1' - \lambda_1'' - 2\lambda_2' + 2\lambda_2'' \leq -4,\ -2\lambda_1' + 2\lambda_1'' - \lambda_2' + \lambda_2'' \leq -1,\ \lambda_1',\ \lambda_1'',\ \lambda_2',\ \lambda_2'' \geq 0\}$. By adding the slack variables, we have the LP in standard form, which is shown in Problem 2.12.

$$\begin{aligned} \text{maximize} \quad & Z(\boldsymbol{\lambda}) = -2\lambda_1 - \lambda_2 \\ \text{subject to} \quad & \lambda_1' - \lambda_1'' - 2\lambda_2' + 2\lambda_2'' + s_1 = -4 \\ & -2\lambda_1' + 2\lambda_1'' - \lambda_2' + \lambda_2'' + s_2 = -1 \\ \text{and} \quad & \lambda_1',\ \lambda_1'',\ \lambda_2',\ \lambda_2'',\ s_1,\ s_2 \geq 0 \end{aligned} \tag{2.12}$$

Alternatively Problem 2.13 shows the standard form using row variables.

$$\begin{aligned} \text{maximize} \quad & Z(\boldsymbol{\lambda}) = -2\lambda_1 - \lambda_2 \\ \text{subject to} \quad & r_1 = \lambda_1' - \lambda_1'' - 2\lambda_2' + 2\lambda_2'' + s_1 \qquad r_1 \leq -4 \\ & r_2 = -2\lambda_1' + 2\lambda_1'' - \lambda_2' + \lambda_2'' \qquad r_2 \leq -1 \\ \text{and} \quad & \lambda_1',\ \lambda_1'',\ \lambda_2',\ \lambda_2'' \geq 0 \end{aligned} \tag{2.13}$$

Matrix representation of an LP problem Given a matrix $A \in \mathbb{Q}^{m \times n}$ and vectors $\boldsymbol{b} \in \mathbb{Q}^m$, $\boldsymbol{o} \in \mathbb{Q}^n$, the LP problem can also be written in matrix form, which is shown in Problem 2.14.

$$\begin{aligned} \text{maximize} \quad & Z(\boldsymbol{\lambda}) = \boldsymbol{o}^\mathsf{T}\boldsymbol{\lambda} \\ \text{subject to} \quad & A\boldsymbol{\lambda} = \boldsymbol{b} \\ \text{and} \quad & \boldsymbol{\lambda} \geq 0 \end{aligned} \tag{2.14}$$

The LP problem is optimal at $\boldsymbol{\lambda}^*$, and the optimal value is Z^* such that:

$$Z^* = Z(\boldsymbol{\lambda}^*) \tag{2.15}$$

A solution is said to be *feasible* if it fulfills all the constraints; and a feasible solution is optimized when the variables appear in the objective function cannot increase/decrease anymore.

Example 2.6. Assume we have an LP problem shown in Problem 2.16.

$$\begin{aligned} \text{maximize} \quad & Z(\boldsymbol{\lambda}) = \lambda_1 - 3\lambda_2 + 2\lambda_3 - \lambda_4 \\ \text{subject to} \quad & 3\lambda_1 - \lambda_2 + \lambda_3 = 5 \\ & \lambda_1 - 3\lambda_2 + 3\lambda_4 = -3 \\ \text{and} \quad & \boldsymbol{\lambda} \geq 0 \end{aligned} \tag{2.16}$$

The objective function will be optimized at $\boldsymbol{\lambda}^* = (0,1,6,0)$, and $Z^* = 9$. The objective function is equivalent to $Z'(\boldsymbol{\lambda}) = -\frac{16}{3}\lambda_1 - 2\lambda_4 + 9$ by substituting $\lambda_2 = \frac{1}{3}\lambda_1 + \lambda_4 + 1,\ \lambda_3 = -\frac{8}{3}\lambda_1 + \lambda_4 + 6$.

As $\boldsymbol{\lambda} \geq 0$, λ_1, λ_4 cannot decrease anymore when they reach 0, and thus the value of $Z'(\boldsymbol{\lambda})$ cannot increase anymore, i.e, it reaches optimum.

2.4 Simplex algorithm

We use the *Simplex algorithm* [24, 25, 28] to solve the LP problems, so we explain it in details in this section. We first introduce some useful denominations, and then explain each step of the Simplex algorithm with an example.

There are *primal Simplex algorithm* and *dual Simplex algorithm*, but we will only present the former. The later is a process of applying the primal Simplex algorithm to the dual of the original problem. In the following context, when we mention Simplex, we mean the primal Simplex algorithm.

The standard Simplex algorithm contains two phases: the first phase looks for a *feasible* solution, and the second searches the *optimum*. The constraints of an LP problem define a convex polyhedron. The Simplex algorithm searches the solution $\boldsymbol{\lambda}^*$ among the vertices of this polyhedron. The first phase either starts from a vertex if the original problem is already feasible, or starts from exterior of the polyhedron otherwise. Once it reaches a vertex, the second phase starts.

Dictionaries An LP problem in the Simplex algorithm can be described by a *dictionary*, which is a system of linear equations. It substitutes variables by their equivalent counterpart expressed in terms of other variables. Consider Problem 2.10 which contains m constraints with n variables. If $n > m$, then m variables can be expressed by the left $n-m$ variables, as it is shown in Problem 2.17.

$$\begin{aligned} Z(\boldsymbol{\lambda}) &= \sum_{j=m+1}^{n} o'_j \lambda_j + c \\ \lambda_i &= b_i - \sum_{j=m+1}^{n} a'_{ij} \lambda_j \quad (i = i, \cdots, m) \end{aligned} \tag{2.17}$$

where c is a constant.

If we set the right-hand side variables to zero, the value of left-hand side variable λ_i will be b_i. As all the variables are non-negative, if all the b_i has non-negative value, we will obtain a feasible solution. The corresponding dictionary is called *feasible dictionary*.

Basic and non-basic variables The variables appear on the left-hand side of a dictionary are called *basic variables*, and that on the right-hand side are *nonbasic variables*. The set of basic variables is called a *basis* of the dictionary. The basic and non-basic variables form a partition of the variables, and the new objective function is obtained by substituting the basic variables with non-basic variables.

Example 2.7. Consider the Example 2.6. When the objective function reaches optimum, the basic variables are $\lambda_3,\ \lambda_4$, and nonbasic variables are $\lambda_1,\ \lambda_2$. We have $\lambda_3 = -3\lambda_1 + \lambda_2 + 5,\ \lambda_4 = -\frac{1}{3}\lambda_1 + \lambda_2 - 1$. The new obtained objective function is $Z'(\boldsymbol{\lambda}) = -\frac{14}{3}\lambda_1 - 2\lambda_2 + 11$.

Entering and leaving variables The process of constructing a new linear system by switching a basic variable with a nonbasic variable is called *pivoting*. The chosen basic variable is called *leaving variable* (as it will leave the basis), and the chosen nonbasic variable is called *entering variable*.

First phase of the Simplex algorithm

If the initial LP problem is feasible, the first phase will be skipped; otherwise an *auxiliary problem* will be solved for obtaining a feasible solution. Assuming Problem 2.10 is not feasible, its auxiliary problem is shown in Problem 2.18.

$$\begin{aligned} \text{maximize} \quad & -w \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij}\lambda_j - w = b_i \quad (\forall i \in \{1,\cdots,m\}) \\ \text{and} \quad & \lambda_j \geq 0, w \geq 0 \quad (\forall j \in \{1,\cdots,n\}) \end{aligned} \tag{2.18}$$

The auxiliary problem can always be transformed into feasibility by choosing w as the entering variable. The original LP problem is feasible if and only if the auxiliary problem has optimal solution and the optimum is $-w = 0$. In other words, if the auxiliary problem reaches optimum but w is in the basis, the original problem is still infeasible. The feasible solution will be obtained by eliminating w, which is the nonbasic variable in the final system. Let us illustrate the process with Example 2.8.

Example 2.8.

$$\begin{aligned} \text{maximize} \quad & Z(\boldsymbol{\lambda}) = 3\lambda_1 + \lambda_2 - 2\lambda_3 - 3\lambda_4 \\ \text{subject to} \quad & \lambda_1 + 3\lambda_2 - 2\lambda_3 = 4 \\ & 2\lambda_1 + \lambda_3 - 3\lambda_4 = -2 \\ \text{and} \quad & \lambda_j \geq 0 \quad (\forall j \in \{1,\cdots,4\}) \end{aligned}$$

First we substitute λ_1, λ_2 by λ_3, λ_4 and obtain:

$$\begin{aligned} Z(\boldsymbol{\lambda}) &= -\frac{8}{3}\lambda_3 + \lambda_4 - \frac{4}{3} \\ \lambda_1 &= -1 - \frac{1}{2}\lambda_3 + \frac{3}{2}\lambda_4 \\ \lambda_2 &= \frac{5}{3} + \frac{5}{6}\lambda_3 - \frac{1}{2}\lambda_4 \end{aligned}$$

By setting λ_3, λ_4 to zero, we can obtain $\lambda_1 = -1$, which means that the dictionary is infeasible. To obtain a feasible solution, we need to perform the first phase of the Simplex algorithm by solving the following auxiliary problem.

$$\begin{aligned} \text{maximize} \quad & Z(\boldsymbol{\lambda}, w) = -w \\ \text{subject to} \quad & \lambda_1 + \frac{1}{2}\lambda_3 - \frac{3}{2}\lambda_4 - w = -1 \\ & \lambda_2 - \frac{5}{6}\lambda_3 + \frac{1}{2}\lambda_4 - w = \frac{5}{3} \\ \text{and} \quad & \lambda_j \geq 0, w \geq 0 \quad (\forall j \in \{1, \cdots, 4\}) \end{aligned}$$

By substituting λ_1 and λ_2 we can obtain the corresponding dictionary:

$$\begin{aligned} Z(\boldsymbol{\lambda}, w) &= -w \\ \lambda_1 &= -1 - \frac{1}{2}\lambda_3 + \frac{3}{2}\lambda_4 + w \\ \lambda_2 &= \frac{5}{3} + \frac{5}{6}\lambda_3 - \frac{1}{2}\lambda_4 + w \end{aligned} \tag{2.19}$$

To obtain a feasible dictionary, in the first pivoting we need to choose w as the entering variable and λ_1 as the leaving variable. Then we will obtain the dictionary as follows:

$$\begin{aligned} Z(\boldsymbol{\lambda}, w) &= -1 - \lambda_1 - \frac{1}{2}\lambda_3 + \frac{3}{2}\lambda_4 \\ w &= 1 + \lambda_1 + \frac{1}{2}\lambda_3 - \frac{3}{2}\lambda_4 \\ \lambda_2 &= \frac{8}{3} + \lambda_1 + \frac{4}{3}\lambda_3 - 2\lambda_4 \end{aligned} \tag{2.20}$$

Then we choose λ_4 to enter the basis, and w to leave. Finally we obtain a feasible dictionary:

$$\begin{aligned} Z(\boldsymbol{\lambda}, w) &= -w \\ \lambda_4 &= \frac{2}{3} + \frac{2}{3}\lambda_1 + \frac{1}{3}\lambda_3 - \frac{2}{3}w \\ \lambda_2 &= \frac{4}{3} - \frac{1}{3}\lambda_1 + \frac{2}{3}\lambda_3 + \frac{4}{3}w \end{aligned} \tag{2.21}$$

As $w \geq 0$, the optimal function reaches the optimum 0. By eliminating w, we obtain the feasible dictionary of the original problem, in which λ_2 and λ_4 are basic variables.

$$\begin{aligned} Z(\boldsymbol{\lambda}) &= \frac{2}{3}\lambda_1 - \frac{7}{3}\lambda_3 - \frac{2}{3} \\ \lambda_4 &= \frac{2}{3} + \frac{2}{3}\lambda_1 + \frac{1}{3}\lambda_3 \\ \lambda_2 &= \frac{4}{3} - \frac{1}{3}\lambda_1 + \frac{2}{3}\lambda_3 \end{aligned} \tag{2.22}$$

Second phase of the Simplex algorithm

The second phase of the Simplex will find the optimal solution or report that the problem does not have optimum, meaning that the objective function is unbounded.

In the standard form, the objective function needs to be maximized, and all the variables must satisfy $\lambda_i \geq 0$. Thus when all the coefficients of the nonbasic variables are non-positive, the objective function reaches optimum; otherwise the objective function can be increased. Then one nonbasic variable whose coefficient is positive should enter the basis. Given the entering variable λ_j, the leaving variable is $\lambda_k = b'_i - \sum_j a'_{ij}\lambda_j$, which minimizes the ratio $\frac{b'_i}{a'_{ij}}$ with $a'_{ij} > 0$, where $j \neq k$ and λ_j are nonbasic variables.

Example 2.9. Let us consider Problem 2.22 of Example 2.8. We obtained a feasible dictionary, but the variable λ_1 in the objective function has a positive coefficient $\frac{2}{3}$, meaning that the objective function does not reach the optimum. It can be increased by augmenting the value of λ_1. Thus we choose the variable whose coefficient is positive, i.e. λ_1, as the entering variable in the next pivoting. Recall that the constraints are in the form $\lambda_k = b'_i - \sum_j a'_{ij}\lambda_j$. The only choice of leaving variable is λ_2, as in the first constraint $\lambda_4 = \frac{2}{3} - (-\frac{2}{3}\lambda_1 - \frac{1}{3}\lambda_3)$ the coefficient of λ_1 is negative. By doing the pivoting, we obtain the final dictionary shown in Problem 2.23, in which the objective function cannot be furthermore increased since the maximal value is only obtained by setting the nonbasic variables λ_2 and λ_3 to 0, and any other choice would decrease the value of $Z(\boldsymbol{\lambda})$.

$$\begin{aligned} Z(\boldsymbol{\lambda}) &= -2\lambda_2 - \lambda_3 + 2 \\ \lambda_4 &= \frac{10}{3} - 2\lambda_2 + \frac{5}{3}\lambda_3 \\ \lambda_1 &= 4 - 3\lambda_2 + 2\lambda_3 \end{aligned} \tag{2.23}$$

All the nonbasic variables are set to their lower bound 0, and we obtain the optimal vertex $(4,0,0,\frac{10}{3})$ with the optimal value 2.

Simplex tableau

In the Simplex algorithm, the dictionary is normally represented in *Simplex tableau*. In the follow sections, we will always use the Simplex tableau to illustrate a dictionary. The Simplex tableau of Problem 2.10 is shown in Table 2.3, where o_i are the coefficients of the objective function.

	λ_1	$\cdots$	λ_n	= constant
$\ldots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
row i	a_{i1}	$\cdots$	a_{in}	b_i
$\ldots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
objective	o_1	$\cdots$	o_n	$-Z^*$

Table 2.3 – Simplex tableau.

The last dictionary of Example 2.8 which gives the optimum can be written in the Simplex tableau as shown in Table 2.4.

	λ_1	λ_2	λ_3	λ_4	= constant
row 1	0	2	$-\frac{5}{3}$	1	$\frac{10}{3}$
row 2	1	3	-2	0	4
objective	0	-2	-1	0	-2

Table 2.4 – Simplex tableau example.

In the Simplex tableau, the objective function is written in the form $Z(\boldsymbol{\lambda}) - Z^* = 0$. In our example, the objective function row in Table 2.4 is $-2\lambda_2 - \lambda_3 - Z^* = -2$, so we have $Z^* = 2$ by setting $\lambda_2,\ \lambda_3$ to 0.

2.5 Parametric linear programming

In this section we will catch a glimpse of the *parametric linear programming (PLP)*, using which we compute the polyhedral projection and convex hull. We will see the definition of the PLP, and also the processes of solving a PLP problem.

2.5.1 Definition of PLP

A PLP problem is a linear optimization problem on the variables $\boldsymbol{\lambda} = [\lambda_1, \cdots, \lambda_m]^\mathsf{T}$ whose objective function contains parameters $\boldsymbol{x} = [x_1, \cdots, x_n]^\mathsf{T}$. Problem 2.24 shows the standard form of a PLP problem[1].

[1] There is another form of PLP, in which the parameters appear on the right-hand side of the constraints.

$$\begin{aligned} \text{maximize} \quad & Z(\boldsymbol{\lambda},\boldsymbol{x}) = \sum_{i=1}^{m}(\sum_{j=1}^{n} o_{ij}x_j - c_i)\lambda_i \\ \text{subject to} \quad & \sum_{j=1}^{m} a_{ij}\lambda_j = b_i \quad \forall i \in \{1,\cdots k\} \\ \text{and} \quad & \boldsymbol{\lambda} \geq 0 \end{aligned} \tag{2.24}$$

The problem can be reformulated in a more concise way by introducing an extra dimension to the vector of parameters for representing the constant part c_i. Given $O \in \mathbb{Q}^{(n+1)\times m}$, $A \in \mathbb{Q}^{k\times m}$, $\boldsymbol{b} \in \mathbb{Q}^k$ and $\boldsymbol{x} = [x_1,\cdots,x_n,1]^\mathsf{T}$, Problem 2.25 shows the PLP in matrix form.

$$\begin{aligned} \text{maximize} \quad & Z(\boldsymbol{\lambda},\boldsymbol{x}) = \boldsymbol{x}^\mathsf{T} O\boldsymbol{\lambda} \\ \text{subject to} \quad & A\boldsymbol{\lambda} = \boldsymbol{b} \\ \text{and} \quad & \boldsymbol{\lambda} \geq 0 \end{aligned} \tag{2.25}$$

2.5.2 Methods to solve PLP problems

The PLP problems can be solved directly by Simplex algorithm, or by solving a set of LP problems which are obtained by instantiating the parameters $\boldsymbol{x}$. We will use both methods in our algorithm of PLP solver. The optimal solution of a PLP problem is a set of couples $(\mathcal{R}_i, Z_i^*(\boldsymbol{x}))$, where $\mathcal{R}_i$ is the region of the space of parameters $\boldsymbol{x}$, in which the basis does not change and $Z_i^*(\boldsymbol{x})$ is the optimal function corresponding to $\mathcal{R}_i$, meaning that any instantiation point $\boldsymbol{x}$ in $\mathcal{R}_i$ leads to the optimal function $Z_i^*(\boldsymbol{x})$. The region is obtained from the sign condition from the objective function, i.e., all the coefficients should be non-positive (or non-negative) if the objective is to be maximized (or minimized). An example of the Simplex tableau of PLP is shown in Table 2.5. The coefficients of the nonbasic variables in the objective function are linear expressions, and that of the basic variables are 0. Assuming we are computing the maximization of the objective function, the coefficients should be non-positive. By this condition we obtain the region $\bigwedge_{i\in\{1,\cdots,q\}} \sum_{k=1}^{n} o_{ik}x_k + c_i \leq 0$.

Solving PLP via Simplex algorithm First we apply the first phase of the Simplex algorithm (Section 2.4) for obtaining a feasible solution with the basic and nonbasic variables. Then we express the original objective function of the PLP problem in terms of the nonbasic variables for obtaining an optimal function with the corresponding region. The next step is to pivot the Simplex tableau for obtaining other optimal functions. Considering the example in Table 2.5, we select one from the nonbasic variables, say λ_q, to enter the basis. The leaving variable is λ_r, which makes the ratio $\frac{b_j}{a_{jq}}$ be minimum for all j, where $a_{jq} > 0$. If $\forall j, a_{jq} \leq 0$, then λ_q cannot enter the basis. We will take an example to explain this method in Example 2.10.

	non-basic variables				basic variables				= constants
	λ_1	$\cdots$	λ_q	$\cdots$	$\cdots$	λ_r	$\cdots$	λ_m	
$\vdots$	.....								
row j	.....		a_{jq}	$\cdots$	$\cdots$	1	$\cdots$	0	b_j
$\vdots$	.....								
objective	$\sum_{k=1}^{n} o_{1k}x_k + c_1$	$\cdots$	$\sum_{k=1}^{n} o_{qk}x_k + c_q$	$\cdots$	$\cdots$	0	$\cdots$	0	$-Z^*(\boldsymbol{x})$

Table 2.5 – Simplex tableau.

Solving PLP via LP problems We randomly select an instantiation point in the parameter space, and substitute the parameters $\boldsymbol{x}$ by this instantiation point for obtaining an LP problem. If the LP is optimal at the vertex $\boldsymbol{\lambda}^*$, the PLP will also reach optimum at the same vertex $\boldsymbol{\lambda}^*$. By solving this LP problem we can obtain an optimal solution and the corresponding basic and nonbasic variables, if it is not unbounded. By expressing the objective function in terms of the nonbasic variables, we obtain an optimal solution $(\mathcal{R}_i, Z_i^*(\boldsymbol{x}))$ of the PLP. Then we select another instantiation point outside $\mathcal{R}_i$, and repeat the same steps for obtaining other optimal solutions. This method will be shown on an example in Example 2.11.

Example 2.10. Considering the same constraints with Example 2.6, the PLP problem in Problem 2.26 can be obtained by replacing the objective vector $\boldsymbol{o}^\mathsf{T}$ with $\boldsymbol{x}^\mathsf{T}O$, where the matrix $O = \begin{bmatrix} -1 & 2 & 1 & 3 \\ 2 & 1 & 3 & -2 \\ 0 & -1 & 3 & 2 \end{bmatrix}$, and $\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & 1 \end{bmatrix}^\mathsf{T}$.

$$\begin{aligned} \text{maximize} \quad & Z(\boldsymbol{\lambda}) = (-x_1 + 2x_2)\lambda_1 + (2x_1 + x_2 - 1)\lambda_2 + (x_1 + 3x_2 + 3)\lambda_3 + (3x_1 - 2x_2 + 2)\lambda_4 \\ \text{subject to} \quad & 3\lambda_1 - \lambda_2 + \lambda_3 = 5 \\ & \lambda_1 - 3\lambda_2 + 3\lambda_4 = -3 \\ \text{and} \quad & \boldsymbol{\lambda} \geq 0 \end{aligned} \tag{2.26}$$

A feasible basis $\lambda_1,\ \lambda_2$ can be obtained by solving the auxiliary problem shown in Problem 2.27. Then we substituting $\lambda_1,\ \lambda_2$ by $\lambda_3,\ \lambda_4$ in the original PLP problem, and the corresponding Simplex tableau is shown in Table 2.6. We obtain the optimal solution $Z_1^*(\boldsymbol{x}) = \frac{5}{4}x_1 + \frac{25}{4} - \frac{7}{4}$ and $\mathcal{R}_1 = \{\frac{9}{8}x_1 + \frac{17}{8}x_2 + \frac{25}{8} \leq 0,\ \frac{39}{8}x_1 - \frac{1}{8}x_2 + \frac{7}{8} \leq 0\}$.

$$\begin{aligned} \text{maximize} \quad & -w \\ \text{subject to} \quad & 3\lambda_1 - \lambda_2 + \lambda_3 - w = 5 \\ & \lambda_1 - 3\lambda_2 + 3\lambda_4 - w = -3 \\ \text{and} \quad & \boldsymbol{\lambda} \geq 0, w \geq 0 \end{aligned} \tag{2.27}$$

In the next pivoting, we choose λ_3 as the entering variable. We can see that the coefficients of λ_3 in row 1 and row 2 are both positive, so we need to compare their ratios of the constant to the coefficient. As $\frac{7}{4}/\frac{1}{8} > \frac{9}{4}/\frac{3}{8}$, we choose λ_1 as the leaving variable. Then we obtain the new basis $\lambda_2,\ \lambda_3$, and the Simplex tableau is shown in Table 2.7. We obtain the optimal solution $Z_2^*(\boldsymbol{x}) = 8x_1 + 19x_2 + 17$ with the region $\mathcal{R}_2 = \{-3x_1 - \frac{17}{3}x_2 - \frac{25}{3} \leq 0,\ 6x_1 + 2x_2 + 4 \leq 0\}$.

As the coefficients of λ_4 in all rows are negative, λ_4 cannot enter the basis. Now all the possible bases have been found, and the algorithm terminates.

	λ_1	λ_2	λ_3	λ_4	= constant
row 1	0	1	$\frac{1}{8}$	$-\frac{9}{8}$	$\frac{7}{4}$
row 2	1	0	$\frac{3}{8}$	$-\frac{3}{8}$	$\frac{9}{4}$
objective	0	0	$\frac{9}{8}x_1 + \frac{17}{8}x_2 + \frac{25}{8}$	$\frac{39}{8}x_1 - \frac{1}{8}x_2 + \frac{7}{8}$	$-\frac{5}{4}x_1 - \frac{25}{4}x_2 + \frac{7}{4}$

Table 2.6 – Simplex tableau example.

	λ_1	λ_2	λ_3	λ_4	= constant
row 1	$-\frac{1}{3}$	1	0	-1	1
row 2	$\frac{8}{3}$	0	1	-1	6
objective	$-3x_1 - \frac{17}{3}x_2 - \frac{25}{3}$	0	0	$6x_1 + 2x_2 + 4$	$-8x_1 - 19x_2 - 17$

Table 2.7 – Simplex tableau example.

Example 2.11. Considering the same example with Problem 2.26, By substituting the parameters $\boldsymbol{x}$ with the point $(-1,0)$, we obtain the LP problem in Example 2.6. As the LP problem is optimized with the basic variables λ_3 and λ_4, the PLP problem also reaches optimum with the same basis. By substituting λ_2 and λ_3 we obtain the final objective function $Z'(\boldsymbol{\lambda}, \boldsymbol{x}) = (-3x_1 - \frac{17}{3}x_2 - \frac{25}{3})\lambda_1 + (6x_1 + 2x_2 + 4)\lambda_4 + (8x_1 + 19x_2 + 17)$, and the optimal solution is $(\mathcal{R}_2, Z_2^*(\boldsymbol{x}))$.

> The set of LP problems which are obtained by instantiating the parameters $\boldsymbol{x}$ with any point in $\mathcal{R}_1$ will reach optimum with the same basis λ_3, λ_4, and thus the PLP will obtain the same optimal function. To obtain other optimal functions, we need to instantiate $\boldsymbol{x}$ with the points outside $\mathcal{R}_1$. With the instantiation point $(-2,-2)$, we obtain the optimal solution $(\mathcal{R}_1, Z_1^*(\boldsymbol{x}))$. Until now all the optimal solutions have been found, as any other point outside $\mathcal{R}_1$ and $\mathcal{R}_2$ will lead to unbounded LP problems.

Summary of the chapter

We presented some fundamental knowledge which is useful for the explanation of our efficient PLP solver in the following chapters. We saw the convex polyhedra and its representations. In our approach we use the constraint-only representation. We introduced some operators in the polyhedra domain. Our approach aimed at computing the polyhedral projection and convex hull. As the minimization operator is required in our PLP solver, we also provided an efficient minimization algorithm based on the work in [29].

We introduced the floating point arithmetic and rounding errors. In the paper we use Equation 2.6 and Equation 2.9 to compute the thresholds which are used for comparing floating point numbers.

We illustrated the processes of solving an LP problem using the Simplex algorithm. The computation processes can be shown in a dictionary or in the Simplex tableau form. In the following context we mainly use the latter. The methods to solve a PLP problem are also been presented. The solution of a PLP is a set of tuples $(\mathcal{R}_i, Z_i^*(\boldsymbol{x}))$, where $Z_i^*(\boldsymbol{x})$ is an optimal function and $\mathcal{R}_i$ is the corresponding region. Please be note that for two different regions $\mathcal{R}_i$ and $\mathcal{R}_j$, their optimal functions $Z_i^*(\boldsymbol{x})$ and $Z_j^*(\boldsymbol{x})$ could be the same. This point will be explained into details in Chapter 4.

Chapter 3

Sequential Parametric Linear Programming Solver

In this chapter we will present the overview of the sequential algorithm of the PLP solver. This work was presented in our prior paper [30].

We provide a flowchart that illustrates the processes of computing the projection (or convex hull) using our PLP solver. We will see that the *minimization* operator is required at several steps in the PLP solver, so we will present the minimization algorithm before explaining the algorithm of the PLP solver. We adapted the algorithm of raytracing minimization [29] to our floating point arithmetic.

Our PLP solver is based on the work of Maréchal et al. in [20]. Different from their work, we mainly used floating point arithmetic, and the rational result will be constructed. We aimed to obtain the rational result using floating point computations without impact on soundness and precision. Soundness means that we need to obtain an over-approximate polyhedron of the correct one; precision means that the resulting polyhedron should be the same as that obtained by rational computation using the same algorithm.

Due to the imprecision of floating point computations, the result may be incorrect and the algorithm may be not able to terminate. To avoid these problems, we use floating point thresholds to test the result and use rational numbers to check and recompute the result when it is necessary.

The whole algorithm of the PLP solver is divided into several steps, and each step will be explained into details. During the explanation of the algorithm, our focus will be on the cooperation of rational and floating point arithmetic, and some specifics of floating point computations.

3.1 Flowchart of the algorithm

The Figure 3.1 shows the flow chart of our algorithm. The rectangles are processes and diamonds are decisions. The processes/decisions colored in orange are computed in floating-point arithmetic, and those in blue use rational numbers. The dotted red frames show cases that rarely happen, which means that most computations in our approach use floating-point numbers.

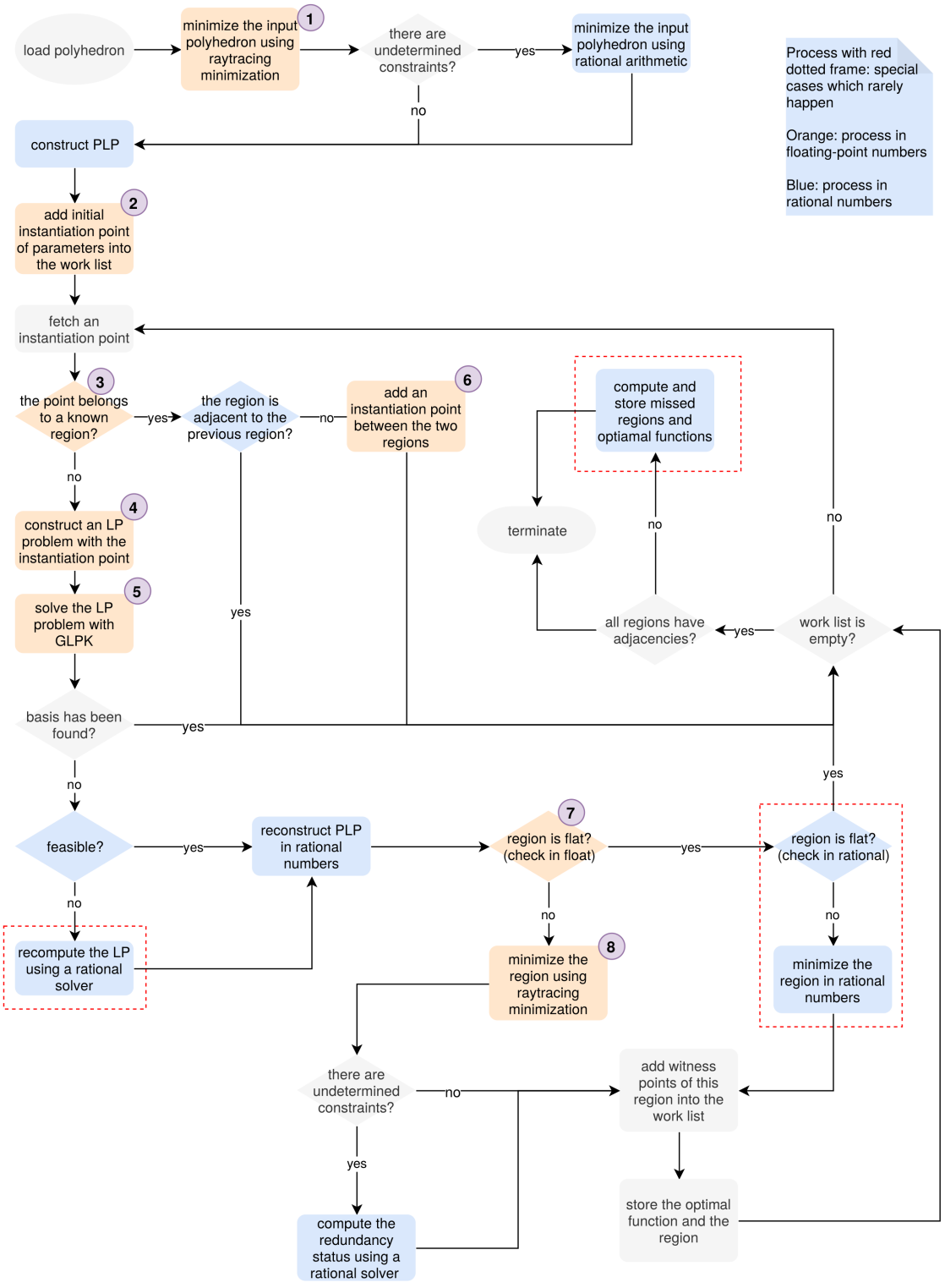


Figure 3.1 – Flowchart of our PLP procedure

3.2 Minimization

In the PLP algorithm, the minimization is required to minimize the input polyhedron and the obtained regions, so we will present the algorithm in this section. We will present two methods of minimization, and both of them will be used in our PLP solver.

3.2.1 Redundant Constraints

In our computation, there are several steps at which *redundant constraints* must be removed by *minimization* of the polyhedron.

> **Definition 3.1** (Redundant constraints)**.** A constraint is said to be redundant if it can be removed without changing the shape of the polyhedron.

> **Example 3.1.** Considering the constraints $\{C_1 : -x_1 + 2x_2 \geq 2,\ C_2 : 2x_1 - x_2 \geq 1,\ C_3 : -x_1 - x_2 \geq -8,\ C_4 : 2x_1 + 4x_2 \geq 7\}$ in Figure 3.2, the red constraint $C_4 : 2x_1 + 4x_2 \geq 7$ is redundant.

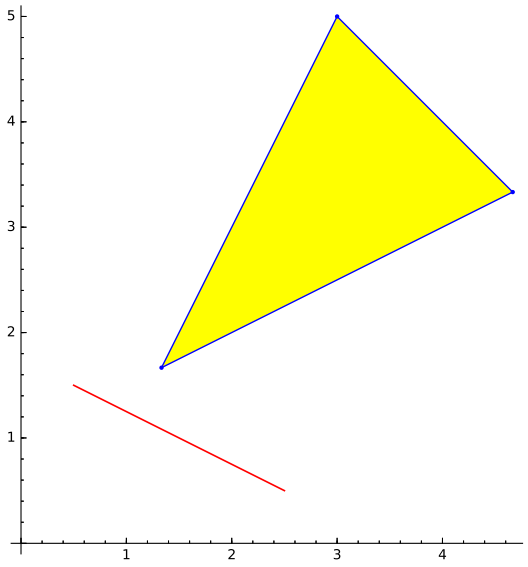


Figure 3.2 – Redundant constraint

3.2.2 Minimization using Farkas' lemma

The redundancy of a constraint can be checked by Farkas' lemma (Theorem 3.1) : a constraint $\mathcal{C}_i$ is redundant if and only if it is a linear combination of other constraints. In other words, $\mathcal{C}_i$: $\boldsymbol{a}_i\boldsymbol{x} \geq b_i$ is redundant if and only if it fulfills Equation 3.1.

$$\exists \boldsymbol{y} = [y_1, \ldots, y_m, y_0] \geq 0,\ \boldsymbol{a}_i\boldsymbol{x} - b_i = \sum_{j=1, j\neq i}^{m} y_j(\boldsymbol{a}_j\boldsymbol{x} - b_j) + y_0 \tag{3.1}$$

Equation 3.1 can be expressed in another way, which is shown in Equation 3.2. By writing it into matrix form, we have Equation 3.3. If the system of linear inequalities has a solution, the first statement of Farkas' lemma is true; otherwise the second statement is true.

$$\begin{aligned} &\exists \boldsymbol{y} = [y_1, \ldots, y_m, y_0] \geq 0 \\ &\sum_{j=1, j\neq i}^{m} a_{jk} y_j = a_{ik} \quad \forall k = \{1, \cdots, n\} \\ &\sum_{j=1, j\neq i}^{m} b_j y_j + y_0 = b_i \end{aligned} \tag{3.2}$$

$$\exists \boldsymbol{y} = [y_1, \ldots, y_m, y_0] \geq 0,\ M\boldsymbol{y} = \boldsymbol{d},\ \text{where } M = \begin{bmatrix} a_{11} & \cdots & a_{m1} & 0 \\ & \vdots & & \\ a_{1n} & \cdots & a_{mn} & 0 \\ b_1 & \cdots & b_m & 1 \end{bmatrix},\ \boldsymbol{d} = [a_{i1}, \cdots, a_{in}, b_i]^{\mathsf{T}} \tag{3.3}$$

Theorem 3.1 (Farkas' lemma)**.** Let $A \in \mathbb{R}^{m\times n}$ and $\boldsymbol{b} \in \mathbb{R}^m$. Then exactly one of the following two statements is true:

- There exists an $\boldsymbol{y} \in \mathbb{R}^n$ such that $A\boldsymbol{y} = \boldsymbol{b}$ and $\boldsymbol{y} \geq 0$.
- There exists a $\boldsymbol{y}' \in \mathbb{R}^m$ such that $A^{\mathsf{T}}\boldsymbol{y}' \geq 0$ and $\boldsymbol{b}^{\mathsf{T}}\boldsymbol{y}' < 0$.

Example 3.2. Let use consider the same example as Example 3.1. To determine if the constraint $\mathcal{C}_4$ of Example 3.1 is redundant with respect to $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$, we need to solve the satisfiability problem $\exists y_1, y_2, y_3, y_0 \geq 0$ such that $\boldsymbol{a}_4\boldsymbol{x} - b_4 = y_1(\boldsymbol{a}_1\boldsymbol{x} - b_1) + y_2(\boldsymbol{a}_2\boldsymbol{x} - b_2) + y_3(\boldsymbol{a}_3\boldsymbol{x} - b_3) + y_0$, i.e., Problem 3.4.

$$\begin{aligned} &-y_1 + 2y_2 - y_3 = 2 \\ &2y_1 - y_2 - y_3 = 4 \\ &-2y_1 - y_2 + 8y_3 + y_0 = -7 \\ &y_0, y_1, y_2, y_3 \geq 0 \end{aligned} \tag{3.4}$$

We obtain $\boldsymbol{y} = [y_1, y_2, y_3, y_0]^{\mathsf{T}} = [\frac{10}{3}, \frac{8}{3}, 0, \frac{7}{3}]^{\mathsf{T}}$, such that $2x_1 + 4x_2 - 7 = \frac{10}{3}(-x_1 + 2x_2 - 2) + \frac{8}{3}(2x_1 - x_2 - 1) + \frac{7}{3}$, meaning that C_4 is redundant. In this case the first statement of Farkas' lemma is true.

On the contrary, C_1 is irredundant, as the second statement of Farkas' lemma is true. By solving Problem 3.5, we can find $\boldsymbol{y}' = [4,0,1]$, and $A^T\boldsymbol{y}' = [7,4,1,1]^\mathsf{T} > 0$, $\boldsymbol{b}^\mathsf{T}\boldsymbol{y}' = -6 < 0$.

$$\begin{aligned} &2y_1 - y_2 + 2y_3 = -1 \\ &-y_1 - y_2 + 4y_3 = 2 \\ &-y_1 + 8y_2 - 7y_3 + y_0 = -2 \\ &y_0, y_1, y_2, y_3 \geq 0 \end{aligned} \tag{3.5}$$

3.2.3 Raytracing method for minimization

When there are much more irredundant constraints than redundant ones, using Farkas' lemma is not efficient. To find out the irredundant constraints more efficiently, Maréchal et al. presented a novel method using raytracing to minimize polyhedra [29]. We improved this approach by using floating point arithmetic instead of rational computations. The use of floating-point numbers here will not cause a soundness problem for static analysis, because in the worst case we eliminate constraints that should not be removed. That results in a larger polyhedron which includes the correct one, i.e., the abstract polyhedron is over approximated. At several steps we need to check the results of floating point computation for guaranteeing the precision and soundness. We first review the minimization algorithm, and then present adaptations we made for floating point computation.

3.2.3.1 Overview of the raytracing minimization

There are two phases in raytracing minimization. The first phase aims at finding as many irredundant constraints as possible. Rays are launched from a point inside the polyhedron toward each boundary. The constraint whose boundary is firstly hit by a ray is irredundant. The remaining constraints whose redundancy status is not solved by the first phase will be determined by the second phase: if we can find an *irredundancy witness* point, then the corresponding constraint is irredundant.

Definition 3.2 (Irredundancy Witness)**.** An irredundancy witness of a constraint $\mathcal{C}_i$ is a point that violates $\mathcal{C}_i$ but satisfies the other constraints.

A ray $\boldsymbol{r}_i$ was launched in the first phase from the interior point toward the boundary of $\mathcal{C}_i : \boldsymbol{a}_i\boldsymbol{x} \leq b_i$ in the direction $\boldsymbol{a}_i$. To determine the redundancy status of $\mathcal{C}_i$, we need to solve the problem $\exists \boldsymbol{x},\ \boldsymbol{a}_i\boldsymbol{x} > b_i, \bigwedge_{j\neq i} \boldsymbol{a}_j\boldsymbol{x} \leq b_j$, where the constraints $\mathcal{C}_j : \boldsymbol{a}_j\boldsymbol{x} \leq b_j$ are the set of constraints whose boundaries hit by $\boldsymbol{r_i}$ before the boundary of $\mathcal{C}_i$ is reached. If the satisfiability problem is infeasible, then $\mathcal{C}_i$ is redundant as the irredundancy witness point cannot be found; otherwise a point $\boldsymbol{p}$ will be obtained, but $\boldsymbol{p}$ is not necessarily an witness of the irredundancy of $\mathcal{C}_i$ (see Figure 3.4). Another ray $\boldsymbol{r}'_i$ will be launched from the interior point to the point $\boldsymbol{p}$. Then $\mathcal{C}_i$ will be first hit by $\boldsymbol{r}'_i$ if it is irredundant, and $\boldsymbol{p}$ is its irredundancy witness; or another point $\boldsymbol{p}'$ will be obtained by solving $\boldsymbol{a}_i\boldsymbol{x} > b_i, \bigwedge_{j\neq i} \boldsymbol{a}_j\boldsymbol{x} \leq b_j, \bigwedge_{k\neq i} \boldsymbol{a}_k\boldsymbol{x} \leq b_k$, where the boundaries of $\mathcal{C}_k : \boldsymbol{a}_k\boldsymbol{x} \leq b_k$ are first hit by $\boldsymbol{r}'_i$. By adding constraints incrementally and solving a set

of satisfiability problems, either the boundary of $\mathcal{C}_i$ will be first hit by a ray if it is irredundant, or an infeasible solution will be obtained otherwise.

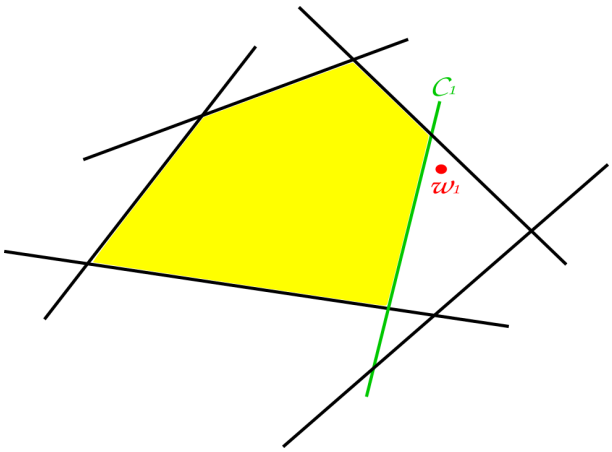


Figure 3.3 – The point w_1 is a *witness of the irredundancy* of $\mathcal{C}_1$: It satisfies all constraints but $\mathcal{C}_1$. Therefore, removing $\mathcal{C}_1$ would change the polyhedron because w_1 would then belong to it.

> **Example 3.3.** Figure 3.4 shows an example of checking the redundancy status of the constraint $\mathcal{C}_1$. In Figure 3.4(a) a ray is launched from the interior point $\boldsymbol{p}$ to $\mathcal{C}_1$, and hits constraints $\mathcal{C}_2$ and $\mathcal{C}_3$ before reaching $\mathcal{C}_1$. In Figure 3.4(b) a point $\boldsymbol{p}'$ is found, which satisfies $\mathcal{C}_2$ and $\mathcal{C}_3$ but violates $\mathcal{C}_1$. The ray launched from $\boldsymbol{p}$ to $\boldsymbol{p}'$ hits $\mathcal{C}_1$ first, which proved that $\mathcal{C}_1$ is irredundant.

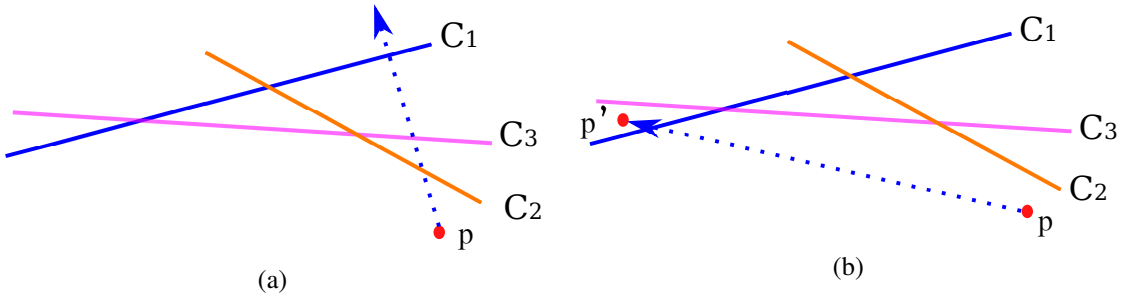


Figure 3.4 – The process of checking the redundancy status of the constraint $\mathcal{C}_1$ using the second phase of raytracing minimization.

3.2.3.2 Adaptations for floating point computation

Now that we review the raytracing minimization algorithm of [29], let use detail the technicalities required for adapting the algorithm to floating point arithmetic.

Solving satisfiability and optimization problems We use GLPK[1] (GNU Linear Programming Kit) to solve the optimization problems in our approach. For solving the satisfiability problems, we just set the

[1] https://www.gnu.org/software/glpk/

objective function to 0. GLPK provides several routines for solving the LP problems, which are written in ANSI C, but we only used the Simplex algorithm.

Launching rays from a central point The raytracing algorithm uses an interior point from which we launch rays in directions perpendicular to the boundaries of the constraints. If we simply ask for any point inside a bounded polyhedron, we can construct an LP problem to compute a point that satisfies the constraints and does not locate on the boundaries, i.e., satisfies $\bigwedge_i \boldsymbol{a}_i\boldsymbol{x} < b_i$ with strict inequalities. Since GLPK does not support inequalities, one could be tempted to shift the non-strict constraints a little to simulate strict ones. Considering the strict constraint $\mathcal{C}_i : \boldsymbol{a}_i\boldsymbol{x} < b_i$, we can obtain a non-strict constraint $\mathcal{C}_i'$ by shifting it:

$$\mathcal{C}_i' : \boldsymbol{a}_i\boldsymbol{x} \leq b_i - \|\boldsymbol{a}\|\delta \tag{3.6}$$

where δ is a positive constant.

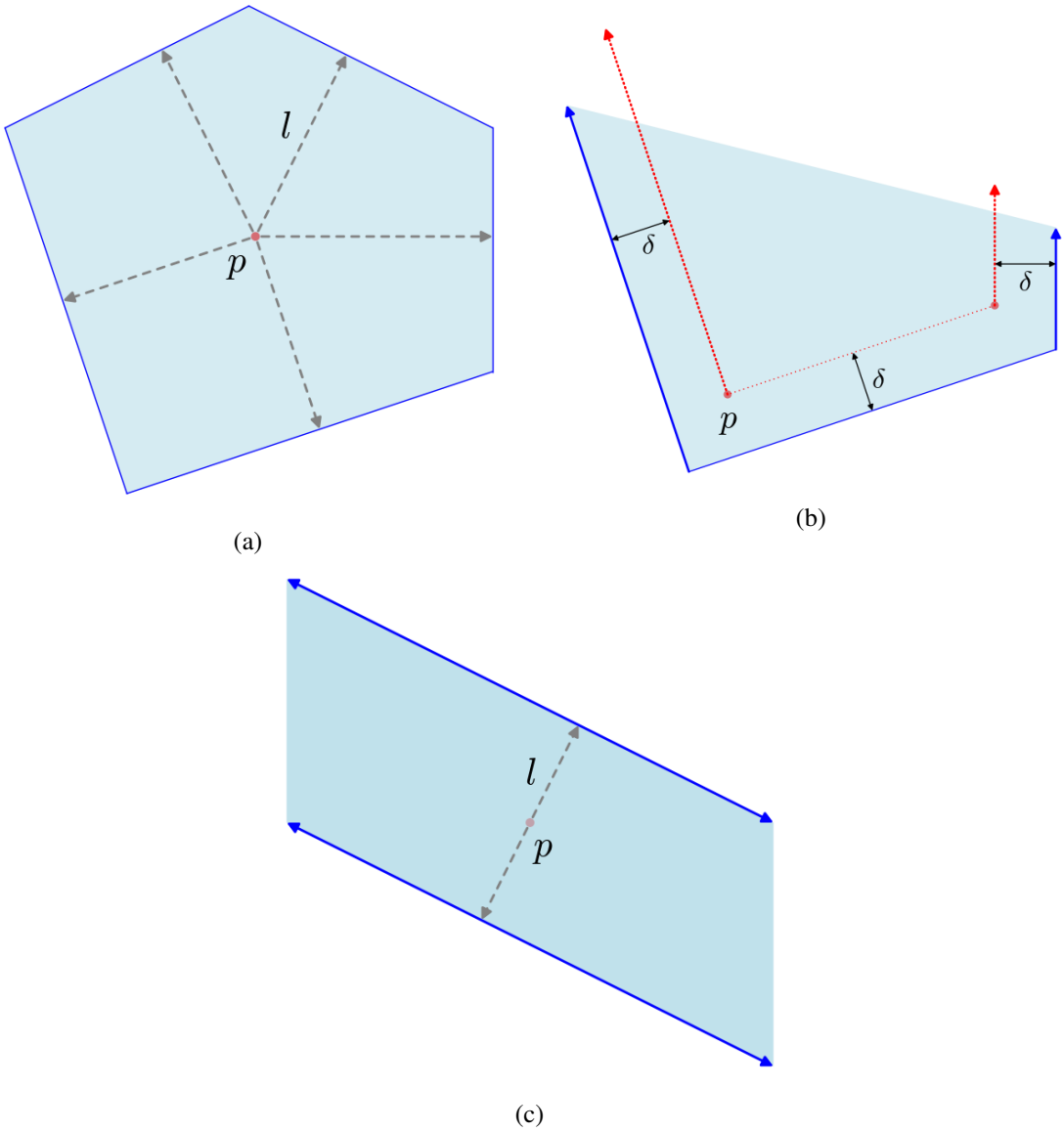
However choosing an appropriate quantity for δ is difficult: if it is small we will obtain a point which is too close to the boundaries; if it is large, we may end up with an infeasible problem. Hence we prefer to compute a point located in a "central position" of the polyhedron, which is far away from the boundaries.

Considering a point $\boldsymbol{p}$ which is inside a polyhedron $\mathcal{P}$, $d_i = \frac{b_i - \boldsymbol{a}_i\boldsymbol{p}}{\|\boldsymbol{a}_i\|}$ is the distance from $\boldsymbol{p}$ to the boundary of constraint $\mathcal{C}_i$ of $\mathcal{P}$, where $\|\boldsymbol{a}_i\|$ is the Euclidean norm of the vector $\boldsymbol{a}_i$. We aim at pushing the point $\boldsymbol{p}$ to a position which is as far as possible from the boundaries. Problem 3.7 shows the LP for finding the central point by adjusting the point $\boldsymbol{p}$ and the variable l. The variables d_i only appear for the sake of readability.

$$\begin{aligned} &\text{maximize} && l \\ &\text{subject to} && d_i = \frac{b_i - \boldsymbol{a}_i\boldsymbol{p}}{\|\boldsymbol{a}_i\|},\ l \leq d_i\ (\forall \mathcal{C}_i \in \mathcal{P}) \end{aligned} \tag{3.7}$$

When the LP is optimized, we get an optimal vertex $[\boldsymbol{p}\ l]$, from which we can obtain the expected central point $\boldsymbol{p}$ (Figure 3.5(a)). For unbounded polyhedra, there is no central point, as the polyhedron does not have a "central" position. In this case it is safe to apply the shifting strategy presented in Problem 3.6: no matter how large δ is, we can always obtain an interior point. The Simplex algorithm will obtain a satisfiability point at one vertex of the shifted polyhedron. As it is shown in Figure 3.5(b), one point of the two will be found as the interior point. One exception is the case where the unbounded polyhedron contains parallel constraints (Figure 3.5(c)). In this case if we shift the constraints with a large threshold, no point could be found. But the LP in Problem 3.7 is suitable for this special situation.

We do not need to test the shape of the polyhedron for choosing the method to find the interior point. We always try to find a central point using Problem 3.7. If the optimization problem is unbounded, we then try to obtain an interior point by shifting constraints.



(a) (b) (c)

Figure 3.5 – How to obtain an interior point in different cases: In Figure (a) and Figure (c) the central points are found; for the case of Figure (b) the constraints are shifted for obtaining an interior point.

Testing emptiness of the interior The polyhedron can also have an empty interior, and thus we cannot find any interior point.

> **Theorem 3.2.** The polyhedron $\mathcal{P}$ has empty interior if and only if the optimal value l in Problem 3.7 is zero.

Proof. If the constraints of the polyhedron are unsatisfiable, we cannot find a point $\boldsymbol{p}$ inside the polyhedron. This case can be easily detected. We here consider the problem of detecting the "flat" polyhedron whose interior is empty. It means that the inequality constraints entail an equality. Consider

a satisfiable polyhedron $\bigwedge_i \boldsymbol{a}_i\boldsymbol{x} \leq b$. There exists at least one point $\boldsymbol{p}$ that satisfies this polyhedron, and we must have $b_i - \boldsymbol{a}_i\boldsymbol{p} \geq 0$, i.e., $d_i \geq 0$. As l is maximized, there must exist d_i such that $l = d_i$, otherwise l can still be increased. Then we know that $l \geq 0$. (If $\boldsymbol{p}$ is outside the polyhedron, we must have $l < 0$, and l is not maximized.)

If the polyhedron has empty interior, the point $\boldsymbol{p}$ must locate on the boundary of the polyhedron, then one of the d_i is equal to 0, so l must be 0.

Now we need to prove that if $l = 0$, $\mathcal{P}$ must have empty interior. It is equivalent to: if $\mathcal{P}$ has non-empty interior, then $l > 0$. Assume $\boldsymbol{p}$ is on the boundary of $\mathcal{C}_i$, i.e. $d_i = l = 0$. As the polyhedron has non-empty interior, if we move the point $\boldsymbol{p}$ a little towards the interior of the polyhedron, we will have $l = d_i > 0$. That means when $l = 0$ the objective function l is not maximized. Therefore, if the polyhedron has non-empty interior, l must be lager than 0. □

We just proved that when $l = 0$ the polyhedron has empty interior, but the Simplex solver in GLPK uses floating-point arithmetic, and thus we cannot perform the test $l = 0$ if l is a floating point number. Therefore we use a relative large threshold 10^{-2} for comparison. If $l < 10^{-2}$, we will check the satisfiability of the obtained interior point by $\boldsymbol{a}_i\boldsymbol{p} \leq b + t(\boldsymbol{a}_i, \boldsymbol{p})$, where $t(\boldsymbol{a}_i, \boldsymbol{p}) = 1.01n\varepsilon|\boldsymbol{a}_i|^{\mathsf{T}}|\boldsymbol{p}|$ is the threshold (Equation 2.9). If the satisfiability test fails, the algorithm will report that the floating point solver cannot find a central point. Then we switch to a rational solver to deal with the problem. We do not need to test the correctness of the optimization, as we just try to find a point at the center, and any point near the center is fine.

Computing an irredundancy witness point In the process of testing if a constraint $\mathcal{C}_i : \boldsymbol{a}_i\boldsymbol{x} \leq b_i$ is redundant with respect to other constraints $\mathcal{C}_j$, where $j \neq i$, we need to solve a satisfiability problem: $\exists \boldsymbol{x},\ \boldsymbol{a}_i\boldsymbol{x} > b_i, \bigwedge_{j\neq i} \boldsymbol{a}_j\boldsymbol{x} \leq b_j$ for computing the irredundancy witness point of the constraint $\mathcal{C}_i$ (Section 3.2.3.1). For efficiency, we solve this problem in floating point using GLPK. However, GLPK does not support strict inequalities, thus we need tricks to deal with them.

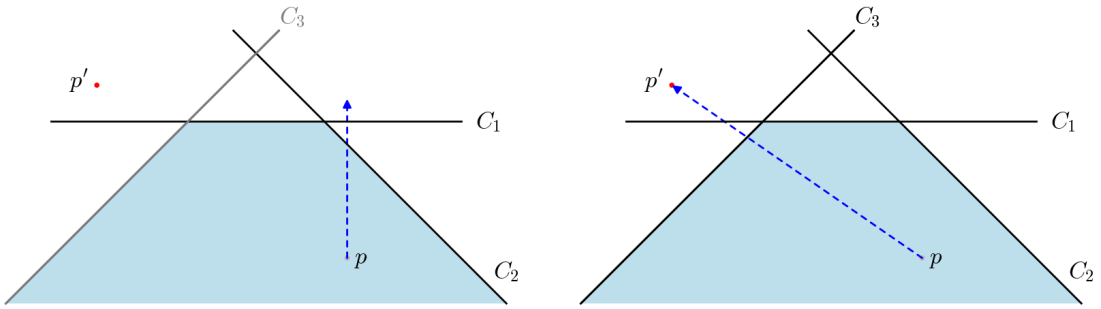
One way is to shift the strict inequality constraint $\mathcal{C}_i$ a little for obtaining a non-strict inequality $\mathcal{C}_i' : \boldsymbol{a}_i\boldsymbol{x} \geq b_i + \varepsilon$, where ε is a positive constant. This method is however requires to provide a suitable ε, which is the same problem that we faced for computing the interior point. One exception is that when the polyhedron is a cone, it is always possible to find a point that satisfies $\boldsymbol{a}_i\boldsymbol{x} \geq b_i + \varepsilon,\ \bigwedge_{j\neq i} \boldsymbol{a}_i\boldsymbol{x} \leq b$ by shifting the constraint $\mathcal{C}_i$, no matter how large ε is.

We designed another method for non-conic polyhedra. Instead of solving a satisfiability problem, we solve the optimization problem shown in Problem 3.8. The found optimal vertex $\boldsymbol{x}^*$ is the solution we are looking for if it satisfies $\boldsymbol{a}_i\boldsymbol{x}^* > b_i + t(\boldsymbol{a}_i, \boldsymbol{x}^*)$ (see Equation 2.8 about t).

$$\begin{aligned} &\text{maximize} && \boldsymbol{a}_i\boldsymbol{x} \\ &\text{subject to} && \boldsymbol{a}_j\boldsymbol{x} \leq b_j \quad (\forall j \neq i) \end{aligned} \tag{3.8}$$

Example 3.4. Let use consider the polyhedron shown in Figure 3.6: $\mathcal{P} = \{\mathcal{C}_1 : x_2 \leq 2, \mathcal{C}_2 : x_1 + x_2 \leq 7, \mathcal{C}_3 : -x_1 + x_2 \leq 0\}$, whose interior point is $\boldsymbol{p}$. We want to test the redundancy status of the constraint $\mathcal{C}_1$. In Figure 3.6(a), the ray launched from $\boldsymbol{p}$ to $\mathcal{C}_1$ hit $\mathcal{C}_2$ first, and found the point $\boldsymbol{p}'$ which violates $\mathcal{C}_1$ and satisfies $\mathcal{C}_2$. Then in Figure 3.6(b) the ray launched from $\boldsymbol{p}$ to $\boldsymbol{p}'$ hit $\mathcal{C}_3$ first, so we need to find a point that violates $\mathcal{C}_1$ and satisfies $\mathcal{C}_2$, $\mathcal{C}_3$, i.e., solving the satisfiability problem $\{x_2 > 2, x_1 + x_2 \leq 7, -x_1 + x_2 \leq 0\}$.

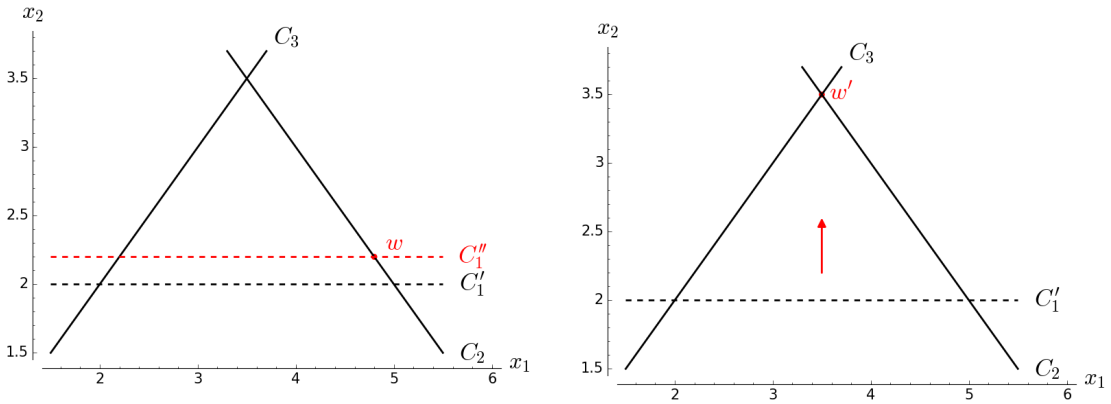
In Figure 3.7(a) we shift the non-strict inequality $\mathcal{C}_1' : x_2 > 2$ and obtain $\mathcal{C}_1''$. By solving the satisfiability problem $\{x_2 \geq 2 + \varepsilon, x_1 + x_2 \leq 7, -x_1 + x_2 \leq 0\}$ we get the witness point $\boldsymbol{w}$. If we compute the optimum in the direction x_2 with constraints $\{-x_1 + x_2 \leq 0, x_1 + x_2 \leq 7\}$, as it is shown in Figure 3.7(b), we obtain a witness point $\boldsymbol{w}'$.



(a) The ray hit $\mathcal{C}_2$ first and found $\boldsymbol{p}'$.

(b) The ray launched from $\boldsymbol{p}$ to $\boldsymbol{p}'$ hit $\mathcal{C}_3$ first.

Figure 3.6



(a) Compute the irredundancy witness point $\boldsymbol{w}$ of $\mathcal{C}_1'$ by ε-shifting. The witness $\boldsymbol{w}$ is on $\mathcal{C}_1''$ and satisfies $\mathcal{C}_2$, $\mathcal{C}_3$ but violates $\mathcal{C}_1'$.

(b) Compute the irredundancy witness point by optimizing in the direction x_2 shown by the red vector.

Figure 3.7

3.2.3.3 Preprocess of raytracing minimization

Before giving the whole floating point raytracing minimization algorithm, there is another point to be explained. The polyhedron to be minimized may contain implicit equalities, i.e., an equality resulting from the combination of several inequalities, for instance $x_1 \leq x_2$, $x_2 \leq x_3$, $x_3 \leq x_1$ producing $x_1 = x_2 = x_3$. In this case the polyhedron to be minimized has an empty interior. As we have explained, the raytracing minimization seeks for a point inside the polyhedron to be minimized. When there are implicit equalities the raytracing minimization method cannot find the interior point, and thus we need to eliminate these implicit equalities before applying the raytracing minimization.

Eliminating implicit equalities Consider the polyhedron $\mathcal{P} = \bigwedge_i \boldsymbol{a}_i\boldsymbol{x} \leq b_i$. The way to remove the implicit constraints is that we select the constraints that may construct the implicit constraints in floating point computation, and then from these selected constraints we use a rational solver to search for the combination of inequalities constructing the equality. The purpose of using the floating point solver is to decrease the number of constraints that will be given to the rational solver and to improve the efficiency.

As what we have mentioned in Section 3.2.3.2, the minimization algorithm contains the step of testing the interior emptiness of the polyhedron $\mathcal{P}$. If the floating point solver reports that $\mathcal{P}$ has an empty interior, there may exist implicit inequalities. Then we try to find a point $\boldsymbol{p}$ which fulfills all the constraints $\mathcal{C}_i : \boldsymbol{a}_i\boldsymbol{x} \leq b_i$ of $\mathcal{P}$. If $\boldsymbol{p}$ can be found, $\mathcal{P}$ probably contains implicit equalities. In this case we select all the constraints $\mathcal{C}_i$ fulfill $\boldsymbol{a}_i\boldsymbol{p} \leq b_i + t(\boldsymbol{a}_i, \boldsymbol{p}), i \in \{1, \cdots, k\}$, i.e, the constraints closed to $\boldsymbol{p}$ (see Equation 2.8 about t). Among these selected constraints we look for a combination producing the equation $\boldsymbol{a}_i\boldsymbol{x} - b_i = -\sum \alpha_j(\boldsymbol{a}_j\boldsymbol{x} - b_j),\ \alpha_j \geq 0,\ i \neq j$. If this combination cannot be found, it means that there is no implicit equalities; otherwise we will obtain a set of equalities. By substituting the equalities into the inequalities, we will obtain the reduced inequalities, which is explained in Example 3.5. All the implicit equalities will be eliminated by repeating this process.

> **Example 3.5.** Given a polyhedron $\mathcal{P} = \{\mathcal{C}_1 : 2x_1 + x_2 \leq 5,\ \mathcal{C}_2 : x_1 - 3x_2 \leq 2, \mathcal{C}_3 : -x_1 + 3x_2 \leq -2\}$, the constraints $\mathcal{C}_2$ and $\mathcal{C}_3$ construct an equality $x_1 - 3x_2 = 2$. As the equality does not appear in the polyhedron explicitly, it is an implicit equality. Once obtain the equality $x_1 - 3x_2 = 2$, we substitute x_1 by x_2, i.e. $x_1 = 3x_2 + 2$. By substituting x_1 of the remaining inequality $2x_1 + x_2 \leq 5$, we obtain $7x_2 \leq 1$.

3.2.3.4 The floating point raytracing minimization algorithm

The pseudo-code of the algorithm of the raytracing minimization is shown in Algorithm 1. As we said, $\mathbb{Q}$ denotes the field of rational numbers, and $\mathbb{F}$ is the set of finite floating-point numbers. The variables stored as floating point numbers are denoted by name$^{\mathbb{F}}$.

Termination There is another point to be explained about Algorithm 1. During the process of computing the witness point, Algorithm 1 may loop forever because of imprecision of floating point

Algorithm 1: Raytracing minimization algorithm.

Input: $poly^{\mathbb{F}}$: the polyhedron to be minimized
Output: the indices of the irredundant constraints and the witness point
Function Minimize($poly^{\mathbb{F}}$)

```
// try to obtain a central point of the polyhedron
p^F = GetCentralPoint(poly^F)
if LP is unbounded then
    // if cannot find a central point, compute any point inside the polyhedron
    p^F = GetInteriorPoint(poly^F)
if p^F == none then
    report polyhedron has empty interior
    terminate
// launch rays from the interior point to each boundary
rays^F = LaunchRays(poly^F, p^F)
/* compute the distance from the interior point to each boundary along the direction
   of ray^F, and set the constraint whose boundary was first met as irredundant */
foreach ray^F in rays^F do
    constraintIdx = FirstHitConstraint(poly^F, ray^F, p^F)
    SetAsIrredundant(constraintIdx)
foreach constraint idx in undetermined constraints do
    /* if the floating point minimization cannot determine the redundancy status,
       set the constraint as redundant for guaranteeing soundness */
    if cannot determine then
        SetAsRedundant(idx)
    else
        /* if the irredundancy witness point exists, the corresponding constraint is
           irredundant; otherwise it is redundant */
        if found irredundancy witness point then
            SetAsIrredundant(idx)
        else
            SetAsRedundant(idx)
return indices of irredundant constraints with their witness point
```

computation. Considering the example in Figure 3.8, in which the polyhedron is colored in blue, and $\boldsymbol{p}$ is the interior point. We want to check if $\mathcal{C}_1$ is redundant. The ray $\boldsymbol{r}$ launched from $\boldsymbol{p}$ to $\boldsymbol{p}'$ hit $\mathcal{C}_1$ first, and then hit $\mathcal{C}_2$. In exact computation the algorithm should report that $\mathcal{C}_1$ is irredundant. But if we compute in floating point numbers, when the gray area is extremely small, the distance l_1 from $\boldsymbol{p}$ to $\mathcal{C}_1$ and the distance l_2 from $\boldsymbol{p}$ to $\mathcal{C}_2$ are nearly equal. Because of rounding errors, the floating point solver may misjudge that $l_2 < l_1$, i.e., the ray $\boldsymbol{r}$ hit $\mathcal{C}_2$ first. Then it would find the same point $\boldsymbol{p}'$, which violates $\mathcal{C}_1$ and satisfies $\mathcal{C}_2$. Then the same ray $\boldsymbol{r}$ would be launched from $\boldsymbol{p}$ to $\boldsymbol{p}'$. Because of the misjudge of the comparison of l_1 and l_2, the process would be repeated, and Algorithm 1 would loop for ever.

To avoid this problem, we need to maintain a list of constraints which are hit before reaching $\mathcal{C}_1$. If the hit constraint exists in the list, the process of checking redundancy should stop and report the constraint $\mathcal{C}_1$ as redundant.

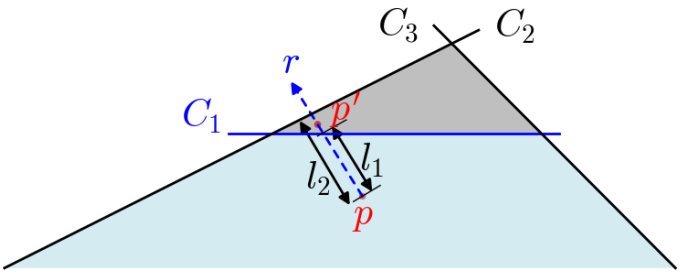


Figure 3.8 – When the gray area is extremely small, the ray may misjudge the first hit constraint as $\mathcal{C}_2$ instead of $\mathcal{C}_1$.

Soundness The algorithm of raytracing minimization using floating point arithmetic is sound, meaning that the polyhedron which is minimized by the floating point minimization algorithm always contains the exact minimized polyhedron. In other words, the minimization algorithm should always report a constraint as redundant when it cannot determine the redundancy status of this constraint.

In the first phase of raytracing minimization, the determination of irredundancy consists in comparing the distances from the interior point to each constraint, which are floating point numbers. A constraint $\mathcal{C}_i$ is irredundant only if $d_i + t < d_j$, $\forall j \neq i$, where d_i (or d_j) is the distance from the interior point to the constraint $\mathcal{C}_i$ (or $\mathcal{C}_j$), $t = max(|d_i|, |d_j|)\varepsilon$ is the threshold and ε is the machine epsilon. If there are several constraints $\mathcal{C}_j$ satisfy $|d_i - d_j| < t$, then $\mathcal{C}_i$ will be marked as undetermined and will be remained to the second phase. In the second phase, if GLPK cannot find an irredundancy witness point $\boldsymbol{w}$ that fulfills $\mathcal{C}_i : \boldsymbol{a}_i\boldsymbol{x} \leq b_i$, we can believe it and mark $\mathcal{C}_i$ as redundant; if GLPK found the irredundancy witness point, we need to test that the point really violates $\mathcal{C}_i$ with a threshold t: $\boldsymbol{a}_i\boldsymbol{w} > b_i + t(\boldsymbol{a}_i, \boldsymbol{p})$ (see Equation 2.8 about t). If the test fails, the constraint $\mathcal{C}_i$ will be marked as redundant.

We have seen that in the second phase, a constraint is reported as redundant if: i) GLPK cannot find the witness point; ii) the thresholded satisfiability test $\boldsymbol{a}_i\boldsymbol{w} > b_i + t(\boldsymbol{a}_i, \boldsymbol{p})$ fails; iii) there is numerical errors during the computation of witness point (the same hit constraint appears repeatedly).

As a result, some irredundant constraints may be misjudged as redundant, which results in over-approximation, but not unsoundness in static analysis.

3.3 Projection and Convex hull

We present in this section how the PLP solver can be used to compute projection and convex hull, following the work of Maréchal et al. [20]. Then we will be able to explain our efficiency improvement using floating point computations.

3.3.1 Projection

Given a polyhedron $\mathcal{P}$ in d dimensions, the polyhedral projection can eliminate 1 to $d-1$ dimensions and obtain a polyhedron $\mathcal{P}'$ in lower dimensions such that $\mathcal{P} \sqsubset \mathcal{P}' \times \mathbb{Q}_1 \times \cdots \times \mathbb{Q}_e$ for elimination of dimensions 1 to e.

> **Example 3.6.** Figure 3.9 shows an example of projecting a 3D polyhedron to 2D. The polyhedron to be projected (in gray) is $\mathcal{P} = \{-15x_1 - x_2 + 6x_3 \geq 0,\ -3x_1 - 4x_2 + 5x_3 \geq 0,\ -2x_2 + x_3 \geq -3, -x_2 + 2x_3 \geq 0, 6x_1 + x_2 + 10x_3 \geq 0, x_1 + 4x_2 - 24x_3 \geq -59, 2x_1 + 8x_2 + 11x_3 \geq 0\}$. We eliminate the variable x_3 and obtain the projected polyhedron (in purple) $\mathcal{P}' = \{-x_1 \geq -1,\ 77x_1 + 32x_2 \geq -259,\ x_1 - 44x_2 \geq -131,\ x_1 + 4x_2 \geq -11\}$.

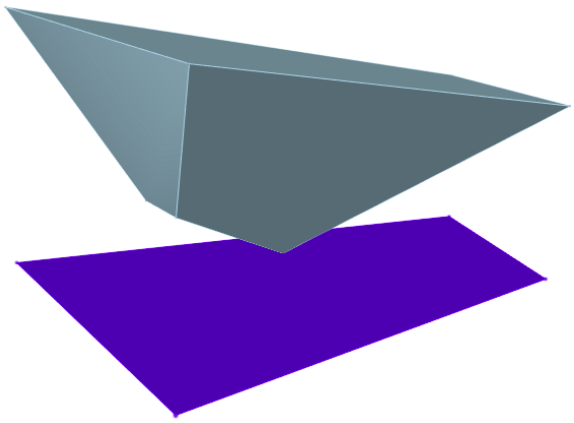


Figure 3.9 – Projecting a 3D polyhedron to 2D.

Fourier-Motzkin elimination [31] is a classic method to project polyhedra in constraint-only description. This method has two drawbacks: i) it can only eliminate one variable each time, ii) many redundant constraints could be produced during the process. Those drawbacks are the motivation to compute projection using PLP [19, 18, 20].

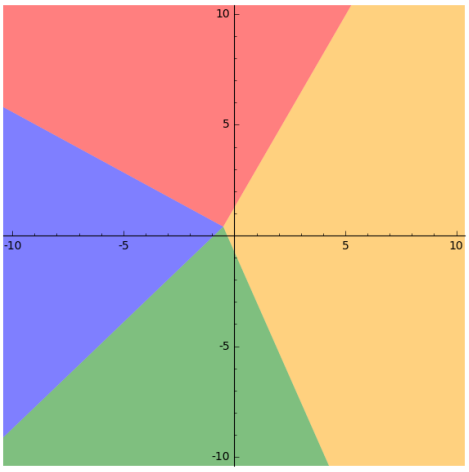
PLP for projection The polyhedron to be projected is $\mathcal{P}$: $A\boldsymbol{x} + \boldsymbol{b} \geq 0$. Assume we will eliminate $x_p, \cdots, x_q$, where $1 \leq p \leq q \leq n$. To compute the projection, following [20] we construct a PLP shown in Problem 3.9, in which $\boldsymbol{x}$ are the parameters, and $\boldsymbol{\lambda}$ are decision variables, with $\boldsymbol{x} = [x_1, \cdots, x_n]^\mathsf{T}$,

$\boldsymbol{\lambda} = [\lambda_0, \cdots, \lambda_m]^\mathsf{T}$.

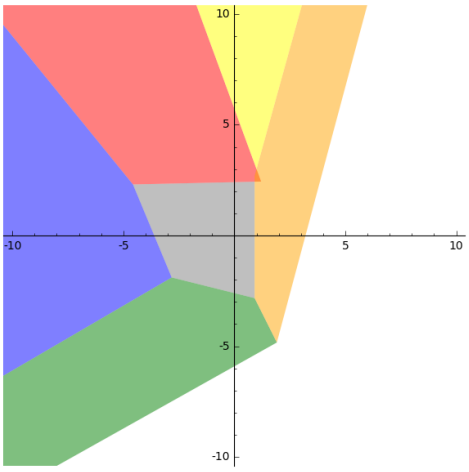
$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m} \lambda_i(\boldsymbol{a}_i\boldsymbol{x} + b_i) + \lambda_0 \\ \text{subject to} \quad & \sum_{i=1}^{m} \lambda_i(\boldsymbol{a}_i\boldsymbol{p} + b_i) + \lambda_0 = 1 \qquad (*) \\ & \sum_{i=1}^{n} a_{ij}\lambda_i = 0 \quad (\forall j \in \{p, \cdots, q\}) \\ \text{and} \quad & \lambda_i \geq 0 \quad (\forall i \in \{0, \cdots, m\}) \end{aligned} \tag{3.9}$$

where $\boldsymbol{p} = [p_1, \cdots, p_n]^\mathsf{T}$ is a point inside the projected polyhedron and it is called *normalization point*.

The (*) equation in Problem 3.9 is called *normalization constraint*, which makes sure that: i) at least one of the λ_i is not zero; ii) all the constraints of regions intersect at the same point $\boldsymbol{p}$; iii) the obtained projected polyhedron is free of redundancy (see [1] for explanation). Let us consider the projected polyhedron shown in Figure 3.9. Figure 3.10(a) shows the regions obtained from the normalized PLP problem, i.e., Problem 3.9. If we replace the normalization constraint with $\sum_{i=0}^{m} \lambda_i = 1$, which only guarantees that not all the variables are zero, we obtain the regions without normalization in Figure 3.10(b). In these two figures, the regions with the same color have the same optimal function. We can see that in Figure 3.10(b) there are two more regions which are in yellow and gray. The yellow region corresponds to a redundant constraint of the projected polyhedron, and the gray region corresponds to the trivial constraint $0 \leq 1$.



(a) The regions with normalization of Problem 3.9.



(b) The regions with the constraint $\sum \lambda_i = 1$.

Figure 3.10

3.3.2 Convex hull

The convex hull of two polyhedra is the smallest convex set that contains the two polyhedra. Given two polyhedra $\mathcal{P}_1 : A'\boldsymbol{x}' + \boldsymbol{b}' \geq 0$, $\mathcal{P}_2 : A''\boldsymbol{x}'' + \boldsymbol{b}'' \geq 0$, the convex hull of $\mathcal{P}_1$ and $\mathcal{P}_2$ is: $\mathcal{P}_1 \sqcup \mathcal{P}_2 = \{\boldsymbol{x} | \boldsymbol{x} = \alpha\boldsymbol{x}' + (1-\alpha)\boldsymbol{x}'', A'\boldsymbol{x}' + \boldsymbol{b}' \geq 0, A''\boldsymbol{x}'' + \boldsymbol{b}'' \geq 0, 0 \leq \alpha \leq 1\}$.

The Figure 3.11 shows an example of the convex hull of two polyhedra in 2D. The orange areas are the polyhedra, and the red frame is the boundary of the convex hull.

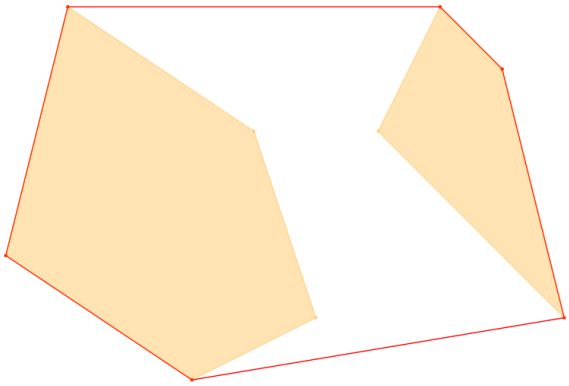


Figure 3.11 – The convex hull of two polyhedra in 2D.

PLP for convex hull We recall here the method designed by Maréchal et al. [20] to compute the convex hull via PLP. Let $\mathcal{P} = \mathcal{P}_1 \sqcup \mathcal{P}_2$ with $\mathcal{P}_1$: $A'\boldsymbol{x}' + \boldsymbol{b}' \geq 0$ and $\mathcal{P}_2$: $A''\boldsymbol{x}'' + \boldsymbol{b}'' \geq 0$, where $A' \in \mathbb{Q}^{m' \times n}$, $\boldsymbol{b}' \in \mathbb{Q}^{m'}$, $A'' \in \mathbb{Q}^{m'' \times n}$ and $\boldsymbol{b}'' \in \mathbb{Q}^{m''}$. As $\mathcal{P}_1 \sqsubseteq \mathcal{P}$ and $\mathcal{P}_2 \sqsubseteq \mathcal{P}$, each constraint of $\mathcal{P}$ is redundant with respect to both $\mathcal{P}_1$ and $\mathcal{P}_2$. According to what we have explained in Section 3.2.2, for each constraint $\mathcal{C}_i : \boldsymbol{a}_i\boldsymbol{x} + b_i \geq 0$ of $\mathcal{P}$, we have $\exists \boldsymbol{\lambda}' \geq 0, \boldsymbol{a}_i\boldsymbol{x} + b_i = \sum_{j=1}^{m'} \lambda'_j(\boldsymbol{a}'_j\boldsymbol{x}' + b'_j) + \lambda'_0 = \sum_{j=1}^{m''} \lambda''_j(\boldsymbol{a}''_j\boldsymbol{x}'' + b''_j) + \lambda''_0$. Thus we have the constraints (**) in Problem 3.10. The constraint (*) is the normalization constraint, as in Problem 3.9.

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{m'} \lambda'_i(\boldsymbol{a}'_i\boldsymbol{x}' + b'_i) + \lambda'_0 & & \\
\text{subject to} \quad & \sum_{i=1}^{m'} \lambda'_i(\boldsymbol{a}'_i\boldsymbol{p} + b'_i) + \lambda'_0 = 1 & & (*) \\
& A'^{\mathsf{T}}\boldsymbol{\lambda}' - A''^{\mathsf{T}}\boldsymbol{\lambda}'' = 0 & & \\
& b'^{\mathsf{T}}\boldsymbol{\lambda}' + \lambda'_0 - b''^{\mathsf{T}}\boldsymbol{\lambda}'' - \lambda''_0 = 0 & & (**) \\
\text{and} \quad & \lambda'_i \geq 0, \lambda''_j \geq 0 \quad (\forall i \in \{0, \cdots, m'\}, \forall j \in \{0, \cdots, m''\}) & & (**)
\end{aligned}
\tag{3.10}
$$

where $\boldsymbol{p} = [p_1, \cdots, p_n]^{\mathsf{T}}$ is the normalization point.

3.4 The sequential algorithm of PLP solver for computing projection and convex hull

In this section we present our sequential algorithm of PLP solver. We will see how to solve the PLP problem in floating point numbers and reconstruct the rational results. Some checkers are provided to ensure the soundness and precision of the results. When it is necessary, rational solvers can be invoked for computing exact solutions.

3.4.1 Overview of the sequential algorithm

In the pseudo-code of the sequential algorithm of the PLP solver (Algorithm 2), we annotate data with $name^{type}$, where *name* is the name of data and $type$ is either $\mathbb{Q}$ or/and $\mathbb{F}$. $\mathbb{Q} \times \mathbb{F}$ means that the data is stored in both rational and floating-point numbers. The PLP problem is stored in both floating point and rational numbers. The optimal functions are in rational numbers, and the corresponding regions are in both floating point and rational numbers.

We use a work list to store a set of tasks. A task is stored as a triple $(\boldsymbol{w}, \mathcal{R}_{from}, F_{from})$, in which $\boldsymbol{w}$ is generated by the region $\mathcal{R}_{from}$ and it is the irredundancy witness point of the constraint whose boundary is F_{from}. The task points $\boldsymbol{w}$ are in the parameters' domain, and we use them to compute unexplored regions. An initial task will be added into the work list, and more tasks will be appended during the execution of the algorithm.

Given a PLP problem, the solution goes through several steps. We first list the steps, and then explain each step into details in the following sections.

Step 1 Construct the PLP for computing projection or convex hull.

Step 2 Pick up a random valuation of the parameters for obtaining a initial task $(\boldsymbol{w}, \mathsf{none}, \mathsf{none})$ and push it into the work list W.

Step 3 Fetch a task $(\boldsymbol{w}, \mathcal{R}_{from}, F_{from})$ from W.

Step 4 Check if the point $\boldsymbol{w}$ is covered by any known region. If it is not, go to Step 5; otherwise go to Step 10.

Step 5 Instantiate the parametric objective with $\boldsymbol{w}$ to obtain a standard (non-parametric) LP problem.

Step 6 Solve the LP problem obtained in Step 5 using GLPK to obtain a basis B.

Step 7 Reconstruct the PLP problem which is equivalent to the original one associated to B.

Step 8 Minimize the region, and store the current solution associated to B.

Step 9 Add more tasks of which the task points are outside the obtained region into the work list.

Step 10 Check if the obtained region and the region $\mathcal{R}_{from}$ are adjacent.

Step 11 Go to Step 2 and repeat the processes until $W = \emptyset$.

3.4.2 Construction of PLP problems (Step 1)

The first step of our approach is to construct the PLP problem for computing projection (Problem 3.9) or convex hull (Problem 3.10). For constructing the PLP problem, we need to choose a normalization point. By instantiating the parameters with this point, we obtain the normalization constraint. The set of constraints of the PLP problem are all equalities, and thus we need to eliminate the redundant equalities, i.e. the equalities which are equivalent to others, which can be performed by *Gaussian Elimination*. Then we substitute the equalities into the inequalities $\boldsymbol{\lambda} \geq 0$ for reconciling our approach with GLPK. After that we minimize the set of inequality constraints. The details are explained below. At the end of Step 1, we store the constraints of the PLP problem in both rational and floating point numbers. The rational constraints are used for reconstructing rational results, and the floating point constraints are given to GLPK for solving the LP problems. The constraints in floating point numbers are inequalities, and then the Simplex algorithm in GLPK will add slacks variables to each inequalities, i.e., the inequalities will be transformed into equalities in GLPK. Thus we store the rational constraints as equalities such that they are in the same form as in GLPK.

Choosing a normalization point Consider the projection operator implemented via PLP and $\mathcal{P}' = \mathcal{P}/\{x_p, \cdots, x_q\}$ is the resulting polyhedron, i.e., the polyhedron obtained by projecting out the variables $\{x_p, \cdots, x_q\}$ of $\mathcal{P}$. The point $\boldsymbol{p}'$ used during PLP computation for normalization of $\mathcal{P}'$ must be in the interior of $\mathcal{P}'$. $\boldsymbol{p}'$ is computed from a point $\boldsymbol{p}$ in the interior of $\mathcal{P}$ by $\boldsymbol{p}' = \boldsymbol{p}/\{x_p, \cdots, x_q\}$, i.e., the components $\{x_p, \cdots, x_q\}$ of $\boldsymbol{p}$ are discarded. For convex hull, $\mathcal{P}' = \mathcal{P}_1 \sqcup \mathcal{P}_2$, the point $\boldsymbol{p}$ is chosen either in $\mathcal{P}_1$ or in $\mathcal{P}_2$. As $\mathcal{P}_1 \sqsubseteq \mathcal{P}'$ and $\mathcal{P}_2 \sqsubseteq \mathcal{P}'$, $\boldsymbol{p}' = \boldsymbol{p}$ must be in the interior of the obtained polyhedron $\mathcal{P}'$.

The normalization point $\boldsymbol{p}'$ in the projected polyhedron is computed in floating-point numbers. As the constraints of the PLP problem will be stored in both floating-point and rational numbers, we need to convert $\boldsymbol{p}'$ into rational for constructing the rational constraints, and make sure that the rational point is still inside the resulting polyhedron. The point $\boldsymbol{p}' \in \mathcal{P}'$ if $\boldsymbol{p} \in \mathcal{P}$, so we compute $\boldsymbol{p}$, convert it into rational numbers, and then check if $\boldsymbol{p}$ is still inside $\mathcal{P}$. This is straightforward: firstly we keep k digits in decimal part and convert the value into rational by computing its continued fraction. If the rational point does not satisfy $\mathcal{P}$, we keep $k+1$ digits and do that repeatedly until a satisfiable point is obtained.

Applying Gaussian Elimination A PLP problem may contain constraints that are equivalent to each other, i.e., redundant equalities, which should be removed. We apply *Gaussian Elimination* to the *augmented matrix* (Definition 3.3) of the constraints of the PLP problem for obtaining the augmented matrix in *reduced row echelon form*[2], and thus all the redundant equalities will be removed.

> **Definition 3.3** (Augmented matrix)**.** Given the system of equalities $A\boldsymbol{x} = \boldsymbol{b}$, its augmented matrix is $[A|\boldsymbol{b}]$.

[2] To adapt to the description in our approach, we define the reduced echelon from differently from that in the textbook.

Definition 3.4 (Reduced row echelon form of a matrix)**.** A matrix is in reduced row echelon form if it fulfills the conditions below:

- All rows with at least one nonzero element are above rows of all zeros.
- Each nonzero row contains an element 1.
- Each column containing the element 1 has zeros everywhere else.

Example 3.7. Considering the matrix $\begin{bmatrix} 3 & -1 & 1 & 0 & 5 \\ 1 & -3 & 0 & 3 & -3 \end{bmatrix}$, its reduced form is $\begin{bmatrix} 1 & 0 & \frac{9}{24} & -\frac{3}{8} & \frac{9}{4} \\ 0 & 1 & \frac{1}{8} & -\frac{9}{8} & \frac{7}{4} \end{bmatrix}$. The matrix $\begin{bmatrix} 1 & \frac{9}{24} & 0 & -\frac{3}{8} & \frac{9}{4} \\ 0 & \frac{1}{8} & 1 & -\frac{9}{8} & \frac{7}{4} \end{bmatrix}$ is also in reduced form.

Reconciling with GLPK To solve LP problems, GLPK always adds row variables to the constraints, no matter they are equalities or inequalities. Assuming we have an equality $\boldsymbol{a}_i\boldsymbol{\lambda} = b_i$, which is expressed in GLPK as an equality $r_i = \boldsymbol{a}_i\boldsymbol{\lambda}$ with a bound $r_i = b_i$, where r_i is a row variable. The systematic introduction of row variables can introduce spurious dimension in the results of our PLP solver, and this will lead to wrong regions, which will be explained in Example 3.8. The idea to reconcile GLPK and our procedure is to prevent GLPK from adding useless row variables to equality constraints. Hence we transform the original problem into a reduced form without equality constraints. Let us illustrate it on an example.

Example 3.8. Consider a polyhedron $\{-40x_2 \leq -7,\ -x_1 \leq 20,\ -x_1 + 5x_2 + 3x_3 \leq 20\}$, from which we want to eliminate the variable x_1 using our PLP solver. We construct the PLP problem for projection in Problem 3.11 with the normalization point $(0, 1.2, 0)$.

$$\begin{aligned} \text{minimize} \quad & \lambda_1(40x_2 - 7) + \lambda_2(x_1 + 20) + \lambda_3(x_1 - 5x_2 - 3x_3 + 20) + \lambda_0 \\ \text{subject to} \quad & 41\lambda_1 + 20\lambda_2 + 14\lambda_3 + \lambda_0 = 1 \\ & \lambda_2 + \lambda_3 = 0 \\ \text{and} \quad & \lambda_1, \lambda_2, \lambda_3, \lambda_0 \geq 0 \end{aligned} \tag{3.11}$$

The PLP solver instantiates the parameters (x_1, x_2) of the objective function with a point, say $(1, 1, 1)$, to obtain a standard (non-parametric) LP problem shown in Problem 3.12. GLPK transforms this LP problem into Problem 3.13.

$$\begin{aligned} \text{minimize} \quad & 33\lambda_1 + 21\lambda_2 + 13\lambda_3 + \lambda_0 \\ \text{subject to} \quad & 41\lambda_1 + 20\lambda_2 + 14\lambda_3 + \lambda_0 = 1 \\ & \lambda_2 + \lambda_3 = 0 \\ \text{and} \quad & \lambda_1, \lambda_2, \lambda_3, \lambda_0 \geq 0 \end{aligned} \tag{3.12}$$

$$\begin{aligned} \text{minimize} \quad & 33\lambda_1 + 21\lambda_2 + 13\lambda_3 + \lambda_0 \\ \text{subject to} \quad & r_1 = 41\lambda_1 + 20\lambda_2 + 14\lambda_3 + \lambda_0, \quad r_1 = 1 \\ & r_2 = \lambda_2 + \lambda_3, \qquad\qquad\quad r_2 = 0 \\ \text{and} \quad & \qquad\qquad\qquad\qquad\quad \lambda_1, \lambda_2, \lambda_3, \lambda_0 \geq 0 \end{aligned} \tag{3.13}$$

Solving Problem 3.13 with GLPK leads to choosing $\lambda_1,\ r_2$ as basic variables. Then we can restate Problem 3.11 into an equivalent PLP with the basis $\lambda_1,\ r_2$:

$$\begin{aligned} \text{minimize} \quad & r_1(\tfrac{40}{41}x_2 - \tfrac{7}{41}) + \lambda_2(x_1 - \tfrac{800}{41}x_2 + \tfrac{960}{41}) + \lambda_3(x_1 - \tfrac{765}{41}x_2 - 3x_3 + \tfrac{918}{41}) + \lambda_0(-\tfrac{40}{41}x_2 + \tfrac{48}{41}) \\ \text{subject to} \quad & \lambda_1 = \tfrac{1}{41}r_1 - \tfrac{20}{41}\lambda_2 - \tfrac{14}{41}\lambda_3 - \tfrac{1}{41}\lambda_0, \quad r_1 = 1 \\ & r_2 = \lambda_2 + \lambda_3, \qquad\qquad\qquad\quad r_2 = 0 \\ \text{and} \quad & \qquad\qquad\qquad\qquad\qquad\quad \lambda_1, \lambda_2, \lambda_3, \lambda_0 \geq 0 \end{aligned} \tag{3.14}$$

We know that $\lambda_2 = -\lambda_3 = 0$, as $\lambda_2, \lambda_3 \geq 0$, and $r_1 = 1$, which means the objective function in Problem 3.14 is equivalent to $(\frac{40}{41}x_2 - \frac{7}{41}) + \lambda_0(-\frac{40}{41}x_2 + \frac{48}{41})$, where λ_2 and r_1 are replaced by their fixed bound. Since λ_0 has a lower bound, the objective function reaches minimum when the parametric coefficient of λ_0 is non-negative, i.e. in the region $\{-\frac{40}{41}x_2 + \frac{48}{41} \geq 0\}$. This is the expected result. But the information $\lambda_2 = -\lambda_3$ is not straightforward for GLPK because of the introduce of λ_2. When our region reconstruction procedure requests GLPK to get the bound of the non-basic variables r_1, λ_2 and λ_3, it obtains fixed bound $r_1 = 1$ but lower bound $\lambda_2 \geq 0,\ \lambda_3 \geq 0$. Based on this information λ_2 and λ_3 remain in the objective function, which leads to the region $\{x_1 - \frac{800}{41}x_2 + \frac{960}{41} \geq 0,\ x_1 - \frac{765}{41}x_2 - 3x_3 + \frac{918}{41} \geq 0,\ -\frac{40}{41}x_2 + \frac{48}{41} \geq 0\}$ in $\mathbb{Q}^3$, whereas we expected a region in $\mathbb{Q}^2$. The error is obvious, as x_1 should have been eliminated.

To avoid this issue, we transform the equalities into inequalities before giving them to GLPK. We substitute $\lambda_1,\ \lambda_2$ in the inequalities and the objective function by $\lambda_1 = \frac{6}{41}\lambda_3 - \frac{1}{41}\lambda_0 + \frac{1}{41},\ \lambda_2 = -\lambda_3$, and obtain the reduced form of the constraints of Problem 3.11 is shown in Problem 3.15. The substitution can be computed by Gaussian Elimination. The augmented matrix of the constraints

$\{41\lambda_1+20\lambda_2+14\lambda_3+\lambda_0=1,\ \lambda_2+\lambda_3=0\}$ is $\begin{bmatrix} 41 & 20 & 14 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$. By applying Gaussian Elimination, we obtain the reduced matrix $\begin{bmatrix} 1 & 0 & -\frac{6}{41} & \frac{1}{41} & \frac{1}{41} \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$, which is equivalent to the substitution $\lambda_1=\frac{6}{41}\lambda_3-\frac{1}{41}\lambda_0+\frac{1}{41},\ \lambda_2=-\lambda_3$.

$$\begin{aligned} \text{subject to} \quad & \frac{6}{41}\lambda_3-\frac{1}{41}\lambda_0+\frac{1}{41}\geq 0 \\ & -\lambda_3\geq 0 \\ \text{and} \quad & \lambda_3,\lambda_0\geq 0 \end{aligned} \tag{3.15}$$

The next step is to remove the redundant constraints. As what explained in Section 3.2.3.3, we need to eliminate the implicit equalities before using the raytracing minimization. We found the implicit equality $\lambda_3=0$, substituted it into the inequality constraints and the objective function, and obtained our final LP problem shown in Problem 3.16. Then we need to use the raytracing minimization to remove the redundant constraints. In our example, the two constraints are both irredundant.

$$\begin{aligned} \text{subject to} \quad & -\frac{1}{41}\lambda_0+\frac{1}{41}\geq 0 \\ \text{and} \quad & \lambda_0\geq 0 \end{aligned} \tag{3.16}$$

GLPK will add a row variable to the constraint and obtain $r_1=\lambda_0,\ r_1\geq 0$. To keep the equivalent representation of the constraint in GLPK and in our approach, we add a slack variable λ_1 to the constraint and obtain the equality $\frac{1}{41}\lambda_0+\lambda_1=\frac{1}{41}$ with $\lambda_0,\ \lambda_1\geq 0$. We store this equality as the constraint of the PLP problem. The final PLP problem is shown in Problem 3.17, which is equivalent to the original PLP in Problem 3.11.

$$\begin{aligned} \text{minimize} \quad & \lambda_0(-\frac{40}{41}x_2+\frac{8}{41})+\frac{40}{41}x_2-\frac{7}{41} \\ \text{subject to} \quad & \frac{1}{41}\lambda_0+\lambda_1=\frac{1}{41} \\ \text{and} \quad & \lambda_1,\ \lambda_0\geq 0 \end{aligned} \tag{3.17}$$

3.4.3 Initial tasks (Step 2)

In Step 2 of the PLP solver, one initial point, which is different from the normalization point, will be randomly selected to trigger the algorithm. The initial task point will be inserted into the work list firstly, and then more task points will be inserted during the computation approach.

3.4.4 Checking belonging of points (Step 3-4)

Before exploring a region using a task point (Step 5), we need to check if the point is covered by any known region for avoiding recomputation. Assume we fetch a task $(\boldsymbol{q}, \mathcal{R}, F)$ from the work list, and test if the task point $\boldsymbol{q}$ is inside the region $\bigwedge_i \boldsymbol{o}_i \boldsymbol{x} \leq c_i$. As there is no strict inequality in floating-point arithmetic, we cannot simply test $\boldsymbol{o}_i \boldsymbol{q} < c_i$, and hence we test $\boldsymbol{o}_i \boldsymbol{q} \leq c_i - t(\boldsymbol{o}_i, \boldsymbol{q})$, where t is defined by Equation 2.8. If the test succeeds, it guarantees that $\boldsymbol{q}$ locates in the interior of the region. However, if it reports the point is not covered, the point may still be inside the region. This may lead to recomputation of the corresponding LP problem at a low probability. This approach guarantees that no task will be missed. The termination is guaranteed, as we maintain a set of bases (stored in a hash table) which are obtained from GLPK. If a basis is found to be existed in the hash table, the computation will terminate.

3.4.5 Obtaining an optimal basis by solving LP using GLPK (Step 5-6)

If the point $\boldsymbol{q}$ taken from the work list is not covered by any known region, we will construct the objective function of the LP problem by instantiating the parameters with $\boldsymbol{q}$. The LP problem is:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n} \lambda_i (\boldsymbol{a}_i \boldsymbol{q} + b_i) + \lambda_0 \\ \text{subject to} \quad & \sum_{i=1}^{n} \lambda_i (\boldsymbol{a}_i \boldsymbol{p} + b_i) + \lambda_0 = 1 \qquad (*) \\ & \sum_{i=1}^{n} a_{ij} \lambda_i = 0 \quad (\forall j \in \{p, \cdots, q\}) \\ \text{and} \quad & \lambda_i \geq 0 \quad (\forall i \in \{0, \cdots, n\}) \end{aligned} \tag{3.18}$$

Then we solve the LP problem with the Simplex method in GLPK. GLPK reconstructs the constraints into the form $(\lambda_B)_i = \sum_{j=1}^{n} a_{ij} (\lambda_N)_j + c_i$, where $(\lambda_B)_i$ is a basic variable, $(\lambda_N)_i$ is a non-basic variable, and c_i is a constant. A new objective function is obtained by substituting the basic variables with non-basic variables. We extract the indices of basic and non-basic variables from GLPK for reconstructing the rational results.

3.4.6 Reconstructing rational matrix and extracting result (Step 7-9)

We solved Problem 3.18 in floating-point numbers using GLPK. Had the solving been done in exact arithmetic, one could retain the optimal point $\boldsymbol{\lambda}^*$, but here we cannot use it directly. Instead, we exploit

the partition of the variables into basic and non-basic variables, and from this partition we can recompute exactly, in rational numbers, the optimal function, as well as a certificate that it is feasible.

Consider the Simplex tableau below in Table 3.1. Let M denote the matrix of constraints and O

	λ_1	$\cdots$	λ_m	= constant
$\vdots$	.....			
row j	a_{j1}	$\cdots$	a_{jm}	b_j
$\vdots$	.....			
objective	$\sum_{k=1}^{n} o_{1k}x_k + c_1$	$\cdots$	$\sum_{k=1}^{n} o_{mk}x_k + c_m$	$-Z^*(\boldsymbol{x}) = \sum_{k=1}^{n} z_k x_k + z_0$

Table 3.1 – Simplex tableau.

the matrix of the PLP objective function in Table 3.1. The ith column corresponds to the coefficients of ith variable, where the variables $\boldsymbol{\lambda} = [\lambda_1, \cdots, \lambda_m]$; the last column of the each matrix represents the constants.

$$M = \begin{bmatrix} a_{11} & \cdots & a_{1m} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nm} & b_n \end{bmatrix} \qquad O = \begin{bmatrix} o_{11} & \cdots & o_{m1} & z_1 \\ \vdots & \ddots & \vdots & \vdots \\ o_{1n} & \cdots & o_{mn} & z_n \\ c_1 & \cdots & c_m & z_0 \end{bmatrix} \tag{3.19}$$

To generalize the result of the LP, we need to reconstruct the matrices M and O associated to the basis B provided by GLPK. We extract the columns corresponding to the basic variables from matrices in Problem 3.19, and use M_B and O_B to denote the sub-matrices of M and O only containing these extracted columns. By linear algebra in rational arithmetic[3] we compute a matrix Θ, representing the substitution performed by the Simplex algorithm, which result in the coefficients of the objective function of the PLP problem being 0. We compute the matrix Θ by $\Theta = O_B M_B^{-1}$, where M_B^{-1} denotes the inverse of M_B (actually, we do not inverse that matrix but instead call a solver for linear equation systems). Then we apply this substitution to the objective matrix O to get the matrix of the new objective function O': $O' = O - \Theta M$. As $O_B = \Theta M_B$, by applying $O - \Theta M$ the columns corresponding to the basic variables become 0.

Detecting optimum in PLP We detect the optimal function and the corresponding region from the matrix O'. The variables of our PLP problem for computing projection have lower bound 0, which means that the objective function of the PLP problem reaches the optimal when all the non-basic variables reach their lower bound and their coefficients should be non-negative. Each non-zero column in O' represents

[3] We use Flint, which provides exact rational scalar, vector and matrix computations, including solving of linear systems. `http://www.flintlib.org/`

a function in $\boldsymbol{x}$, which is the coefficient of a non-basic variable, say λ_j. The conjunction of constraints $\bigwedge_{j \notin B} (O'_{\bullet j})^{\mathsf{T}} \boldsymbol{x} \geq 0$ defines a region of $\boldsymbol{x}$ where j belongs to the indices of non-basic variables. Note that the region itself is a polyhedron. Therefore, the conjunction of constraints may contain redundancy and we must apply the minimization procedure over it. The minimization algorithm produces irredundancy witness points of all the irredundant constraints, which are points located over each constraint. Thus they can be used for instantiating the PLP objective and exploring new part of the parameter space. We insert these witness points into the work list and use them to do the next explorations.

3.4.7 Checking adjacency regions (Step 10)

Once we found a new region, the witness points of this region are added into the work list as new task points. Then more regions will be explored by these tasks. But there is a subtlety: the region found by the witness point may not be adjacent to the original region, i.e. they do not share the same boundary (see Figure 3.12).

Adjacent regions Consider the example in Figure 3.12. Assume we found a new region $\mathcal{R}_1$, and one of its constraints is $\mathcal{C}_1$. The exploration using the witness point $\boldsymbol{w}_1$ of $\mathcal{C}_1$ finds a region $\mathcal{R}_2$ which is not adjacent to $\mathcal{R}_1$, i.e., there is a gap between the boundary of $\mathcal{C}_1$ belonging to $\mathcal{R}_1$ and that of $\mathcal{C}_2$ belonging to $\mathcal{R}_2$. It is possible that the witness point $\boldsymbol{w}_2$ of $\mathcal{C}_2$ will not find the region between $\mathcal{R}_1$ and $\mathcal{R}_2$ ($\boldsymbol{w}_2$ could be in $\mathcal{R}_1$ or beyond). Hence there is a risk of missing a region. To prevent missing regions, we need to check adjacency of regions once we found a new region. Considering the same example in Figure 3.12: once found the region $\mathcal{R}_2$ with the point $\boldsymbol{w}_1$, we check adjacency of $\mathcal{R}_1$ and $\mathcal{R}_2$. The adjacency test is designed to provide a point $\boldsymbol{w}'$ between the two regions if adjacency does not hold. Thus $\boldsymbol{w}'$ is inserted into the work list for exploring the missing region between $\mathcal{R}_1$ and $\mathcal{R}_2$.

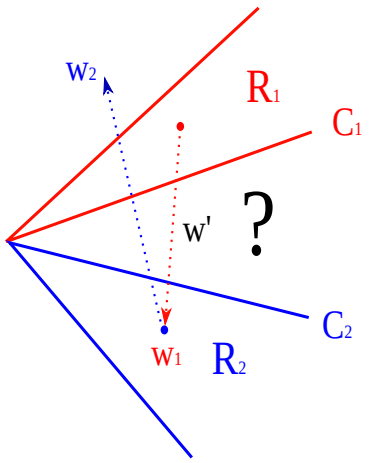


Figure 3.12 – An example of missing regions

The method to check adjacency To check adjacency we compare each constraint of the two regions syntactically in rational numbers. We can use a syntactic test thanks to the normalization constraint (*) of Problem 3.9. If two regions have a common boundary, they are likely to be adjacent, but this is not a sufficient condition. We need to consider the situation in Figure 3.13, in which the constraints $\mathcal{C}_1$ and $\mathcal{C}_2$ have a common boundary, which is denoted by F, but their regions $\mathcal{R}_1$ and $\mathcal{R}_2$ are not adjacent. To detect this situation we need an additional test. If two regions are adjacent, they should have the same optimal value along the direction of the common boundary. We choose a point randomly on the boundary F, and evaluate the optimal functions of $\mathcal{R}_1$ and $\mathcal{R}_2$ with this point. If we obtain the same value, then the two regions are adjacent. Recall that the boundary F of a constraint $\boldsymbol{ox}+c \geq 0$ is an equation $\boldsymbol{ox}+c=0$. To get a point $\boldsymbol{p}$ on the boundary F, we choose a coefficient o_i which is not 0, and set the other components to zero. Then by solving $o_1 \times 0 + \cdots + o_i p_i + \cdots + o_n \times 0 + c = 0$, we obtain $p_i = -\frac{c}{o_i}$. The point $\boldsymbol{p} = (0, \cdots, -\frac{c}{o_i}, \cdots, 0)$ is on the boundary. As we said, if $Z_1^*(\boldsymbol{p}) = Z_2^*(\boldsymbol{p})$ the two regions $\mathcal{R}_1$ and $\mathcal{R}_2$ are adjacent.

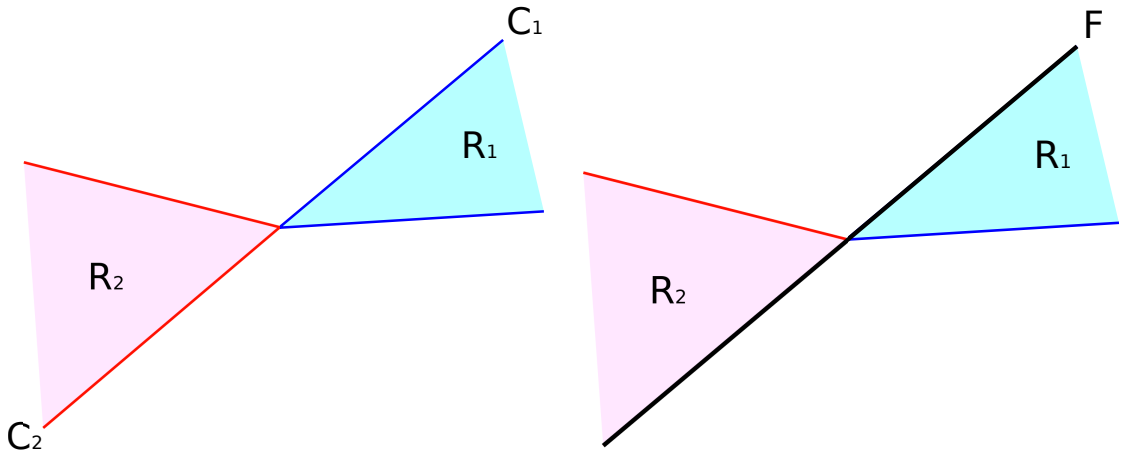


Figure 3.13 – Limitation of syntactic adjacency checking

Finding a point between non-adjacent regions When the previous test reports that regions $\mathcal{R}_1$ and $\mathcal{R}_2$ are not adjacent, we compute an additional point between them, as it is shown in Figure 3.14. We compute the length l_0 between the two points $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$. The length from $\boldsymbol{w}_1$ to the nearest frontier of $\mathcal{C}_1$ is l_1 and that from $\boldsymbol{w}_2$ to the frontier of $\mathcal{C}_2$ is l_2. The segment from $\boldsymbol{w}_1$ to $\boldsymbol{w}_2$ intersects the boundaries of $\mathcal{C}_1$ and $\mathcal{C}_2$ at w_1' and w_2' respectively. The new instantiation point $\boldsymbol{w}_3$ is the middle of the segment $w_1'w_2'$. The adjacency test is performed in rational computation. The new point $\boldsymbol{w}_3$ is computed in floating point arithmetic, as $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ are in floating point numbers.

Generally speaking, we do not need to consider the imprecision of floating point arithmetic here, as the point $\boldsymbol{w}_3$ can locate at any place inside the missed region. What matters is that $\boldsymbol{w}_3$ should not be too closed to the boundary, otherwise the exploration of regions may loop forever. Considering the same example in Figure 3.14, if the added point $\boldsymbol{w}_3$ in $\mathcal{R}_3$ is extremely closed to $\mathcal{C}_1$, it may find the region $\mathcal{R}_1$ (although the algorithm will terminate once found that the basis has already been found). Then $\mathcal{R}_1$ is

checked to be not adjacent to $\mathcal{R}_2$, and the same point would be added. Repeatedly $\mathcal{R}_1$ would be found, and so on. To avoid this situation, we check $l_0 > l_1 + l_2 + t$, where $t = \varepsilon \max(|l_0|, |l_1|, |l_2|)$. If this test fails, we discard the point $\boldsymbol{w}'$, and rely on a rational checker to make up the missed region (Chapter 4.4). Note that missing a region cannot break the soundness of the result.

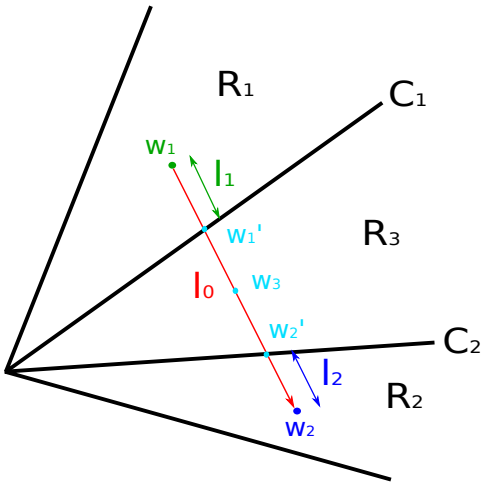


Figure 3.14 – Add point between regions

Algorithm 2: Sequential algorithm of PLP solver for computing projection.

Input: $poly^{\mathbb{Q}}$: the polyhedron to be projected
$[x_p, ..., x_q]$: the variables to be eliminated
n: number of initial points
Output: ($optimums^{\mathbb{Q}}$,$regions^{\mathbb{Q}\times\mathbb{F}}$) the set of optimal functions with their regions
Function Plp($poly^{\mathbb{Q}}$, $[x_p, ..., x_q]$, n)

```
/* convert the polyhedron to be projected which is stored in rational numbers into
   floaing point numbers */
poly^F = GetFloat(poly^Q)
// minimize the polyhedron to be projected
irredundantIdx = Minimize(poly^F)
minimizedPoly^{Q×F} = GetSubPoly(poly^Q, irredundantIdx)
// construct the PLP problem for computing projection
plp^{Q×F} = ConstructPlp(minimizedPoly^{Q×F}, [x_p, ..., x_q])
// take any point different from the normalization point as an initial task point
worklist^F = GetInitialTasks(minimizedPoly^Q, n)
(optimums^Q,regions^{Q×F}) = none
while worklist^F ≠ none do
    // take a task in the work list
    (w^F, R_from^Q, F_from) = GetTask(worklist^F)
    /* R_curr^Q is none if w^F is not inside any known region; otherwise R_curr^Q is the
       region w^F located in */
    R_curr^Q = CheckCovered(regions^F, w^F)
    // compute a new region with w^F
    if R_curr^Q == none then
        /* set the parameters to w^F, use GLPK to solve the obtained LP problem, and
           obtain the basic and nonbasic variables from GLPK */
        (basicIndices, nonbasicIndices) = GlpkSolveLp(w^F, plp^F)
        // reconstruct the PLP problem for obtaining rational result
        reconstructMatrix^Q = Reconstruct(plp^Q, basicIndices)
        (newOptimum^Q, newRegion^{Q×F}) = ExtractResult(reconstructMatrix^Q,
          nonbasicIndices)
        // minimize the new region using floating point arithmetic
        (irredundantIdx, witnessList^F) = Minimize(newRegion^F)
        /* obtain minimized rational region by extracting the irredundant constraints
           from the rational region */
        minimizedR^Q = GetRational(newRegion^Q, irredundantIdx)
        // store the result
        Insert((optimums^Q,regions^{Q×F}), (newOptimum^Q, minimizedR^{Q×F}))
        // add the irredundancy witness points into the work list as new task points
        AddWitnessPoints(witnessList^F, worklist)
        R_curr^Q=minimizedR^Q
    /* if the the two regions are not adjacent, add an extra task point between them;
       otherwise store them as adjacent regions */
    if AreAdjacent(R_curr^Q, R_from^Q, F_from) then
        F_curr = GetCrossBoundary(R_curr^Q, R_from^Q, F_from)
        StoreAdjacencyInfo(R_from^Q, F_from, R_curr^Q, F_curr)
    else
        AddExtraPoint(worklist, R_curr^Q, R_from^Q)
```

3.5 Checkers and Rational Solvers

We implemented most part of our procedure in floating-point numbers for the search of a solution followed by a reconstruction of the exact solution using efficient algorithm of linear algebra. But there are several cases where we should switch to exact checkers (and solvers if necessary) during the search in order to ensure the soundness and precision of the result. In the following sections we will address these rational processes.

3.5.1 Detecting flat regions

Our regions are obtained by reconstructing the rational matrix of the PLP objective function. They are converted into floating-point representation when the solver builds an LP problem for GLPK by instantiating the parameters in the PLP objective with a point (in floating point representation) picked up in the work list. As the regions are normalized and intersect at the same point, they are in the shape of cones. During the conversion, the constraints will lose accuracy, and thus a cone could be misjudged as flat, meaning it has empty interior. For instance, we have a cone $\{\mathcal{C}_1 : -\frac{100000001}{10000000}x_1 + x_2 \leq 0,\ \mathcal{C}_2 : \frac{100000000}{10000000}x_1 - x_2 \leq 0\}$, which is not flat. After conversion, $\mathcal{C}_1$ and $\mathcal{C}_2$ will be represented in floating-point numbers as $\{\mathcal{C}_1 : -10.0x_1 + x_2 \leq 0,\ \mathcal{C}_2 : 10.0x_1 - x_2 \leq 0\}$, and the floating-point cone is flat. Thus the flat region will be discarded which results in imprecision of the result.

Our algorithm of raytracing minimization will report that the region is flat if it cannot find a point inside the floating point region. In this case we need to invoke a rational checker to test if the region is really flat. In The rational checker, all the constraints are shifted to the direction in the interior of the region. As the region is a cone, the constraints of a flat region will become infeasible after shifting, and a non-flat region will still be a cone. Thus we invoke our pure rational Simplex solver to test the satisfiability of the shifted constraints. If the shifted constraints becomes infeasible, i.e., the region is flat, we launch a rational minimization algorithm, which is implemented using Farkas' lemma, to obtain the minimized region.

3.5.2 Verifying feasibility of the result provided by GLPK

GLPK uses a threshold (10^{-7} by default) when checking the feasibility of the solution. It may thus report a feasible result when the problem is in fact infeasible. Assume that we have an LP problem whose constraints are $C_1 : \lambda_1 \geq 0, C_2 : \lambda_2 \geq 0, C_3 : \lambda_1 + \lambda_2 \leq 10^{-8}$, GLPK will obtain $(\lambda_1, \lambda_2) = (0,0)$ as a solution, whereas this point does not fulfill the constraints.

A solution is feasible if all the nonbasic variables reach their lower bound 0, and the basic variables fulfill their bound $\boldsymbol{\lambda} \geq 0$. That means when the solution is infeasible, at least one of the basic variable is negative, i.e., $\exists \lambda_i \in B,\ \lambda_i < 0$, where B is the basis provided by GLPK. In this case the objective function of the PLP is not a non-negative combination of the constraints of the polyhedron to be projected, which results in an unsound result (see Example 3.9). Therefore we must test the feasibility of the result provided by GLPK in rational numbers.

Example 3.9. Consider the example in Figure 3.15: we have a polyhedron $\mathcal{P} = \{x_1 + x_2 - 5 \geq 0,\ -x_1 + x_2 + 3 \geq 0,\ x_1 - x_2 + 2 \geq 0,\ -x_1 - x_2 + 8 \geq 0\}$, and we want to eliminate x_1. The PLP problem for computing this projection is shown below.

$$\begin{aligned} \text{minimize} \quad & \lambda_1(x_1 + x_2 - 5) + \lambda_2(-x_1 + x_2 + 3) + \lambda_3(x_1 - x_2 + 2) + \lambda_4(-x_1 - x_2 + 8) + \lambda_0 \\ \text{subject to} \quad & \lambda_1 + 3\lambda_2 + 2\lambda_3 + 2\lambda_4 + \lambda_0 = 1 \\ & \lambda_1 - \lambda_2 + \lambda_3 - \lambda_4 = 0 \\ \text{and} \quad & \lambda_i \geq 0 \quad (\forall i \in \{0, \cdots, 4\}) \end{aligned} \tag{3.20}$$

The expected projected polyhedron is $\mathcal{P}' = \{x_2 \geq 1,\ x_2 \leq 5\}$. If we choose $\lambda_1,\ \lambda_3$ as basic variables, we will have $\lambda_1 = 5\lambda_2 + 4\lambda_4 + \lambda_0 - 1$, and $\lambda_3 = -4\lambda_2 - 3\lambda_4 - \lambda_0 + 1$. By setting $\lambda_2,\ \lambda_4,\ \lambda_0$ to 0, we obtain $\lambda_1 = -1$, and thus the basis $\lambda_1,\ \lambda_3$ leads the LP problem to infeasibility. If we use the infeasible basis $\lambda_1,\ \lambda_3$ to reconstruct the objective function, we will obtain a new objective function:

$$\lambda_2(10x_2 - 30) + \lambda_4(6x_2 - 18) + \lambda_0(2x_2 - 7) + (-2x_2 + 7)$$

from which we can obtain a face $x_2 \leq \frac{7}{2}$. This face narrows the expected polyhedron $\mathcal{P}'$ to $\{x_2 \geq 1,\ x_2 \geq \frac{7}{2}\}$, and thus the obtained polyhedron is not an over-approximation of the correct polyhedron. We end up with an unsound result.

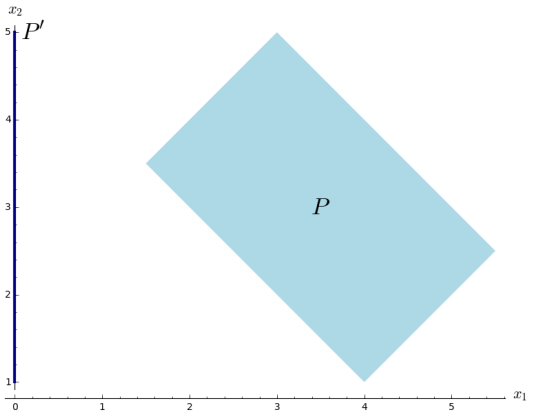


Figure 3.15 – The expected projected polyhedron by projecting out x_1 of $\mathcal{P}$ is $\mathcal{P}'$.

We use the basis B for reconstructing the matrix A of the constraints $A\boldsymbol{\lambda} = \boldsymbol{b}$ such that the basic variables have coefficient 1, and obtain the new constraints $A'\boldsymbol{\lambda} = \boldsymbol{b}'$. When the LP problem reaches an optimum, the non-basic variables are at their lower bound 0, so the value of the basic variables are just the value of $\boldsymbol{b}'$. We just need to verify that all coordinates in $\boldsymbol{b}'$ are non-negative. If it is not in this case, GLPK must encounter a precision problem. Then we call our Simplex algorithm in rational arithmetic which follows the standard textbook implementation. But it is not pure-rational. We construct the objective function of the LP problem with a point in floating point numbers, and thus we still use

floating-point numbers for checking if all the coefficients of non-basic variables in the objective function are non-negative, i.e. checking if $o_j \geq \varepsilon$, where o_j is the floating-point coefficient of the objective function and ε is the machine epsilon. But we do the pivoting in the Simplex tableau in rational numbers. This mixing of rational and floating point computations may provide a fake optimal solution, i.e., a solution which is feasible but not optimized, but in the context of projection and convex hull of polyhedra for static analysis it will not lead to unsoundness. A fake optimal solution may lead to less precise solution, i.e., some faces are missing, but our adjacency checker (see Section 4.4) will reconstruct all the missing faces.

Algorithm 3: Rational feasibility checker and solver.

```
Input: plp^Q: the PLP problem to be solved
       w^F: the current task point
       basicIndices: the indices of basic variables obtained from GLPK
Function FeasibilityChecker(plp^Q, basis)
    reconstructMatrix^Q = Reconstruct(plp^Q, basicIndices)
    feasible = IsFeasible(reconstructMatrix^Q)
    /* if GLPK returns an infeasible solution, use a rational solver to solve the LP
       problem */
    if feasible == false then
        // try to get a feasible solution
        feasibleBasis = RationalSimplexPhase1(plp^Q)
        if feasibleBasis == none then
            report infeasible
        // test if the solution is optimal
        currObjective^F = GetObj(w^F, plp^Q, feasibleBasis)
        optimal = IsOptimal(currObjective^F)
        // compute an optimal solution
        while optimal == false do
            enteringIdx = GetEnteringVar(currObjective^F)
            newBasis = RationalSimplexPivot(plp^Q, enteringIdx)
            currObjective^F = GetObj(w^F, plp^Q, feasibleBasis)
            optimal = IsOptimal(currObjective^F)
        return newBasis
    else
        return basis
```

3.5.3 Obtaining sound and exact solution

We presented the rational checkers and solvers. In this section we summarize the mean for obtaining a sound and exact solution. We consider in turn all the processes (1 to 8) of our PLP solver depicted in the flowchart of Figure 3.1 and for each one we study the potential impact of floating point computations on the exact solution.

The result should always be sound if all the variables fulfill $\boldsymbol{\lambda} \geq 0$, i.e., if the PLP problem is feasible, as the objective is a non-negative combination of the constraints of the polyhedron to be projected. As we explained in Section 3.5.2, we used a rational checker (and a rational solver) to ensure the feasibility of the PLP, the result must be sound.

We have explained in Section 3.2.3 that in raytracing minimization all the constraints which cannot be determined by the floating point solver are reported as redundant, and meanwhile they are added into a undetermined list. As the regions are stored in both floating point and rational numbers, all the constraints with an undetermined status (redundant or irredundant) can be resolved using rational solver which looks for the existence of a Farkas' combination expressing the undetermined constraint. So the results of processes 1 (minimize the input polyhedron using raytracing minimization) and 8 (minimize the region using raytracing minimization) can be fixed by rational solvers and exact solutions will always be produced.

The process 2 (add initial instantiation point of parameters into the work list) does not affect the result, as any point in the parameter space can be chosen for the initial exploration. We explained in Section 3.4.4 that the process 3 (test if the point belongs to a known region) will not discard any instantiation point, as a point which is not covered by any region can never be reported to be covered. Thus the process 3 cannot lead to impression.

Considering the processes 4 (construct an LP problem with the instantiation point) and 5 (solve the LP problem with GLPK), in Section 3.5.2 we explained that GLPK may obtain a solution which is infeasible, and thus we need to verify the feasibility of the result provided by GLPK. GLPK may also report a solution which is not optimal for the given direction but optimal for a slightly different LP problem. We did not provide a checker for this situation, as in any case we will be able to use the returned basis to construct a region. This step may lead to missing regions, but they can be reconstructed by a rational procedure (Section 4.4).

The process 6 (add an instantiation point between the two regions) may discard some points in extreme cases (explained in Section 3.4.7), and this may cause the miss of regions. But the missing regions will be made up by the adjacency checker at the end, as what we presented in Section 4.4.

The process 7 (check if a region is flat in floating point numbers) has been presented in Section 3.5.1: a rational checker (and a rational solver) will be launched when the floating point solver reports that the region is flat.

As a summary, all the processes running in floating point arithmetic and having an impact on soundness and precision are verified by rational checkers and the result is repaired by rational processes when it is necessary. The adjacency checker at the end guarantees completeness of the exploration. Hence our approach is able to obtain the same solution as a pure rational approach.

3.6 Dealing with Equalities

Until now all the constraints we mentioned are inequalities. In this section we will talk about how to deal with equalities.

3.6.1 Minimization with equalities

Initially the equalities and inequalities are divided and stored separately. We choose a variable in each equality and substitute it in the inequalities and equalities by its definition in terms of other variables. Then replace these selected variables in the inequalities by the others.

Example 3.10. Given the polyhedron $\mathcal{P} = \{3x_1 + x_2 + 2x_4 + 4x_5 \geq 5,\ -x_1 + x_3 + 4x_4 + 2x_5 \geq -2,\ 6x_1 - 2x_3 = 5,\ 8x_1 - 3x_2 = 2,\ -2x_1 + 3x_2 - 2x_3 = 3\}$, the constraints are divided into

$$\mathcal{P}_{eqn} = \{6x_1 - 2x_3 = 5, 8x_1 - 3x_2 = 2, -2x_1 + 3x_2 - 2x_3 = 3\}$$
$$\mathcal{P}_{ineq} = \{3x_1 + x_2 + 2x_4 + 4x_5 \geq 5, -x_1 + x_3 + 4x_4 + 2x_5 \geq -2\}$$

As $x_1 = \frac{1}{3}x_3 + \frac{5}{6}$, we can obtain

$$\begin{aligned} x_1 &= \frac{1}{3}x_3 + \frac{5}{6} \\ -3x_2 + \frac{8}{3}x_3 &= -\frac{14}{3} \\ 3x_2 - \frac{8}{3}x_3 &= \frac{14}{3} \end{aligned}$$

As $x_2 = \frac{8}{9}x_3 + \frac{14}{9}$, we have

$$\begin{aligned} x_1 &= \frac{1}{3}x_3 + \frac{5}{6} \\ x_2 &= \frac{8}{9}x_3 + \frac{14}{9} \end{aligned} \tag{3.21}$$

The third equality becomes $\frac{14}{3} = \frac{14}{3}$, so it is eliminated. The Equation 3.21 shows the equalities without redundancy.

Now we substitute x_1 and x_2 with x_3 in the inequalities. The obtained constraints are shown in Equation 3.22.

$$\begin{aligned} 34x_3 + 36x_4 + 72x_5 &\geq 17 \\ 4x_3 + 24x_4 + 12x_5 &\geq -7 \end{aligned} \tag{3.22}$$

Then we minimize the inequalities using either method presented in Section 3.2. The minimized polyhedron is: $\mathcal{P}' = \{6x_1 - 2x_3 = 5,\ 9x_2 - 8x_3 = 14,\ 34x_3 + 36x_4 + 72x_5 \geq 17,\ 4x_3 + 24x_4 + 12x_5 \geq -7\}$.

3.6.2 Projection of a polyhedron with equalities

Before computing projection of the polyhedron, we select variables defined by the equalities among the ones to be eliminated. Then we apply the projection operators to the equalities and inequalities

separately. The variable x_e to be eliminated no more appears in the inequalities and its value is defined by an equality in terms of other variables. Thus this equality does not restrict the projected polyhedron and it can simply be removed.

Example 3.11. Consider the polyhedron $\mathcal{P} = \{x_1 - x_2 = 0, x_1 + x_3 \geq 5\}$. If we want to eliminate x_1, then the polyhedron will be transformed into $\mathcal{P} = \{x_1 = x_2, x_2 + x_3 \geq 5\}$. The variable x_1 no more appears in the inequalities, and only in equality $x_1 = x_2$. The projected polyhedron is then $\mathcal{P} \backslash \{x_1\} = \{x_2 + x_3 \geq 5\}$.

3.6.3 Convex hull of polyhedra with equalities

To compute the convex hull of two polyhedra using PLP, we need to pick up an interior point for minimization in one of the polyhedra. In the case where both polyhedra contain equalities (thus they both have empty interior), it is impossible to find such an interior point.

One strategy to solve this problem is that instead of selecting a point inside one of the polyhedra, we can find a point in the interior of their convex hull, as long as the convex hull has non-empty interior. By the definition of the convex hull $\mathcal{P}_1 \sqcup \mathcal{P}_2 = \{\boldsymbol{x} | \boldsymbol{x} = \alpha \boldsymbol{x}' + (1-\alpha)\boldsymbol{x}'',\ \boldsymbol{x}' \in \mathcal{P}_1,\ \boldsymbol{x}'' \in \mathcal{P}_2,\ 0 \leq \alpha \leq 1\}$, we can choose a point $\boldsymbol{x}'$ inside $\mathcal{P}_1$ and another point $\boldsymbol{x}''$ inside $\mathcal{P}_2$ for computing a point $\boldsymbol{x}$ inside their convex hull.

Another strategy to deal with this situation is to compute the convex hull via projection. Benoy et al. presented in [32] a method to get the convex hull by computing a polyhedral projection. Recall the convex hull of two polyhedra $\mathcal{P}_1$ and $\mathcal{P}_2$:

$$\begin{aligned} \mathcal{P}_1 \sqcup \mathcal{P}_2 = \{\boldsymbol{x} | \boldsymbol{x} = \alpha \boldsymbol{x}' + (1-\alpha)\boldsymbol{x}'',\ A'_I \boldsymbol{x}' + \boldsymbol{b}'_I \geq 0,\ A'_E \boldsymbol{x}' + \boldsymbol{b}'_E = 0, \\ A''_I \boldsymbol{x}'' + \boldsymbol{b}''_I \geq 0,\ A''_E \boldsymbol{x}'' + \boldsymbol{b}''_E = 0,\ 0 \leq \alpha \leq 1\} \end{aligned} \tag{3.23}$$

To obtain the convex hull, we need to eliminate $\boldsymbol{x}$, $\boldsymbol{x}'$ and α. As the equation $\boldsymbol{x} = \alpha \boldsymbol{x}' + (1-\alpha)\boldsymbol{x}''$ is not linear, we need to change variables. By introducing $\boldsymbol{y}' = \alpha \boldsymbol{x}'$ and $\boldsymbol{y}'' = (1-\alpha)\boldsymbol{x}''$, we obtain Equation 3.24.

$$\begin{aligned} \mathcal{P}_1 \sqcup \mathcal{P}_2 = \{\boldsymbol{x} | \boldsymbol{x} = \boldsymbol{y}' + \boldsymbol{y}'',\ A'_I \boldsymbol{y}' + \alpha \boldsymbol{b}'_I \geq 0,\ A'_E \boldsymbol{y}' + \alpha \boldsymbol{b}'_E = 0, \\ A''_I \boldsymbol{y}'' + (1-\alpha)\boldsymbol{b}''_I \geq 0,\ A''_E \boldsymbol{y}'' + (1-\alpha)\boldsymbol{b}''_E = 0,\ 0 \leq \alpha \leq 1\} \end{aligned} \tag{3.24}$$

By substituting $\boldsymbol{y}'' = \boldsymbol{x} - \boldsymbol{y}'$ into the inequalities, we obtain the encoding of convex hull which is shown in Equation 3.25.

$$\begin{aligned} \mathcal{P}_1 \sqcup \mathcal{P}_2 = \{\boldsymbol{x} | A'_I \boldsymbol{y}' + \alpha \boldsymbol{b}'_I \geq 0,\ A'_E \boldsymbol{y}' + \alpha \boldsymbol{b}'_E = 0,\ A''_I (\boldsymbol{x} - \boldsymbol{y}') + (1-\alpha)\boldsymbol{b}''_I \geq 0, \\ A''_E (\boldsymbol{x} - \boldsymbol{y}') + (1-\alpha)\boldsymbol{b}''_E = 0,\ 0 \leq \alpha \leq 1\} \end{aligned} \tag{3.25}$$

To obtain the polyhedron which represents the convex hull, i.e. the set of $\boldsymbol{x}$, we need to eliminate $\boldsymbol{y}'$ and α.

Summary of the chapter

In this chapter we presented our sequential algorithm of PLP solver with two minimization methods. We mainly used the floating point raytracing minimization. When the floating point computation cannot determine the redundancy status of any constraint, the rational minimization implemented using Farkas' lemma will be invoked. Because of the rounding errors of the floating point arithmetic, we did some adaptation to the raytracing minimization for guaranteeing termination of the algorithm and soundness of the result, i.e., the resulting polyhedron is an over-approximation of the correct one.

We illustrated the polyhedral projection and convex hull geometrically, and also explained the way to compute them using PLP. We used GLPK to solve the LP problems in floating point numbers. To reconcile our approach with GLPK, we need to simplify the equality constraints of PLP into inequalities. In each step of the PLP solver, we must consider the impact of floating point arithmetic. Due to the floating point computations, the results may be unsound or imprecise. The result is unsound if GLPK obtain a reports an infeasible solution as a feasible one. In other words, GLPK obtain an infeasible basis but returns it as a feasible one, which means that the real feasible basis has not been found. Thus we provided a checker for testing the feasibility of the result obtained from GLPK. The checker for guaranteeing the precision will be presented in Section 4.4. The focus of our PLP solver is on computing the polyhedra with inequalities. As we explained, the equalities need to be dealt with separately.

Chapter 4

Degeneracy and Overlapping Regions

Degeneracy [33, 25, 28, 34] is a difficulty of our approach. It is a general issue in LP which may lead to the cycling of the Simplex algorithm, and there are mature solutions to deal with it. In PLP it leads to overlapping regions. In this chapter we present the fundamental of degeneracy, the impact on LP and PLP, and the methods to handle it.

Base on the work in [39] we implement an algorithm to deal with the degeneracy in our PLP solver, and thus the overlapping regions can be avoided. Once the overlapping regions are avoided, we can check the precision of the resulting polyhedron. If there are missing regions, it means that the floating point computation is not precise enough, and we use a rational procedure to compute these regions.

4.1 Introduction to degeneracy

There are two kinds of degeneracy. The primal degeneracy [33, 28] means that the unique optimum can be expressed by several different bases, and the dual degeneracy [25, 34] happens when there exist more than one optimums.

Now let us consider the example shown in Figure 4.1. The pyramid in 3 dimensions is made of 5 constraints and 4 of them intersect at the top point.

4.1.1 Primal degeneracy

Let us compute the optimum in the direction toward the top of the pyramid shown in Figure 4.1: the optimal solution is given by the top vertex.

Once the system is rewritten as equality constraints by introducing slack variables. The number of basic variables, defined in terms of the others, corresponds to the number of equalities. The original variables are nonbasic variables, and thus the number of non-basic variables equals the number of dimensions of the constraints $\bigwedge_{i=1}^{m} \boldsymbol{a}_i \boldsymbol{x} \leq b_i$, which is 3 in this example. A vertex in 3 dimensions is specified by 3 faces. That means 3 of the constraints associated to the nonbasic variables intersect at the optimal vertex, and the non-basic variables reach their lower bound (for minimization problems), which is 0 in our case. But in this example, a fourth constraint also intersects at the same vertex, which

makes the corresponding basic variable equal to 0. In this situation the vertex could be specified in 4 different ways, i.e., we may have 4 bases if any three constraints of the four could be chosen as non-basic variables. The optimal vertex remains the same, but the partition of basic and nonbasic variables varies. This is a case of primal degeneracy.

Let us take Example 4.1 to figure out how primal degeneracy occurs during computations. We will see that given a entering variable, there could be multiple choices of leaving variables as their ratios (the constant over the coefficient) are the same. We choose one of them to leave the basis, and in the next pivoting the constants corresponding to the remaining candidates will become zero.

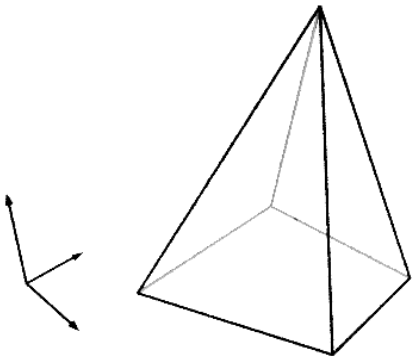


Figure 4.1 – Degeneracy

Example 4.1. Consider the example in Figure 4.1, in which the polyhedron is $\mathcal{P} = \{x_3 \geq 0,\ 2x_1 - x_3 \geq 2,\ -2x_1 + 4x_2 - 3x_3 \geq 2,\ -2x_1 - x_3 \geq -6,\ -2x_2 - x_3 \geq -8\}$, and we compute the maximal value in the direction $(0,0,1)$. The LP problem is shown in Problem 4.1.

$$\begin{aligned} \text{maximize} \quad & x_3 \\ \text{subject to} \quad & 2x_1 - x_3 \geq 2 \\ & -2x_1 + 4x_2 - 3x_3 \geq 2 \\ & -2x_1 - x_3 \geq -6 \\ & -2x_2 - x_3 \geq -8 \\ \text{and} \quad & x_1,\ x_2,\ x_3 \geq 0 \end{aligned} \tag{4.1}$$

We transform this problem into the standard form by adding slack variables, and obtain Problem 4.2.

$$\begin{aligned} \text{maximize} \quad & x_3 \\ \text{subject to} \quad & -2x_1 + x_3 + x_4 = -2 \\ & 2x_1 - 4x_2 + 3x_3 + x_5 = -2 \\ & 2x_1 + x_3 + x_6 = 6 \\ & 2x_2 + x_3 + x_7 = 8 \\ \text{and} \quad & x_i \geq 0 \quad \forall i \in \{1,2,3,4,5,6,7\} \end{aligned} \tag{4.2}$$

We solve this LP problem by Simplex algorithm. Let us use the slack variables x_4, x_5, x_6, x_7 as the initial basic variables. We can see that the problem is infeasible currently, and we need to find a feasible dictionary by applying the first phase of Simplex algorithm.

After several pivoting, we obtained the feasible dictionary with basic variable $x_1,\ x_2,\ x_6,\ x_7$, and the corresponding Simplex tableau is shown in Table 4.1. Now we start the second phase of the Simplex algorithm. We choose x_3 as the entering variable, as its coefficient in the objective function is positive. We have two choice of leaving variable, as both row 3 and row 4 bound x_3 by 2. The primal degeneracy occurs as we have multiple choices of leaving variable. We choose x_6 to leave the basis. By doing one pivoting, we obtain Table 4.2. The obtained objective function does not contain positive coefficient, so the problem is maximized, and the optimal value is 2. The basic variables are $x_1,\ x_2,\ x_3,\ x_7$. By setting the nonbasic variables to 0, we obtain the value of basic variables, and the optimal vertex is $(x_1,x_2,x_3,x_4,x_5,x_6,x_7) = (2,3,2,0,0,0,0)$. Geometrically the optimal point is $(2,3,2)$ (Figure 4.2(a)). In Table 4.2 x_7 is a basic variable, but its value is 0. As we said, the degeneracy occurs when there are basic variables have value 0. Indeed exchanging the basic variable x_7 with any of the nonbasic variables $x_4,\ x_5,\ x_6$ will bound x_8 to the same value 0, and thus the value of the objective function will remain the same.

If we choose x_7 as the leaving variable instead of x_6, the optimal value and the optimal vertex will be the same, but the basis will become $x_1,\ x_2,\ x_3,\ x_6$. If we do one more pivoting, another basis $x_1,\ x_2,\ x_3,\ x_5$ at the same optimal vertex will be found. Another possible construction of the basis is $x_1,\ x_2,\ x_3,\ x_4$. We force x_4 to enter the basis, and obtain Table 4.3. However, although it leads to the optimal value 2, the objective function contains a positive coefficient. Thus the Simplex algorithm believes that the LP did not reach the optimum. We obtained 3 possible bases which lead the LP problem to the optimum.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	= constant
row 1	1	0	$-\frac{1}{2}$	$-\frac{1}{2}$	0	0	0	1
row 2	0	1	-1	$-\frac{1}{4}$	$-\frac{1}{4}$	0	0	1
row 3	0	0	2	1	0	1	0	4
row 4	0	0	3	$\frac{1}{2}$	$\frac{1}{2}$	0	1	6
objective	0	0	1	0	0	0	0	0

Table 4.1

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	= constant
row 1	1	0	0	-$\frac{1}{4}$	0	$\frac{1}{4}$	0	2
row 2	0	1	0	$\frac{1}{4}$	$-\frac{1}{4}$	$\frac{1}{2}$	0	3
row 3	0	0	1	$\frac{1}{2}$	0	$\frac{1}{2}$	0	2
row 4	0	0	0	-1	$\frac{1}{2}$	$-\frac{3}{2}$	1	0
objective	0	0	0	$-\frac{1}{2}$	0	$-\frac{1}{2}$	0	-2

Table 4.2

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	= constant
row 1	1	0	0	0	$-\frac{1}{8}$	$\frac{5}{8}$	$-\frac{1}{4}$	2
row 2	0	1	0	0	$-\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{4}$	3
row 3	0	0	1	0	$\frac{1}{4}$	$-\frac{1}{4}$	$\frac{1}{2}$	2
row 4	0	0	0	1	$-\frac{1}{2}$	$\frac{3}{2}$	-1	0
objective	0	0	0	0	$-\frac{1}{4}$	$\frac{1}{4}$	$-\frac{1}{2}$	-2

Table 4.3

4.1.2 Dual degeneracy

When dual degeneracy occurs, the objective function reaches the same optimum at multiple vertices. The optimal value remains the same, but the optimal vertices are different. Let us illustrate the dual degeneracy on Example 4.2.

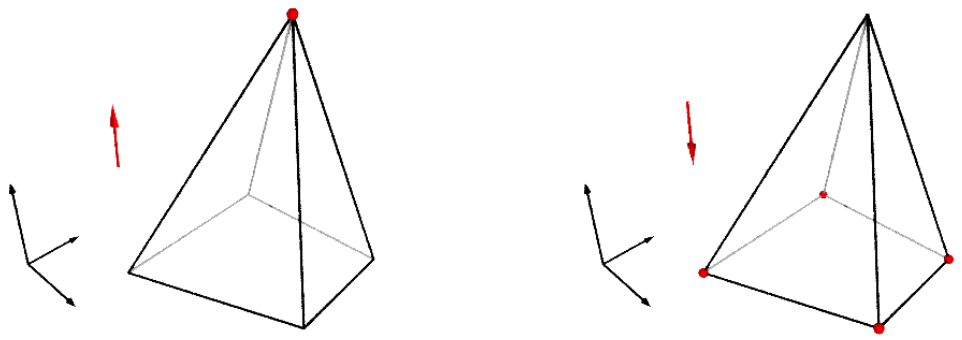
Example 4.2. Considering the polyhedron in Figure 4.1, let us compute the maximization in the direction $-x_3$. By applying the first phase and several pivoting in the second phase of the Simplex algorithm, we obtain Table 4.4, in which the basis is x_1, x_2, x_6, x_7, and the corresponding optimal vertex is $(x_1,x_2,x_3,x_4,x_5,x_6,x_7) = (1,1,0,0,0,0,4,6)$.

We could also obtain three other optimal bases: x_1, x_2, x_4, x_7 with optimal vertex $(3,2,0,2,0,-2,4)$, x_1, x_2, x_4, x_5 with optimal vertex $(3,4,0,4,8,0,0)$ and x_1, x_2, x_5, x_6 with optimal vertex $(1,4,0,0,12,4,0)$. In all these situations, the optimal value are the same which is 0, but the vertices are different. That is the difference from the primal degeneracy.

Geometrically we could obtain the four vertices at the bottom in Figure 4.1, which are $(1,1,0)$, $(3,2,0)$, $(3,4,0)$ and $(1,4,0)$ (Figure 4.2(b)).

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	= constant
row 1	1	0	$-\frac{1}{2}$	$-\frac{1}{2}$	0	0	0	1
row 2	0	1	-1	$-\frac{1}{4}$	$-\frac{1}{4}$	0	0	1
row 3	0	0	2	1	0	1	0	4
row 4	0	0	3	$\frac{1}{2}$	$\frac{1}{2}$	0	1	6
objective	0	0	-1	0	0	0	0	0

Table 4.4



(a) Primal degeneracy: one optimum, one optimal vertex with 4 possible bases.

(b) Dual degeneracy: one optimum, 4 possible vertices and one basis for each of them.

Figure 4.2 – Optimal vertices.

4.2 Degeneracy in linear programming

The problem of degeneracy in LP has been known for decades. In this section we talk about the primal degeneracy firstly, and then the dual degeneracy. Several methods to deal with the primal degeneracy for avoiding cycling will be introduced.

4.2.1 Primal degeneracy

The occurrence of primal degeneracy leads to inefficiency of the algorithm, and moreover it may cause the Simplex algorithm to cycle. Many methods have been proposed to avoid cycling, for example Bland's rule [35] and the perturbation method [36].

Bland's rule guarantees the termination of the Simplex algorithm by selecting the entering and leaving variables which have the smallest indices. Bland's rule is easy to implement but the value of objective function increases slowly.

> **Theorem 4.1** (Bland's rule)**.** Among all candidates to enter the basis, select the variable having the lowest index; among all candidates to leave the basis, select the variable having the lowest index.

Another well-known method is the perturbation method. To avoid multiple choices of the leaving variable, this method perturbs the constraints by adding a small value ε_i to each constraint C_i and make sure that $\varepsilon_1 \gg \varepsilon_2 \gg \cdots \gg \varepsilon_{n-1} \gg \varepsilon_n > 0$. But it is difficult to choose the value of $\boldsymbol{\varepsilon}$, and there will be arithmetic problem. A common way to choose the value for $\boldsymbol{\varepsilon}$ is that $\varepsilon_1 = c, \varepsilon_2 = c^2, \cdots, \varepsilon_n = c^n$, where c is a very small positive constant. But it still cannot ensure that $\varepsilon_1 > k\varepsilon_2$ after amount of additions, where k is a large positive constant. A better way to implement the perturbation method is the lexicographic method [37], in which $\boldsymbol{\varepsilon}$ are kept symbolic. Let us present the lexicographic perturbation with an example.

> **Example 4.3.** Considering again the example of Problem 4.2, the perturbated problem is shown in Problem 4.3.
>
> $$\begin{aligned} \text{maximize} \quad & x_3 \\ \text{subject to} \quad & -2x_1 + x_3 + x_4 = -2 + \varepsilon_1 \\ & 2x_1 - 4x_2 + 3x_3 + x_5 = -2 + \varepsilon_2 \\ & 2x_1 + x_3 + x_6 = 6 + \varepsilon_3 \\ & 2x_2 + x_3 + x_7 = 8 + \varepsilon_4 \\ \text{and} \quad & x_i \geq 0 \quad \forall i \in \{1,2,3,4,5,6,7\} \end{aligned} \tag{4.3}$$
>
> After perturbation, the Table 4.1 (in Page 62) becomes Table 4.5. Given the entering variable x_3, the only choice of leaving variable is x_7, as $2 + \frac{1}{2}\varepsilon_1 + \frac{1}{2}\varepsilon_3 > 2 + \frac{1}{6}\varepsilon_1 + \frac{1}{6}\varepsilon_2 + \frac{1}{3}\varepsilon_4$. In other words, the

dictionary in Table 4.2 will become infeasible as we can obtain $x_7 = -\varepsilon_1 + \frac{1}{2}\varepsilon_2 - \frac{3}{2}\varepsilon_3 < 0$. For the same reason, the basis $x_1,\ x_2,\ x_3,\ x_4$ will also lead to infeasibility.

Geometrically the vertex $\boldsymbol{v}$ is split into two vertices $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, as it is shown in Figure 4.3. $\boldsymbol{v}_1 = (2 - \frac{5}{12}\varepsilon_1 + \frac{1}{12}\varepsilon_2 + \frac{1}{6}\varepsilon_4,\ 3 - \frac{1}{12}\varepsilon_1 - \frac{1}{12}\varepsilon_2 + \frac{1}{3}\varepsilon_4,\ 2 + \frac{1}{6}\varepsilon_1 + \frac{1}{6}\varepsilon_2 + \frac{1}{3}\varepsilon_4)$ is the optimal vertex without primal degeneracy.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	= constant
row 1	1	0	$-\frac{1}{2}$	$-\frac{1}{2}$	0	0	0	$1 - \frac{1}{2}\varepsilon_1$
row 2	0	1	-1	$-\frac{1}{4}$	$-\frac{1}{4}$	0	0	$1 - \frac{1}{4}\varepsilon_1 - \frac{1}{4}\varepsilon_2$
row 3	0	0	2	1	0	1	0	$4 + \varepsilon_1 + \varepsilon_3$
row 4	0	0	3	$\frac{1}{2}$	$\frac{1}{2}$	0	1	$6 + \frac{1}{2}\varepsilon_1 + \frac{1}{2}\varepsilon_2 + \varepsilon_4$
objective	0	0	1	0	0	0	0	0

Table 4.5

An alternative way to implement the perturbation method is to compare the perturbation terms in lexicographic order, i.e., the lexicographic perturbation, which will be explained in details later in this chapter.

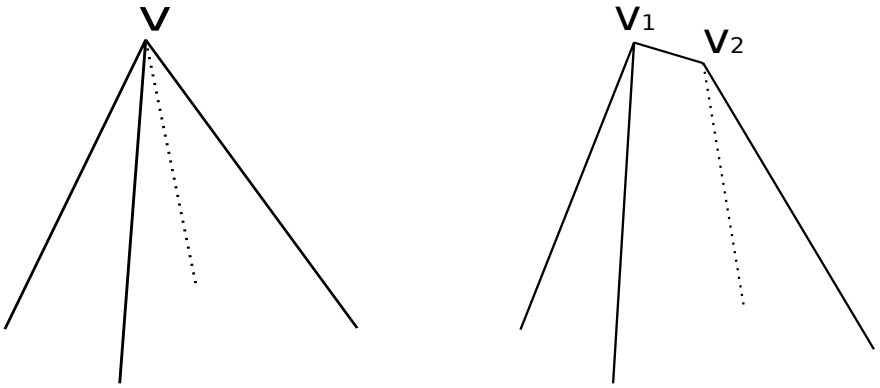


Figure 4.3 – Split vertex.

4.2.2 Dual degeneracy

It is proposed in [38] that: cycling can only occur in the presence of primal degeneracy. That means the dual degeneracy will not lead to cycling. When there are multiple optimal vertices, the Simplex algorithm will reach one of them. If we only care about the optimal value the dual degeneracy will not affect the result.

4.3 Degeneracy in Parametric Linear Programming

In this section we show the impact of the degeneracy on the PLP. We will first prove that there is no dual degeneracy our PLP problems for computing projection and convex hull thanks to the normalization constraint. Then we will propose a method to deal with the primal degeneracy for avoiding overlapping regions.

4.3.1 Overlapping regions

Ideally, the parametric linear programming outputs a partition of the space of parameters, meaning that the produced regions do not have overlap except on their boundaries (we shall from now on say "do not overlap" for short) and cover the whole parameter space. This may not be the case due to two reasons: i) geometric degeneracy, which leads to overlapping regions; ii) imprecision due to floating point arithmetic, which result in insufficient coverage. The latter will be dealt with by our rational checker, which will be explained in Section 3.5.

When the regions do not overlap, it is possible to check if the space of parameters is fully covered. It needs to verify that each boundary of a region is shared by one of its adjacent regions (proof in Section 3.5). If a boundary has no region on the other side, the space is not fully covered. This simple test cannot be used if there exist overlapping regions. We thus need to get rid of overlapping regions.

In a non-degenerate parametric linear program, for a given optimization objective, there is only one optimal vertex (no *dual degeneracy*), and this optimal vertex is described by only one basis (no *primal degeneracy*), i.e., the optimal solution corresponds to a unique partition of variables into basic and non-basic. Thus, in a non-degenerate parametric linear program, for a given instantiation of parameters there is one single optimal basis (except at boundaries), meaning that only one basis corresponds to one region. However when there is degeneracy, there will be multiple bases corresponding to one optimal vertex/function, and each of them computes a region. These regions may be overlapping. We call the regions corresponds to the same optimal function *degenerated regions*.

Theorem 4.2. There will not be overlapping regions if there is no degeneracy.

Proof. Recall that the solution of a PLP problem is a set of tuples $(\mathcal{R}_i, Z_i^*(\boldsymbol{x}))$, each of which associates to a basis obtained from GLPK. Each optimal function may correspond to multiple regions, i.e., $\exists i, j$ such that $\mathcal{R}_i \neq \mathcal{R}_j$ but $Z_i^*(\boldsymbol{x}) = Z_j^*(\boldsymbol{x})$.

In parametric linear programming, the regions are yielded by the partition of variables into basic and non-basic, i.e., each region is defined by a unique basis. The parameters within one region lead the PLP problem to the same partition of variables. If there are overlapping regions, say $\mathcal{R}_i$ and $\mathcal{R}_j$, the PLP problem will be optimized by multiple bases when the parameters belong to $\mathcal{R}_i \sqcap \mathcal{R}_j$. In this case there must be degeneracy: these multiple bases may lead to multiple optimal vertices when we have dual degeneracy (in fact we will prove that there is no dual degeneracy in our algorithm of PLP solver), or the same optimal vertex when we have primal degeneracy. By transposition, we know that if there is

	non-basic variables				basic variables					constants
	λ_1	$\cdots$	λ_q	$\cdots$	λ_r	$\cdots$	λ_s	$\cdots$	λ_n	
$\vdots$	.....									
row j	.....		a_{jq}	$\cdots$	1	$\cdots$	0	$\cdots$	0	b_j
$\vdots$	.....									
row k	.....		a_{kq}	$\cdots$	0	$\cdots$	1	$\cdots$	0	b_k
$\vdots$	.....									
objective	f_1	$\cdots$	f_q	$\cdots$	0	$\cdots$	0	$\cdots$	0	$-Z^*(\boldsymbol{x})$

Table 4.6 – Simplex tableau.

no degeneracy, the PLP problem will always obtain a unique basis with given parameters, and there will be no overlapping regions. □

4.3.2 Dual degeneracy in PLP

Theorem 4.3. For projection and convex hull, the parametric linear program exhibits no dual degeneracy.

Proof. We shall see that the *normalization constraint* (the constraint $(*)$ in Problem 3.9) in the parametric linear programs defining projection and convex hull prevents dual degeneracy.

Assume that at the optimum $Z^*(\boldsymbol{x})$ we have the Simplex tableau in Table 4.6. Let λ_k denote the decision variables with $\lambda_k \geq 0$, and let $f_k = \boldsymbol{o}_k\boldsymbol{x} + c_k$ be the parametric coefficients of the objective function. Assuming the basic variable leaving the basis is λ_r and the nonbasic variable entering the basis is λ_q. Currently λ_r is defined by the jth row as $\sum_j a_{jp}\lambda_p + \lambda_r = b_j$, where the λ_p are nonbasic variables. That means $\lambda_r = b_j$ when the nonbasic variables are set to their lower bound, which is 0 here.

We look for another optimum in another direction by doing one pivoting. As the current system is feasible, we have $b_j \geq 0$. To maintain the feasibility, we must choose λ_q such that $a_{jq} > 0$. As we only choose the non-basic variable who has negative objective coefficient to enter the basis, then we know $f_q < 0$. By pivoting we obtain the new objective function $Z'(\boldsymbol{\lambda},\boldsymbol{x}) = Z(\boldsymbol{\lambda},\boldsymbol{x}) - f_q\frac{b_j}{a_{jq}}$. The new basis defines a new solution λ^* and a new optimal function shown in Equation 4.4.

$$Z^{*'}(\boldsymbol{x}) = Z^*(\boldsymbol{x}) - f_q\frac{b_j}{a_{jq}} \tag{4.4}$$

Let us assume that we are faced with the dual degeneracy, which means that we obtain the same objective function after the pivoting, i.e., $Z^{*'}(\boldsymbol{x}) = tZ^*(\boldsymbol{x})$, where t is a positive scalar. Let us denote the normalization point by $\boldsymbol{p}$. Due to the normalization constraint at the point $\boldsymbol{p}$ enforcing $Z^{*'}(\boldsymbol{p}) =$

$Z^*(\boldsymbol{p}) = 1$, we have $t = 1$. Hence we obtain Equation 4.5.

$$Z^{*'}(\boldsymbol{x}) = Z^*(\boldsymbol{x}) \tag{4.5}$$

Considering Equation 4.4 and Equation 4.5 we obtain

$$f_q \frac{b_j}{a_{jq}} = 0 \tag{4.6}$$

Since $f_q \neq 0$, b_j must equal to 0, which means that we in fact faced a primal degeneracy.

□

4.3.3 Primal degeneracy in PLP

Many methods to prevent the non-parametric LP problem from cycling are known [35, 25, 36]. These methods can be applied to the PLP problems. But in PLP problems we also need to avoid overlapping regions besides avoiding the cycling. We implemented an approach to avoid overlapping regions based on the work of Jones et al. [39], which used the lexicographic perturbation method [25]. This method adds the perturbation terms to the right-hand side of the constraints. We need to modify their approach for two reasons: i) GLPK does not support the lexicographic perturbation method, and thus we cannot solve an LP problem with the perturbation matrix; ii) we need to adapt the parallel algorithm.

Our strategy is the following: once entering a new region, we check if there is primal degeneracy: it occurs when one or several basic variables equal zero. In this case we will explore all degenerated regions for the same optimal function, using, as explained below, a method avoiding overlaps. Our algorithm is shown in Algorithm 4.

How to avoid the primal degeneracy in PLP? First of all, let us see a case where the regions are overlapping.

> **Example 4.4.** Figure 4.4 shows two regions in 3D, and $\boldsymbol{p}$ is the normalization point. We cut the regions and obtain a 2D view of them, which is shown in blue and pink. Figure 4.5 shows the 2D view of the regions, and these regions correspond to the same optimal function. Figure 4.5(a) and Figure 4.5(b) are two possible compositions of degenerated regions that cover the whole corresponding space. Figure 4.5(c) shows the problematic case where the regions are overlapping. The reason is that when the parameters locate in the purple area, two different bases will lead the constructed LP problem to optimum. We aim to avoid the overlap and obtain the result either in Figure 4.5(a) or in Figure 4.5(b).

Our solution against overlaps is to avoid the primal degeneracy, as we have proved that if there is no degeneracy there will be no overlapping regions (Theorem 4.2). The method to avoid primal degeneracy is adding perturbation terms to the right-hand side of the constraints [39]. These perturbation terms are "infinitesimal", meaning that the right-hand side, instead of being a vector of rational scalars, becomes a

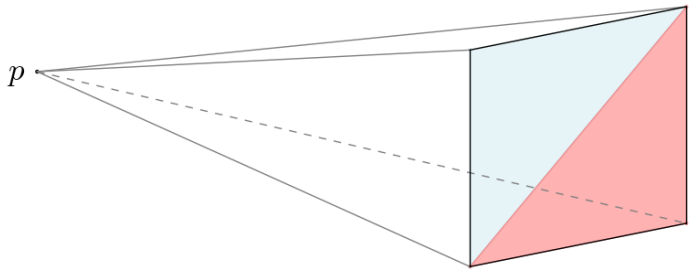
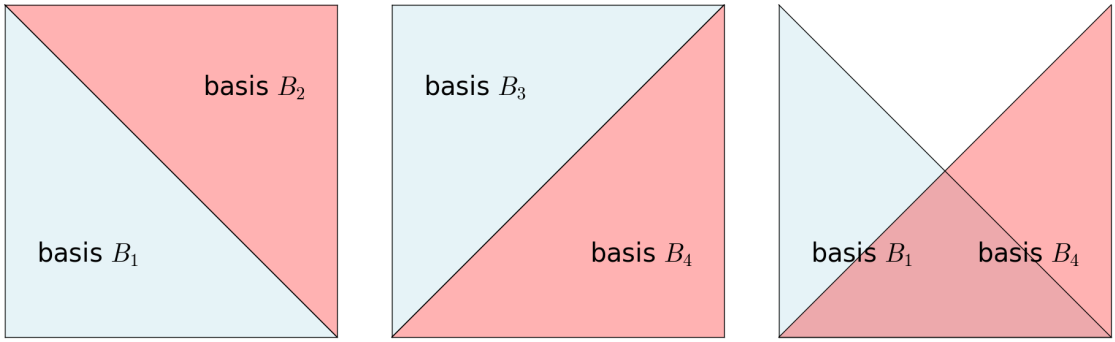


Figure 4.4 – A region in 3D.



(a) (b) (c)

Figure 4.5 – Example of overlapping regions. The regions obtained from bases B_1 and B_2 cover the whole space corresponding to their face without overlapping; similar for B_3 and B_4. But we may obtain the bases B_1 and B_4, which results in overlaps and an incomplete coverage of the space.

matrix where the first column corresponds to the original vector $\boldsymbol{b}$, the second column corresponds to the first infinitesimal ε_1, the third column to the second infinitesimal ε_2, etc. Instead of comparing scalar coordinates using the usual ordering on rational numbers, we compare row vectors of rationals using the lexicographic ordering. After the perturbation, there will be no primal degeneracy. As the perturbed vectors $b' + (k'_1\varepsilon_1, \cdots, k'_n\varepsilon_n)$ and $b'' + (k''_1\varepsilon_1, \cdots, k''_n\varepsilon_n)$ in two different rows cannot be equal.

Implementation of the perturbation method The initial perturbation matrix is a $k * k$ identity matrix: $M_p = I$, where k is the number of constraints. Then the perturbation matrix will be updated during the reconstruction of the constraint matrix. After adding this perturbation matrix, the right-hand side becomes $B = [\boldsymbol{b}|M_p]$. The new constants are vectors in the form $\boldsymbol{v}_i = [b_i\ 0\ \cdots\ 1\ \cdots\ 0]$. We compare the vectors by lexico-order: $\boldsymbol{v} >_l \boldsymbol{v}'$ if the first non-zero element of $\boldsymbol{v}$ is greater than that of $\boldsymbol{v}'$, where $>_l$ represents lexicographically greater. For instance, $\boldsymbol{v}_1 = [1\ 0\ 0]$ is greater than $\boldsymbol{v}_2 = [0\ 1\ 1]$.

To obtain a new basis, in contrast to working with non-degeneracy regions, we do not solve the problem using floating point solver, as GLPK does not support features for perturbation method. Instead, we pivot directly on the perturbated rational matrix.

	non-basic variables					basic variables			constants
	λ_1	$\cdots$	λ_q	$\cdots$	$\cdots$	λ_r	$\cdots$	λ_n	
$\vdots$	.....								
row j	.....		a_{jq}	$\cdots$	$\cdots$	1	$\cdots$	0	b_j
$\vdots$	.....								
objective	$\cdots$	$\cdots$	$\sum_{k=1}^{n} o_{qk}x_k + c_q$	$\cdots$	$\cdots$	0	$\cdots$	0	$-Z^*(\boldsymbol{x})$

Table 4.7

	λ_1	$\cdots$	λ_q	$\cdots$	$\cdots$	λ_r	$\cdots$	λ_n	
$\vdots$	.....								
row j	.....		1	$\cdots$	$\cdots$	$\frac{1}{a_{jq}}$	$\cdots$	0	$\frac{b_j}{a_{jq}}$
$\vdots$	.....								
objective	$\cdots$	$\cdots$	0	$\cdots$	$\cdots$	$-\frac{1}{a_{jq}}(\sum_{k=1}^{n} o_{qk}x_k + c_q)$	$\cdots$	0	$-Z^*(\boldsymbol{x})$

Table 4.8

In each pivoting, one non-basic variable will be chosen for entering the basis. All the non-basic variables will be considered for finding out all the adjacent regions. Given an entering variable λ_j, we select λ_l in C_i as the leaving variable from all the constraints in which $b_i = 0$ such that the ratio $\frac{\boldsymbol{v}_i}{a_{ij}}$ is the smallest using the comparison $>_l$. If such a leaving variable with $b_i = 0$ does not exist, a new optimal function will be obtained by crossing the corresponding boundary and it will not be treated by Algorithm 4, but will be computed with a task point by Algorithm 2. On the other hand if a leaving variable with $b_i = 0$ exists, we will obtain a degenerated region (the optimal function will not change). In this case we store the basis associated to that region and we maintain a set of bases that have been explored. After each pivot, we will check if the obtained basis exists in the set of bases, and thus the already explored regions will not be recomputed. We search all the degenerated regions by crossing all the boundaries of the already explored degenerated regions. Once all the boundaries have been crossed, all the degenerated regions corresponding to the same optimal function are found.

We benefit from Algorithm 4 in another aspect: there is no need to check adjacency of the degenerated regions. As we compute an degenerated region by doing only one pivoting across a boundary, the obtained region must be adjacent to the original one, as it is shown in Theorem 4.4.

Theorem 4.4. The region obtained by doing one pivoting in the Simplex tableau must be adjacent to the original region.

Proof. Considering the Table 4.7, we want to cross the boundary $\sum_{k=1}^{n} o_{qk}x_k + c_q \geq 0$ of the region associated to the variable λ_q, i.e., we choose λ_q as entering variable. Assuming the leaving variable is λ_r,

by doing one pivoting we obtain the tableau in Table 4.8. The obtained region is $-\frac{1}{a_{jq}}(\sum_{k=1}^{n} o_{qk}x_k + c_q) \geq 0$. As $\frac{1}{a_{jq}} \geq 0$ (otherwise the dictionary becomes infeasible), the region is equivalent to $-(\sum_{k=1}^{n} o_{qk}x_k + c_q) \geq 0$, which has the same boundary of the previous region $\sum_{k=1}^{n} o_{qk}x_k + c_q \geq 0$ but refer to the complementary half-space, and the two regions are adjacent. □

As a summary, we transform from Algorithm 2 to Algorithm 4 when there are degenerated regions, which is detected by checking the existence of $b_i = 0$. For preventing the regions from overlapping, we need to avoid the primal degeneracy by adding perturbation terms to the right-hand side of the constraints of the PLP problem. When solving the PLP problem we do pivoting in the rational Simplex tableau instead of solving the instantiated LP problems using GLPK, as GLPK does not support the perturbation method.

Algorithm 4: Algorithm to avoid overlapping regions.

Input: $w^{\mathbb{F}}$: the task point
$plp^{\mathbb{Q}\times\mathbb{F}}$: the PLP problem to be solved
Output: degeneracy regions correspond to the same optimal solution
Function DiscoverNewRegion($w^{\mathbb{F}}$, $plp^{\mathbb{Q}\times\mathbb{F}}$)

```
// get the basis which leads the LP problem to the optimum
glpkBasis = GlpkSolveLp(w^F, plp^F)
basicIdx = FeasibilityChecker(plp^Q, glpkBasis)
// test if there is primal degeneracy
degenerate = Reconstruct(plp^Q, basicIdx)
/* if there is primal degeneracy, compute all the degenerated regions corresponding
   to the same optimal function */
if degenerate then
    size = GetSize(basicIdx)
    // add a perturbation matrix, which is an identity matrix initially
    perturbM^Q = GetIdentityMatrix(size)
    basisList = none
    Insert(basisList, basicIdx)
    degBasic = none
    foreach basic variable v do
        /* if the basic variable v equals to 0, then it is degenerated, and insert
           it into degBasic */
        if v == 0 then
            Insert(degBasic, GetIdx(v))
    while basisList ≠ none do
        // get a basis from the list of bases
        currBasis = GetBasis(basisList)
        if currBasis has been found then
            continue
        // use the basis to compute a region, and store the solution
        nonBasicIdx = GetNonBasic(currBasis)
        (reconstructM^Q, perturbM^Q) = Reconstruct(plp^Q, basicIdx, perturbM^Q)
        (newOptimum^Q, newRegion^{Q×F}) = ExtractResult(reconstructM^Q, nonbasicIdx)
        irredundantIdx=Minimize(newRegion^F)
        minimizedR^Q = GetRational(newRegion^Q, irredundantIdx)
        Insert((optimums^Q,regions^{Q×F}), (newOptimum^Q,minimizedR^{Q×F}))
        // compute new basis by crossing each boundary of the minimized region
        foreach constraint i in minimizedR^Q do
            // use the nonbasic variable associated to i as the entering variable
            enteringV = GetIdx(i)
            // choose the leaving variable using lexico-order comparison
            leavingV = SearchLeaving(degBasic, perturbM^Q)
            /* if the leaving variable can be found, there exists a degenerated
               region associated to the basis newBasis beyond the boundary of i  */
            if leavingV ≠ none then
                newBasis = GetNewBasis(basicIdx, enteringV, leavingV)
                Insert(basisList, newBasis)
```

4.4 Adjacency checker

We have proved that if no degeneracy occurs, there will be no overlapping regions. Thus all the regions have their adjacencies. We shall now prove that no face of the projected polyhedron will be missed if each region can find all the adjacent regions across each of its boundaries.

We will show that a situation as the one shown in Figure 4.6 cannot happen: the four regions correspond to different optimal functions. $\mathcal{R}_1, \mathcal{R}_2$ and $\mathcal{R}_3$ all found their adjacent regions, but the region of the space denoted by $\mathcal{R}_4$ has not been explored. In Figure 4.6 there exist two adjacent regions for boundaries of $\mathcal{R}_4$. We here show that this is an impossible situation in a PLP problem.

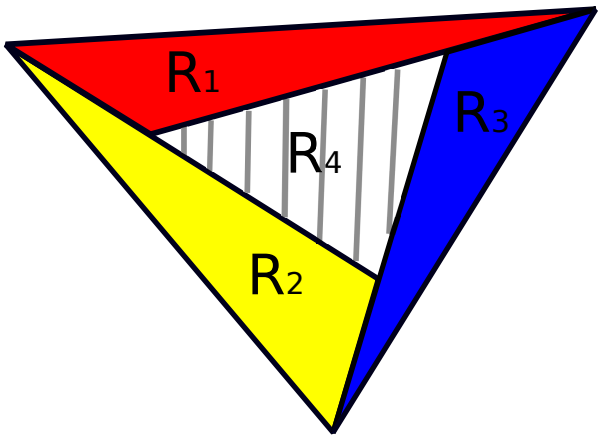


Figure 4.6 – An impossible situation: $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$ have their adjacent regions, but $\mathcal{R}_4$ is missing.

Theorem 4.5. No face is missed if all regions found their adjacent regions.

Proof. Consider the situation in Figure 4.7: we cross the boundary $\mathcal{F}$ of the region $\mathcal{R}_i$, and the adjacent regions related to $\mathcal{F}$ are $\mathcal{R}_j$ and $\mathcal{R}_k$ with corresponding optimal functions Z_j and Z_k, and $Z_j \neq Z_k$. We need to do one pivoting to cross $\mathcal{F}$ from $\mathcal{R}_i$. Assume λ_q is the entering variable associated to $\mathcal{F}$ in the objective. If there are two adjacent regions, there will be two possible leaving variables, say λ_r and λ_s. In the Simplex algorithm we always choose the variable with the smallest ratio of the constant and the coefficient as the leaving variable. When there are two possible leaving variables, the value of these two ratios must be equal, that is $\frac{b_j}{a_{jq}} = \frac{b_k}{a_{kq}}$. In this case we face the primal degeneracy, and $f^*(\boldsymbol{x}) - \frac{b_j}{a_{jq}} f_q = f^*(\boldsymbol{x}) - \frac{b_k}{a_{kq}} f_q$. This is a contradictory to the assumption $Z_j \neq Z_k$. Hence the situation can not happen. □

Theorem 4.5 shows that to find out all the faces, we just need to ensure that all the regions have their adjacencies. Although we tried to pick up task points between the regions which are not adjacent, in practice there may be still missed region because of floating point arithmetic, We invoke an adjacency checker in the last phase of the algorithm. If a boundary is not shared by two regions we will use our rational solver to explore beyond that boundary. We detail that last phase. Note that the information of

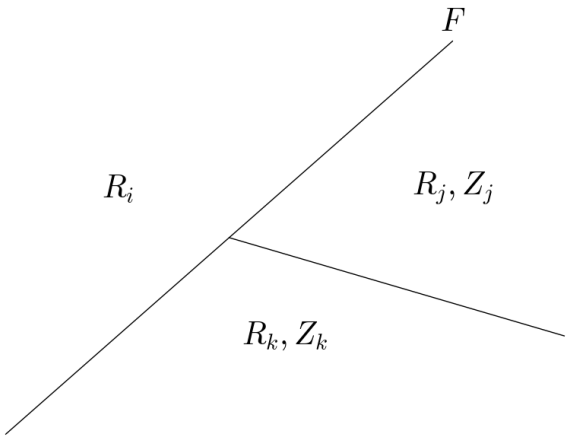


Figure 4.7 – The region $\mathcal{R}_i$ has two adjacent regions $\mathcal{R}_j$ and $\mathcal{R}_k$, whose optimal function Z_j and Z_k are not equal.

adjacency is saved in Algorithm 2: if the regions $\mathcal{R}_i$ and $\mathcal{R}_j$ are adjacent by crossing the boundaries $\mathcal{F}_m$ of $\mathcal{R}_i$ and $\mathcal{F}_n$ of $\mathcal{R}_j$, we set the cell of $(\mathcal{R}_i, \mathcal{F}_m)$ and $(\mathcal{R}_j, \mathcal{F}_n)$ to true in the adjacency table. When the adjacency checker finds out a pair $(\mathcal{R}_k, \mathcal{F}_p)$ whose cell in the adjacency table contains false, we cross the boundary $\mathcal{F}_p$ and use a rational procedure to compute the missing region and the corresponding optimal function. If the region r does not have its adjacency beyond the boundary b, we compute the missing region by crossing b. In other words, we choose the variable associated to b as the entering variable and do one pivoting for obtaining the missing region. Then we check if all the boundaries of the new obtained region have known adjacent regions. The main algorithm terminates when all the new obtained regions have complete adjacencies. Algorithm 5 summarizes this process.

Algorithm 5: Algorithm to check adjacency and add missed regions.

Input: $plp^{\mathbb{Q}\times\mathbb{F}}$: the PLP problem to be solved
adjInfo: the graph that contains the information of adjacency
$(optimums^{\mathbb{Q}}, regions^{\mathbb{Q}\times\mathbb{F}})$: the list of optimal functions with their regions

Function AdjacencyChecker($plp^{\mathbb{Q}\times\mathbb{F}}$, *adjInfo*, $(optimums^{\mathbb{Q}}, regions^{\mathbb{Q}\times\mathbb{F}})$)

```
toBeFound = none
// find out each region r whose adjacent region is missing beyond the boundary b
foreach r in 0 to Size(regions^{Q×F}) do
    foreach b in 0 to BoundarySize(r) do
        if HasAdjacency(adjInfo, r, b) == false then
            Insert(toBeFound, (r,b))
while toBeFound ≠ none do
    (r,b) = GetBoundary(toBeFound)
    adj = false
    // try to find adjacent regions among all the regions whose adjacency is missing
    foreach (r',b') in toBeFound do
        adj = AreAdjacency(regions^{Q}, r, b, r', b')
        if adj == true then
            SetAdjInfo(adjInfo, r, b, r', b')
            Remove(toBeFound,(r',b'))
            break
    /* if the adjacent region has not been found, compute the missing region using a
       rational procedure */
    if adj == false then
        (newOptimum^{Q}, newRegion^{Q}) = CrossBoundary(plp^{Q}, regions^{Q}, r, b)
        newRegion^{F} = GetFloat(newRegion^{Q})
        minimizedR^{Q×F} = Minimize(newRegion^{Q×F})
        foreach boundary in newRegion^{Q×F} do
            (newR,newB) = GetIdx(minimizedR^{Q×F}, boundary)
            Insert(toBeFound, (newR,newB))
```

Summary of the chapter

In this chapter we illustrated the primal and dual degeneracy with examples. For LP problems the primal degeneracy is more noticed than dual degeneracy as the former may lead to cycling. There are various methods of avoiding cycling, such as Blind's rule and the perturbation method.

For PLP the degeneracy leads to overlapping regions. We proved the absence of dual degeneracy in our PLP problems for computing projection and convex hull, and thus we only need to deal with the primal degeneracy. Based on the work in [39] we adapted the lexicographic perturbation to our PLP solver. After the perturbation we can always obtain a unique basis with any task point in each region, and thus overlapping regions are avoided.

We proved that, without overlapping regions, if all the regions have their adjacent regions, there is no missing region. If any adjacent region is missing, we use a rational procedure to compute this region. The adjacency checker guarantees the precision of our result.

Chapter 5

Parallelism

In this chapter we first present some preliminaries of our parallel algorithm. Then we show the parallel algorithm of the PLP solver with a second level parallelism for the raytracing minimization. In the parallel raytracing minimization, each thread computes the redundancy status of one constraint. The parallelism is simple to implement as the computations of the redundancy status of the constraints are independent. In the parallel PLP solver, each thread computes one optimal function with the corresponding region. It is more complex to parallelize this algorithm as the number of regions is unknown in advance, i.e., the tasks are dynamically added. Besides there is data shared by multiple threads. To implement the parallelism we need to modify our sequential algorithm shown in Algorithm 2 and Algorithm 4 at several steps. The work in this chapter was presented in our paper [40].

5.1 Preliminaries

We will explain *race conditions*, which may lead the parallel algorithm to an undesirable situation, and the techniques to avoid it. Then TBB and OpenMP will be introduced, using which we implemented the parallelism.

5.1.1 Race conditions and thread safe

A *race condition* occurs when the result of an algorithm depends on the sequence of the accessing of threads. In other words, if the sequence of multiple threads is not controlled, because of the race condition, the result may be incorrect.

Let us see a simple example. Assuming we have two pieces of code in Algorithm 6. The value of *val*2 could be 5, if the thread A increases *val* after the thread B reading it, or 6 otherwise.

Thread safe means that the data types or the piece of code can be executed by multiple threads without race conditions.

Algorithm 6: Example of race condition

```
array = {1,3,5,7,9}
Function Thread_A
    foreach val in array do
        val = val + 1
Function Thread_B
    val2 = array[2] + 1
```

5.1.2 Mutex and lock-free algorithm

Mutex (Mutual exclusion) is a strategy to prevent race conditions. If a piece of code is blocked by a mutex, only one (or a given number) thread can access it. By locking the shared data, the race conditions can be avoided.

Lock-free data structures can be accessed by multiple threads safely without using mutex locks. Lock-free strategy could provide better performance than using mutex.

5.1.3 TBB and OpenMP

We implemented the parallel algorithm using Intel's Thread Building Blocks (TBB)[1], and OpenMP tasking[2] [4] respectively, which can be selected during compilation time.

Intel Threading Building Blocks (TBB) is a C++ template library for task-based parallel programming. It provides concurrent data structures and task scheduling algorithms. To use TBB, we just need to specify the tasks without any details about threads, and the TBB library will map the tasks onto threads efficiently. Besides, TBB also supports nested parallelism.

OpenMP (Open Multi-Processing) is an application program interface (API) that is used for shared memory multiprocessing programming in C/C++ and Fortran on multi-platform. It includes both compiler directives and runtime library routines. Parameters such as the number of threads to be used can be set both programmatically and through environment variables.

5.2 Parallel raytracing minimization

Now we explain the parallelism on raytracing minimization. As the first phase of the algorithm only contains a matrix multiplication and selecting the minimum value which is the minimum distance from the interior point to the constraint, the execution time can be negligible with respect to the second phase. Thus we only parallelized the second phase. The parallelism is easy to be implemented as the rays are launched independently.

The only shared data is the list containing the redundancy status of the constraints, which is stored in `std::vector<Status>`, where `Status` is a user-defined structure. Any constraint has one of the three

[1] `https://www.threadingbuildingblocks.org/`
[2] The OpenMP version is implemented by Camile Coti and David Monniaux.

status: irredundant, redundant, or undetermined. Initially all constraints are undetermined. In the first phase, some constraints are determined as irredundant. In the second phase, each thread will take one undetermined constraint and check its redundancy status. The list of the state is preassigned and has fixed size, which is the same as the number of constraints. So there is no concurrency problem of access and reallocation, i.e., the container will not grow and the stored elements will not be moved. Another problem we need to take care is the race condition. In this algorithm we do not need any extra work to protect the shared data because of two reasons: i) each thread will only modify a specific cell in the vector; ii) one thread will not read other cells in the list.

Parallelism using OpenMP The most straightforward way to parallelize the algorithm is to use the OpenMP directive `omp parallel for`, which is the combination of `omp parallel` and `omp for`. The directive `omp parallel` invokes additional threads, and the original and new created threads will execute the code inside the scope marked by this directive. The directive `omp for` splits the work and assign them among the threads.

We can also select the strategy of scheduling among `static`, `dynamic` and `guided`. The `static` strategy assigns the iterations to the threads evenly; the `dynamic` assigns a given number to each thread, and the thread which finishes its work firstly will take the next group of tasks; `guided` is similar to `dynamic`, but the number of tasks in each group to be allocated will decrease exponentially.

We put the second phase of the raytracing minimization inside the `omp parallel for` scope using `dynamic` strategy, and the undetermined constraints will be assigned to the active threads.

Parallelism Using TBB TBB provides `tbb::parallel_for` function, which implements similar strategy to `omp parallel for`. It accepts a range of tasks, and the range will be partitioned into subranges. As it is shown in Algorithm 7, `tbb::parallel_for` calls the function `operator()` of a given class automatically with a divided subranges.

5.3 Parallel Parametric Linear Programming Solver

We have seen in Algorithm 2 that the algorithm of the PLP solver performs independent tasks, and thus the algorithm can be parallelized. Here we will present the parallel algorithm of the PLP solver.

5.3.1 Scheduling tasks

We use `tbb::parallel_do` to assign tasks to the threads, and use the provided feeder to add new tasks. Once one thread finishes its task, another task will be assigned.

In OpenMP version, we used the OpenMP *tasking* model to manage the tasks, in which a single thread creates tasks, and each task will be fetched by one thread. The tasking feature is introduced since OpenMP 3.0. Different from the `omp parallel` directive, in which the number of iterations is known in advance, OpenMP tasks can be dynamically created, and thus it can be used in while loops and recursive functions.

Algorithm 7: Parallel minimization.

Input: $undeterminedCons^{\mathbb{F}}$: the constraints to be checked for redundancy
Function Determine($undeterminedCons^{\mathbb{F}}$)
    size = GetSize($undeterminedCons^{\mathbb{F}}$)
    range = [0,*size*)
    #ifdef _TBB
    `// this function will call Body.operator() automatically`
    tbb::parallel_for(*range*,Body())
    #elif defined _OPENMP
    #pragma omp parallel for schedule(dynamic)
    for *idx* in *range* **do**
        DetermineStep(*idx*)
    #else
    `// without TBB or OpenMP, run as a sequential algorithm`
    for *idx* in *range* **do**
        DetermineStep(*idx*)
    #endif

`// the parameter subrange is provided by TBB`
Function Body.operator(*subrange*)
    for *idx* in *subrange* **do**
        DetermineStep(*idx*)

`// test the redundancy status by searching an irredundancy witness point`
Function DetermineStep(*idx*)
    if cannot determine **then**
        SetAsRedundant(*idx*)
    else
        if found irredundancy witness point **then**
            SetAsIrredundant(*idx*)
        else
            SetAsRedundant(*idx*)

5.3.1.1 Dealing with multiple threads

As we explained, we do not need to take care of the assignment of tasks, as it will be scheduled by TBB or OpenMP. The only matter we need to pay attention to is avoiding multiple threads computing on the same problem, which will result in the inefficiency of the algorithm. Consider that when the thread A is computing on the region $\mathcal{R}_1$, the thread B completed the computation on $\mathcal{R}_2$ and generated a witness point $\boldsymbol{w}$ which is inside $\mathcal{R}_1$. Then the thread C can fetch $\boldsymbol{w}$ and compute on the same region as the thread A, as $\mathcal{R}_1$ has not been generated.

It is impossible to completely avoid multiple threads computing on the same region, but we can reduce the possibility of its occurrence by Algorithm 8. If two threads compute on the same region, they

must obtain the same basis. Thus it is possible to detect the repetition of computations by comparing the basis before obtaining the region. Once obtain an optimal basis from GLPK, we store it into a hash table. If the basis has already been encountered, the thread should stop and abandon the current task. As the cost of solving an LP problem is much less than the whole process of computing a region, we can nearly avoid the repeated computation.

Algorithm 8: Checking bases

```
(basicIndices, nonbasicIndices) = GlpkSolveLp(w^F, plp^F)
if basicIndices is infeasible then
    (basicIndices, nonbasicIndices) = RationalSimplex(w^F, plp^Q)
// test if the basis has been obtained for avoiding recomputation
atomic existed = AddBasis(basicIndices)
if existed then
    return
else
    ...
```

5.3.2 Dealing with shared data

The optimal functions are stored in a shared list. The corresponding regions are also stored for checking weather a task point belongs to any known region and testing adjacency of two regions. The elements in the list may be accessed by multiple threads while other threads are appending new regions into the list.

In TBB version, the set of regions are stored in `tbb::concurrent_vector`, which is a concurrent data structure provided by TBB. Multiple threads can access the stored elements, grow the container and append new elements concurrently. In OpenMP version, we use our simple lock-free implementation[3] based on an array with a large, statically defined maximal capacity, using atomic operations for increasing the current number of items.

There is a problem in both versions. The size of the container will be increased before the new item is appended, and thus not all the items in the list are ready for reading. A segmentation fault will occur if we try to fetch the new item when it is not completely appended yet. For instance, if we append a new element to the container containing n elements, whose indices are 0 to $n-1$, at a moment the size of the container is $n+1$ while it only has n elements. Thus if we try to get the new element by its index n, the segmentation fault will occur.

To solve this problem, we use the mutex lock in C++ standard library or a lock-free implementation, in which we store the number of items which are ready for reading. If the number of ready items n_{ready} is less than the size of the container n_{fill}, then the threads need to wait until all the items are ready for reading. The algorithm of concurrent appending is shown in Algorithm 9.

[3] The lock-free data structure is provided by David Monniaux.

Algorithm 9: Current appending

Input: $(newOptimum^{\mathbb{Q}}, newRegion^{\mathbb{Q}\times\mathbb{F}})$: the new optimal function with its region that will be appended
$(optimums^{\mathbb{Q}}, regions^{\mathbb{Q}\times\mathbb{F}})$: the list of optimal functions with their regions

Function CurrentAppend
- **atomic** { $i = n_{fill}$, $n_{fill} = n_{fill} + 1$ }
- Insert($(optimums^{\mathbb{Q}}, regions^{\mathbb{Q}\times\mathbb{F}})$, $(newOptimum^{\mathbb{Q}}, newRegion^{\mathbb{Q}\times\mathbb{F}})$)
- **while** $n_{ready} < i$ **do**
  - spinning or using a condition variable
- // $n_{ready} == i$ here
- **atomic** { $n_{ready} = i+1$ }

5.3.3 Updating algorithm of preventing degeneracy

In sequential version of Algorithm 4, we simply start from any basis and find out all the degenerated regions which correspond to the same optimal function. But this is not enough for avoid overlapping regions in parallel algorithm. Consider the situation in Figure 5.1: thread 1 took the task point $\boldsymbol{w}_1$, and it is computing on the region $\mathcal{R}_1$; meanwhile thread 2 obtained the task point $\boldsymbol{w}_2$, which is not covered by any known region as the computation of $\mathcal{R}_1$ has not completed. Thus thread 2 obtains the basis corresponding to the region $\mathcal{R}_2$ by solving the LP problem. As the obtained basis has not been found, thread 2 will continue the computation and obtain $\mathcal{R}_2$, which overlaps with $\mathcal{R}_1$. Thus we need to update Algorithm 4 for avoiding this situation, and the updated algorithm is shown in Algorithm 11.

Please note that in the sequential version this problem does not appear because the single thread needs to complete the computation on $\mathcal{R}_1$ before fetching the task point $\boldsymbol{w}_2$. Then it will report that $\boldsymbol{w}_2$ is covered by $\mathcal{R}_1$.

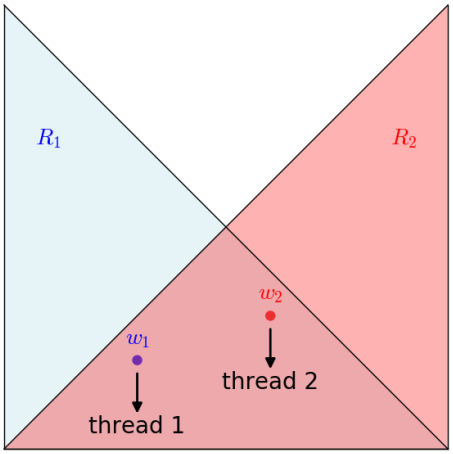


Figure 5.1 – Multiple threads obtained different basis in parallel algorithm.

Our solution to against this problem is to find out the "smallest" basis among all the bases associated to the degenerated regions, meaning that the basic variables have smallest subscripts. Once this basis is found, we append it into a hash table containing the bases that have been explored. Thus when another thread obtains a task point inside a degenerated region corresponding to the same optimal function, it will also find out the same smallest basis, which is always the same one. Then it will figure out that the basis has been seen and stop the computation.

Algorithm 10 shows the algorithm to find out the ordered basis list, from which we can obtain the "smallest" basis. We firstly count the number of degenerated basic variables, i.e., the number of zeros appears on the right-hand side of the reconstructed constraints and denote this number by *degNum*. Then from the degenerated basic variables and the nonbasic variables select *degNum* variables that will construct the basis with the remaining basic variables.

Example 5.1. Given the basic variables λ_2, λ_4 of which $\lambda_2 = 0$, and the nonbasic variables λ_1, λ_3, we need to choose one variable among λ_1, λ_2 and λ_3 to construct the basis with λ_4. The first choice is λ_1 as it has the smallest subscript. We can obtain an ordered list of candidates of bases: $\{(\lambda_1, \lambda_4),\ (\lambda_2, \lambda_4),\ (\lambda_3, \lambda_4)\}$. We firstly take (λ_1, λ_4), and test if the corresponding region has non-empty interior. If the region is flat, we then take (λ_2, λ_4) and so on.

5.3.4 Advantages and drawbacks of two versions

5.3.4.1 TBB

In TBB version we benefit from the provided interfaces and concurrent containers. One drawback of TBB is that in general cases it provides stack-like constructs, which means that the tasks are last-in-first-out[4]. The algorithm of projection with PPLP is essentially a graph search algorithm. In our case it is better to do a breadth-first search to avoid repeatedly extracting the same regions (although the recomputation has been stopped by testing the existence of the basis obtained from GLPK), whereas the last-in-first-out containers lead to depth-first search.

OpenMP is easy to use. We do not need to install any library, and only use the provided compiler directives. But we need to guarantee that the compiler we used supports OpenMP.

[4] If a thread has enough tasks to do, it will do depth-first execution; otherwise it will do temporary breadth-first execution. More details are presented in https://software.intel.com/en-us/node/506103.

Algorithm 10: Obtain the basic variables who has smallest subscript.

Input: *basicIdx*: the indices of basic variables
nonbasicIdx: the indices of nonbasic variables
Output: a list contains the bases, which are sorted in lexico order of their subscript.
Function GetBasisList(*basicIdx, nonbasicIdx*)

```
idxList = none
remained = none
Insert(idxList,nonbasicIdx)
degNum = 0
// find all the basic variables which equal to 0
foreach basic variable v do
    if v == 0 then
        Insert(idxList, GetIdx()v))
        degNum = degNum + 1
    else
        Insert(remained, GetIdx(v))
/* now idxList contains the indices of nonbasic variables and that of the basic
   variables which equal to 0 */
Sort(idxList)
/* get all possible combinations of degNum variables from idxList, and the
   combinations are sorted by their indices in lexico-order */
selectedList = ChooseFrom(idxList, degNum)
/* construct a set of possible bases, which are sorted by their indices in
   lexico-order */
newBasisList = GetBasis(remained, selectedList)
return newBasisList
```

Algorithm 11: Updated algorithm to avoid overlapping regions.

Input: $w^{\mathbb{F}}$: the task point
$plp^{\mathbb{Q}\times\mathbb{F}}$: the PLP problem to be solved
Output: degeneracy regions correspond to the same optimal solution
Function DiscoverNewRegion($w^{\mathbb{F}}$, $plp^{\mathbb{Q}\times\mathbb{F}}$)

```
glpkBasis = GlpkSolveLp(w^F, plp^F)
basicIdx = FeasibilityChecker(plp^Q, glpkBasis)
degenerate = Reconstruct(plp^Q, basicIdx)
if degenerate then
    nonBasicIdx = none
    /* compute the set of possible bases, which are sorted by their indices in
       lexico-order */
    orderedBasisList = GetBasisList(basicIdx,nonBasicIdx)
    // find the smallest basis in lexico-order
    for b in orderedBasisList do
        nonBasicIdx = GetNonBasic(basicIdx)
        (reconstructM^Q) = Reconstruct(plp^Q, b)
        (newOptimum^Q, newRegion^(Q×F)) = ExtractResult(reconstructM^Q, nonbasicIdx)
        if newRegion^Q has non-empty interior then
            basicIdx = b
            break
    // start from the smallest basis, compute all the degenerated regions
    size = GetSize(basicIdx)
    perturbM^Q = GetIdentityMatrix(size)
    basisList = none
    Insert(basisList, basicIdx)
    degBasic = none
    foreach basic variable v do
        if v == 0 then
            Insert(degBasic, GetIdx(v))
    // the remaining part is the same as the sequential algorithm
    while basisList ≠ none do
        ...
```

Summary of the chapter

We parallelized our algorithms of raytracing minimization and PLP solver using TBB and OpenMP separately. Thanks to TBB and OpenMP the tasks are scheduled automatically, and we do not need to assign them to the threads by hand. One advantage of TBB is that it provides concurrent data structures, and thus the shared data can be accessed and appended concurrently. In OpenMP version we provided a simple implementation of lock-free data structure. In both version we need to pay attention to the shared data stored in the concurrent vector (for TBB version) or the lock-free data structure (for OpenMP version) when we access the items by their indices, since appending items and increasing the size of the container is not atomic.

The parallel PLP solver is different from the sequential algorithm in some details. To guarantee a unique basis for each region, we found out the "smallest" possible basis for avoiding two threads obtaining different bases with the task points inside the same region, and thus the overlapping regions were avoided.

Chapter 6

Experiments

In this section we will present some experiments on raytracing minimization and the PLP solver for computing the projection and convex hull. We first analyze the advantages and disadvantages of our sequential algorithms, and then we analyze the speedup (the ratio of the runtime of the sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem) of our parallelism by running it on various of threads.

We used three libraries in our implementation:

- Eigen 3.3.2 for floating-point vector and matrix operations;
- FLINT 2.5.2 for rational arithmetic, vector and matrix operations;
- GLPK 4.6.4 for solving linear programs in floating-point.

The experiments of minimization and some experiments on the projection and convex hull are performed using our randomly generated benchmarks, as we need a single varying parameter for analyzing the pros and cons. These benchmarks contain randomly generated polyhedra in which the coefficients of constraints range from -50 to 50. In each experiment, we use 10 polyhedra generated with the same parameters. To smooth out experimental noise, we perform the experiments on each polyhedron for 5 times resulting in 50 executions for each set of parameters. Then we sum up the running time of 50 executions, and divide the total time by 5 for obtaining the average execution time of 10 polyhedra. The second part of experiments on projection (and convex hull) used SV-COMP benchmarks [5], by which we will see the performance of our algorithm on real-world programs. We compare the performance on these benchmarks with ELINA library [13].

The experiments with sequential algorithms are carried out on a machine with Intel(R) Core(TM) i5-6200U CPU. And the parallel experiments are performed on two servers: S_1 has 2 Intel(R) Xeon(R) Gold 6138 CPU (each has 20 cores) and 192 GB of RAM; S_2 has 2 Intel(R) Xeon(R) CPU E5-2650, each CPU has 8 cores and 64 GB RAM. We used GCC 7.2.0 to compile the code.

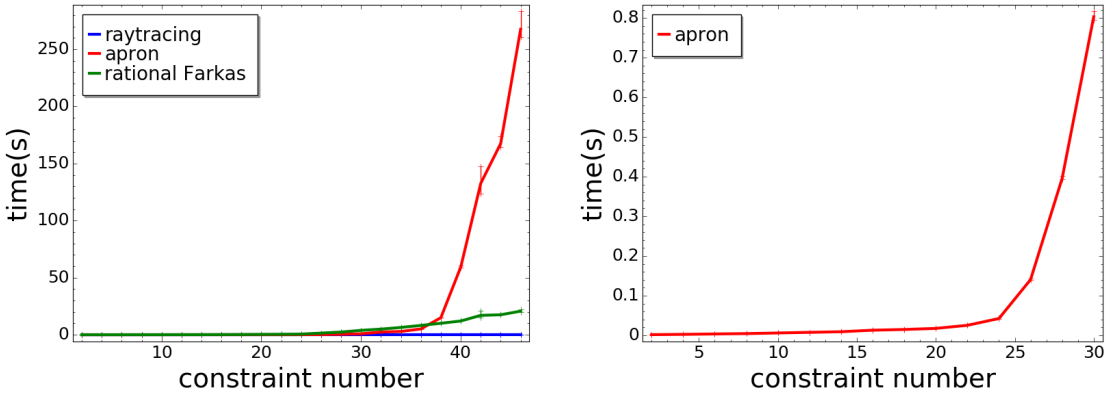
The figures of performance are shown in line charts, and the variance of 5 executions are shown by vertical bars.

6.1 Sequential algorithm for minimization

We first compare the performance of three minimization methods: the minimization in NewPolka library of Apron[1], rational minimization algorithm using Farkas' lemma, and the raytracing method. We consider three parameters: the number of constraints (C), the number of variable (V) and the redundancy ratio (R). For example, if we have 10 constraints where 3 are redundant, then $R = 30\%$. The redundant constraints are created as combinations of irredundant constraints. All the irredundant constraints have 20% density, i.e., there are 20% coefficients of the constraints are 0.

Then we compare the raytracing minimization and the floating point minimization algorithm using Farkas' lemma for analyzing the advantage of raytracing minimization on the polyhedra having low redundancy ratio.

Number of constraints In this experiment we used the polyhedra whose number of constraints ranges from 2 to 46. We can see in Figure 6.1 that when the number of constraints is small the performance of the three methods are similar, but the execution time of Apron increases significantly after that. This can be explained. As Apron uses double description of polyhedra, the cost depends on the number of generators. We have said that the number of generators increases exponentially when the polyhedra are hypercubes, meaning that a polyhedron in n dimension having 2^n generators. When $C' = V + 1$ the polyhedra are simplexes, where C' denotes the number of irredundant constraints. A simplex is a n-dimensional polyhedron whose has $n+1$ generators, i.e. the number of generators increases linearly. When $C' > V + 1$ the number of generators increases at a high growth rate. Figure 6.1(b) is a zoomed figure of Figure 6.1(a), and the variation of the number of generators is shown in Table 6.1.



(a) (b)

Figure 6.1 – Variation of constraints. C=[2,46], V=12, R=50%.

Number of variables In Figure 6.2 the line of Apron shows a peak, and when $C' \geq 2V$ the execution time of Apron increases sharply. Then the execution time decreases. This because the polyhedron tends

[1] `https://github.com/antoinemine/apron`

constraint number	2	4	6	8	10	12	14	16
generator number	130	130	130	130	130	130	130	130
constraint number	18	20	22	24	26	28	30	32
generator number	130	130	130	130	474	1252	2689	4836
constraint number	34	36	38	40	42	44	46	
generator number	8999	15725	22472	35947	50000	72448	97582	

Table 6.1 – Number of generators. C=[2,46], V=12, R=50%.

to be a simplex when C' and V tend to be equal, and the number of generators (see Table 6.1) will decrease if C' is large enough corresponding to V.

The execution time of rational minimization using Farkas' lemma increases steadily, and it performs better than Apron when the generator number is large (the middle in Figure 6.2). The line of raytracing minimization is nearly flat, which shows the advantage of floating-point arithmetic.

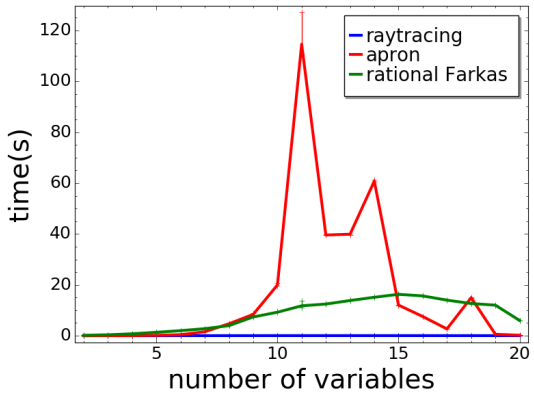


Figure 6.2 – Variation of variables. C=40, V=[2,20], R=50%

variable number	2	3	4	5	6	7	8	9	10	11
generator number	200	360	802	1780	3604	6568	11022	17800	23213	31606
variable number	12	13	14	15	16	17	18	19	20	
generator number	32320	32200	30734	21230	15067	8550	3613	1042	210	

Table 6.2 – Number of generators. C=40, V=[2,20], R=50%

Number of redundant constraints This experiment shows the effect of redundancy ratio, which is from 10% to 90%. In Figure 6.3 the cost of Apron is high when $R < 50\%$, i.e., the number of irredundant

constraints $C' < 20$, which is more than twice that of variables. In this case the polyhedra are hypercubes, and the cost of double description explodes. The number of generators is shown in Table 6.3. This shows the disadvantage of double description. The other methods seem to be less sensitive to the redundancy ratio compared to Apron.

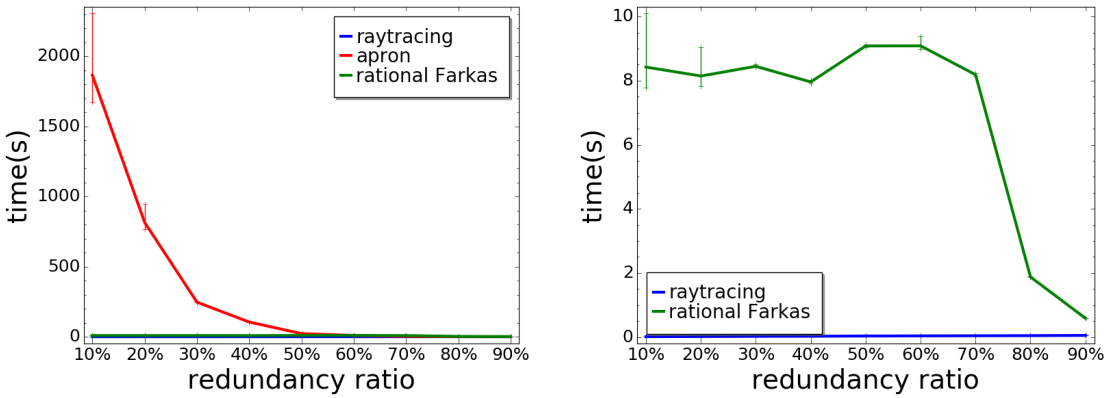


Figure 6.3 – Variation of redundancy. C=40, V=10, R=[10%,90%].

redundancy ratio	10%	20%	30%	40%	50%	60%	70%	80%	90%
generator number	383823	240358	135260	61426	24435	6334	827	110	110

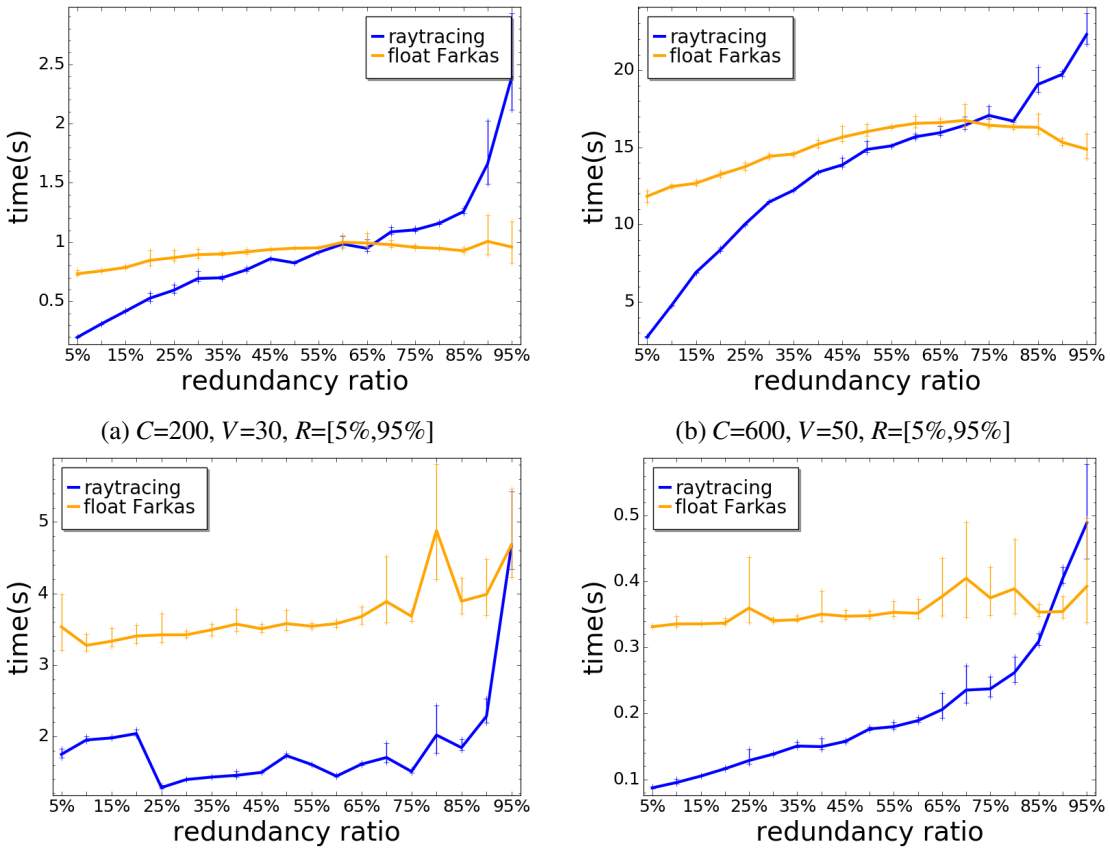
Table 6.3 – Number of generators. C=40, V=10, R=[10%,90%]%

In higher dimension We have seen that the floating-point raytracing method has advantages with respect to the rational methods. Here (Figure 6.4) we focuses on the two floating-point methods: raytracing minimization and floating point minimization using Farkas' lemma. We did experiments on polyhedra which have more constraints in higher dimensions. All the experiments show the advantage of the raytracing method when the polyhedra contain more irredundant constraints than redundant ones. In higher dimension the raytracing method has more obvious advantages.

Results analysis In general the raytracing method is more efficient than the rational ones. The constraint-only description is more efficient when the polyhedra are hypercubes. The raytracing method is more efficient than the floating point algorithm using Farkas' lemma when the ratio of redundancy is low. Therefore, it is better to use raytracing minimization for the polyhedra contain more irredundant constraints than redundant ones.

6.2 Parallel minimization

These experiments are performed on the server S_2 whose characteristics are described on Page 87. Figure 6.5 to Figure 6.8 shows the performance of the raytracing minimization with different number



(a) C=200, V=30, R=[5%,95%]

(b) C=600, V=50, R=[5%,95%]

(c) C=200, V=150, R=[5%,95%]

(d) C=60, V=200, R=[5%,95%]

Figure 6.4 – Performance of floating-point algorithms for minimization in higher dimension

of threads. The figures on the left shows the execution time of using 1 to 32 threads, and the zoomed figures are shown on the right.

As it is shown in Figure 6.5 and Figure 6.6, when the size of the polyhedra is not large, i.e. $C = 200$ and $V = 30$ in our experiment, using more threads than 8 does not improve the performance, especially when the redundancy ratio is low, as there are not enough parallel tasks. Besides the libraries need to pay extra effort for arranging the needless threads. In Figure 6.7 and Figure 6.8, the size of polyhedra are increased to $C = 600$ and $V = 50$. Using a larger number of threads shows better performance. Figure 6.9 compares the performance of OpenMP and TBB version on a polyhedron generated with $C = 600$, $V = 50$ and $R = 95\%$. We can see that the parallelisms using OpenMP and TBB have similar performance.

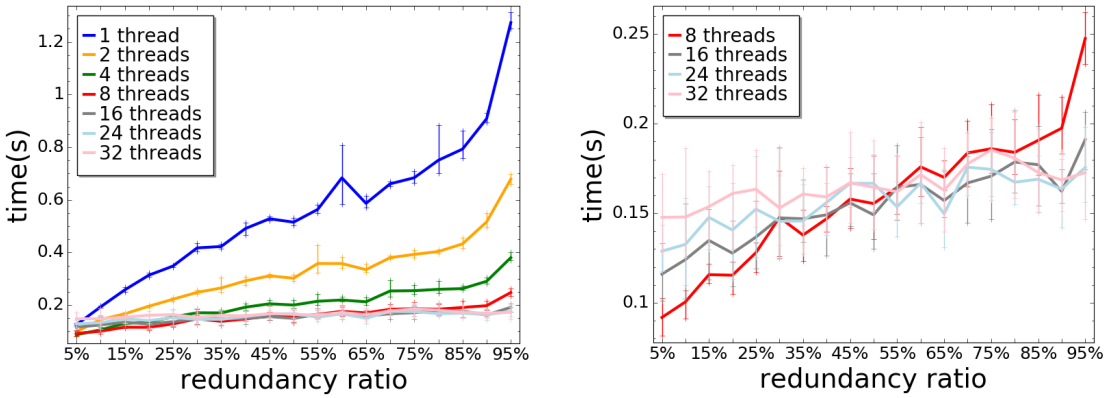


Figure 6.5 – Parallel raytracing minimization with OpenMP, C=200, V=30, R=[5%,95%]

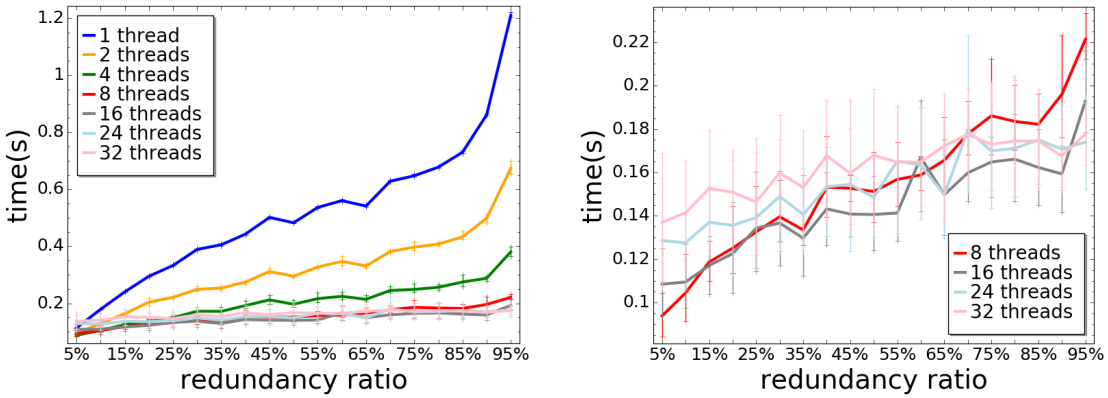


Figure 6.6 – Parallel raytracing minimization with TBB, C=200, V=30, R=[5%,95%]

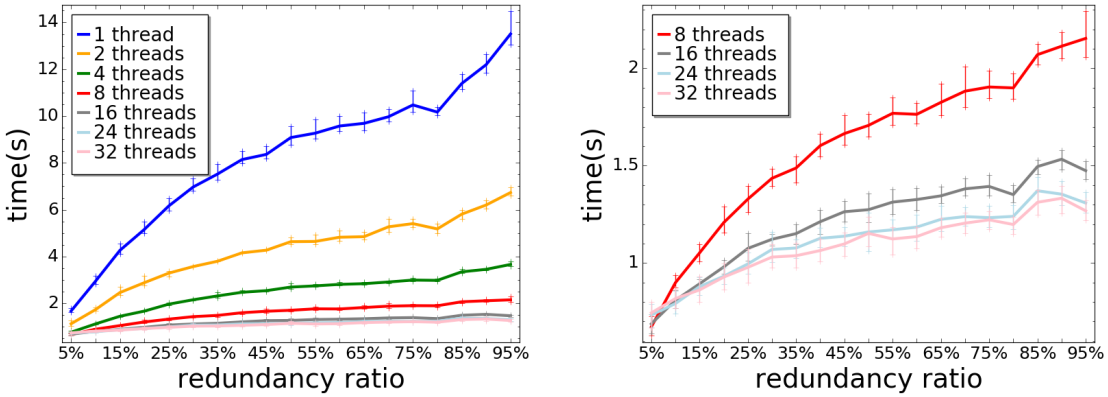


Figure 6.7 – Parallel raytracing minimization with OpenMP, C=600, V=50, R=[5%,95%]

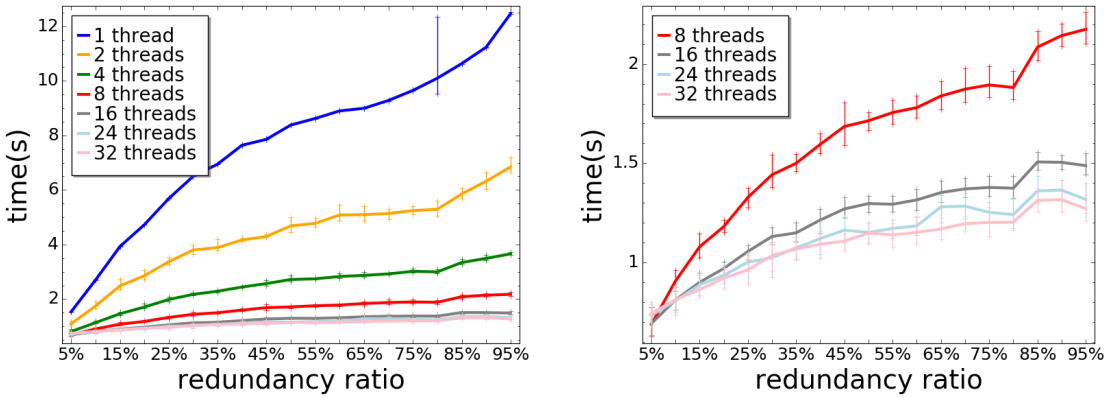
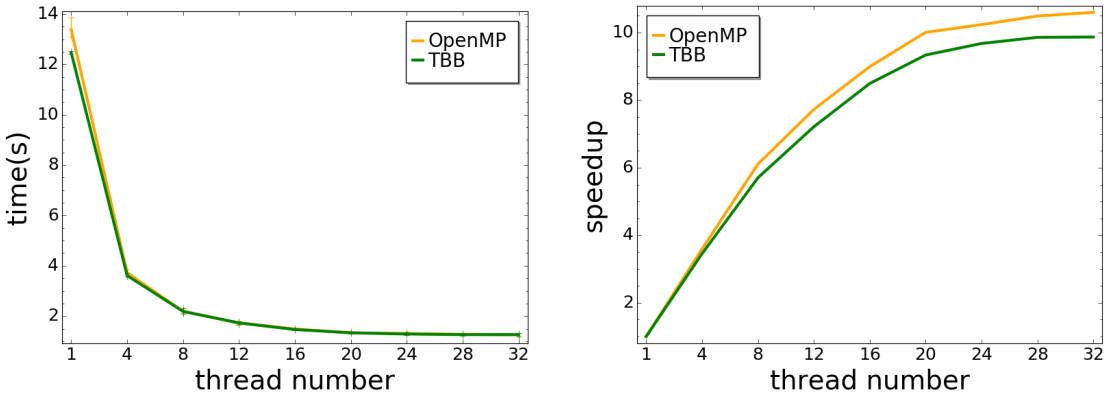


Figure 6.8 – Parallel raytracing minimization with TBB, C=600, V=50, R=[5%,95%]



(a) Performance.

(b) Speedup.

Figure 6.9 – Parallel raytracing minimization with different threads, C=600, V=50, R=95%

6.3 Projection via sequential PLP

In this section, we analyze the performance of our PLP solver on projection operators. We compare its performance with that of the NewPolka library of Apron.Since NewPolka does not exploit parallelism, we compare it to our library running with only one thread, and thus we consider our PLP solver as a sequential algorithm.

6.3.1 Experiments on randomly generated benchmarks

In the line charts, the execution time of NewPolka library of Apron is depicted in blue, and our solver (denoted by PLP) appears in red. To illustrate the performance benefits from the floating-point arithmetic, we turned off GLPK and always use the Simplex algorithm implemented in rational numbers (denoted by rational), whose execution time is shown by the orange lines[2]. It reveals that solving the LP problems in floating-point numbers and reconstructing the rational Simplex tableau leads to significant improvement of performance.

The generation of polyhedra is controlled by 4 parameters: number of constraints (C), number of variables (V), projection ratio (PR) and density (D). The projection ratio is the proportion of eliminated variables: for example if we eliminate 6 variables out of 10, the projection ratio is 60%. The density represents the ratio of zero coefficients: if there are 2 zeros in 10 coefficients, the density is 20%. Through numerous experiments, we found that when the parameters $C = 19, V = 8, PR = 62.5\%$ and $D = 37.5\%$, the execution time of PLP and Apron are similar, so we maintain three of them and make the other one varying to analyze the variation of performance.

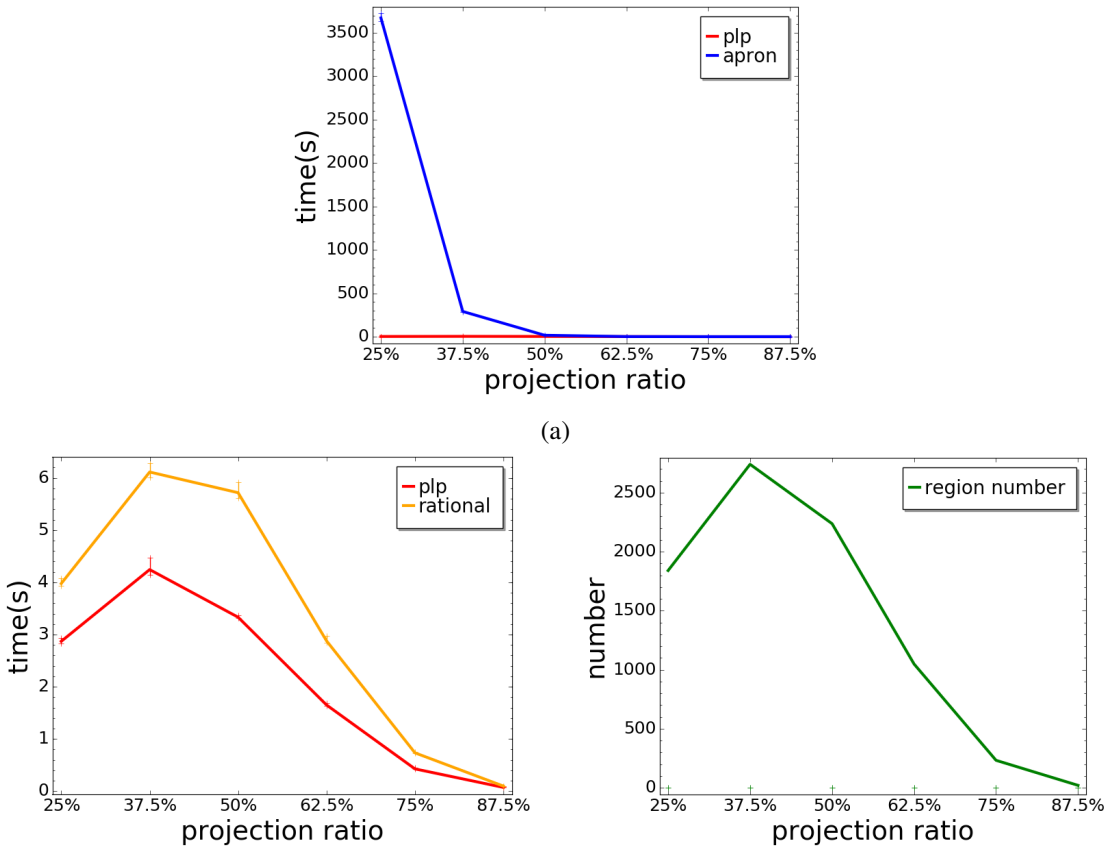
Recall that in order to obtain the projection of a convex polyhedron $\mathcal{P}$ defined by constraints, Apron (and all libraries based on the same approach, including PPL) computes a generator description of $\mathcal{P}$, projects it and then computes a minimized constraint description.

Projection ratio In Figure 6.10(a) we study the impact of the projection ratio. We can see that the execution time of PLP is almost the same for all the cases, whereas that of Apron changes significantly. Apron spends a lot of time on computing the generator description of each polyhedron. We plot the execution time of PLP (Figure 6.10(b)) and the number of regions (Figure 6.10(c)), which vary with the same trend. That means the cost of our approach depends on the number of regions to be explored.

The more variables are eliminated, the lower dimension the projected polyhedron has. Then the cost of Chernikova's algorithm for converting from the generators into the constraints will be less (Table **??**). This explains why Apron is slow when the projection ratio is low, and becomes faster when the number of variables to be eliminated is larger.

Number of constraints Fixing the other parameters, we increase the number of constraints from 12 to 30. The result is shown in Figure 6.11. We can see that Apron is faster than PLP when constraints are

[2] The minimization is still computed in floating-point numbers.



(a)

(b)

(c)

Figure 6.10 – C=19, V=8, D=37.5%, PR=[25%,87.5%]

fewer than 19; beyond that, its execution time increases significantly. In contrast, the execution time of PLP grows much more slowly.

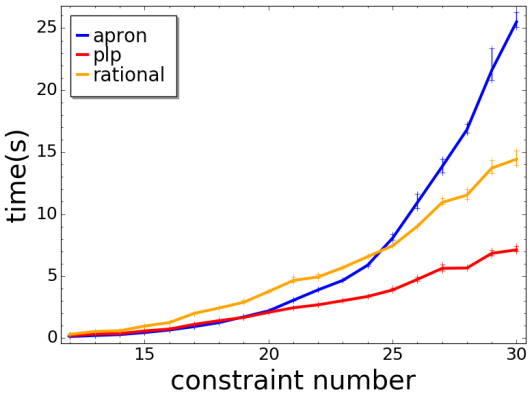
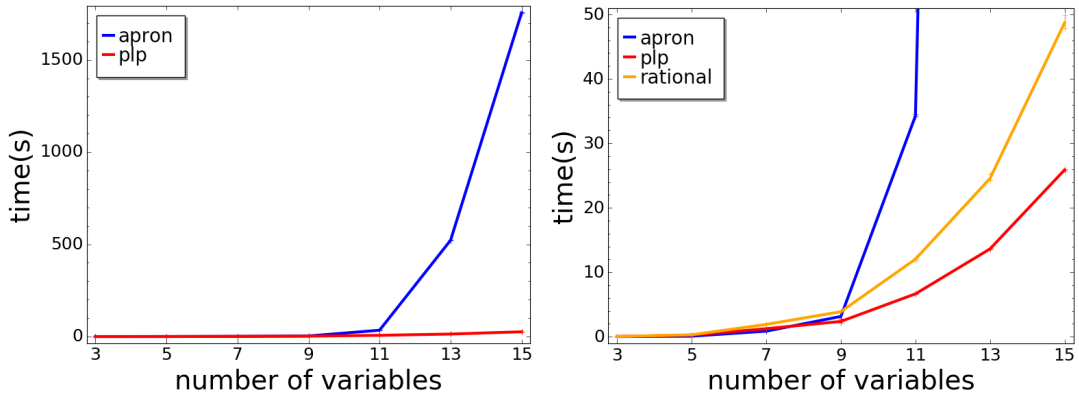


Figure 6.11 – C=[12,30], V=8, D=37.5%, PR=62.5%

Number of variables In Figure 6.12(a) the range of variables is 3 to 15. It shows that the performance are similar for Apron and PLP when variables are fewer than 11, but beyond that the execution time of Apron explodes as the number of variables increases. The zoomed figure is shown in Figure 6.12(b). This experiment shows that in high dimensions our algorithm using PLP solver is more efficient.



(a) (b)

Figure 6.12 – C=19, V=[3,15], D=37.5%, PR=62.5%

Density The Figure 6.13 shows the effect of density. The execution time varies for both Apron and PLP with the increase of density, with the same trend.

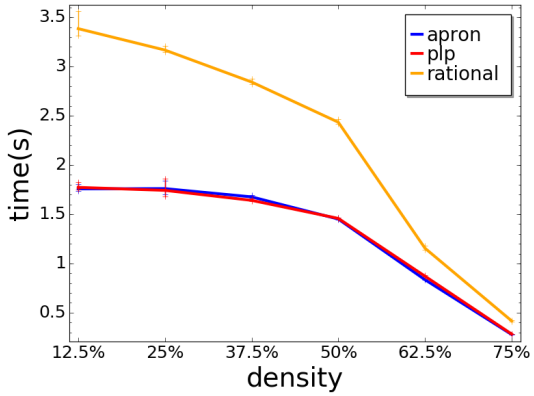


Figure 6.13 – C=19, V=8, D=[12.5%,75%], PR=62.5%

6.3.2 Experiments on SV-COMP benchmarks

In this experiment we used the analyzer Pagai [41] and SV-COMP benchmarks [5]. We randomly selected some C programs from the category of Concurrency Safety, Software System and Reach Safety. Then we used Pagai to analyze the program and extracted the polyhedra to be projected. We compute the projection on these polyhedra, and compare the results of our PLP algorithm on projection, NewPolka

Program	Apron (ms)	ELINA (ms)	PLP (ms)	ACN	AVN	AR
pthread-complex-buffer	0.29	0.18	0.32	3.25	3.06	2
ldv-linux-3.0-module-loop	**0.57**	0.22	**0.37**	3.16	**16.19**	2
ssh-clnt-01	0.29	0.18	0.32	3.53	2.61	1.98
ldv-consumption-firewire	0.45	0.28	0.7	7.19	6.14	3.2
busybox-1.22.0-head3	0.75	0.38	1.31	10.94	6.14	5.5
ldv-linux-3.0-usb-input	0.27	0.16	0.29	3	2	2
bitvector-gcd	0.33	0.19	0.73	5	3	4
array-example-sorting	0.33	0.2	0.63	4.78	3.67	2.89
ldv-linux-3.0-bluetooth	**255.66**	2.46	**12.46**	**20.62**	**17.66**	18.86
ssh-srvr-01	0.43	0.24	1.2	5.91	4.68	4.61

Table 6.4 – Performance of PLP algorithm on projection on SV-COMP benchmarks.

of Apron and ELINA (introduced in Chapter 1). In Table 6.4, we present the name of programs, the average time spent on projection in milliseconds, the average number of constraints (ACN), variables (AVN) and regions (AR).

PAGAI was designed to minimize the complexity of the polyhedra and thus it takes a small subset of the variables into the invariants; relationship with other variables are ignored. Thus in the benchmarks most polyhedra have small number of constraints and dimensions. As a result of our experiments on the random generated polyhedra, our approach does not have advantage in this case. As it is shown, our algorithm has advantage over Apron when the polyhedra contain more constraints and/or in higher dimension, e.g, polyhedra in ldv-linux-3.0-module-loop and ldv-linux-3.0-bluetooth, as we get rid of maintaining double description. ELINA is the most efficient on SV-COMP benchmarks.

6.3.3 Analysis

We conclude that our approach has remarkable advantage over Apron for projecting polyhedra having large number of constraints or/and variables; but it is not a good choice for solving problems with few constraints in low dimensions. Our sequential algorithm is less efficient than ELINA on SV-COMP benchmarks, as the number of constraints/dimensions of the polyhedra is small, in which case our approach does not have advantage.

6.4 Convex hull via sequential PLP

In this section we present several experiments on computing the convex hull of two polyhedra. We first perform the experiments on our randomly generated benchmarks, then on the SV-COMP benchmarks.

We use V to denote the number of variables and $C1$ and $C2$ to denote the number of constraints of the two polyhedra respectively. We can see in Figure 6.14 and Figure 6.15 that when the polyhedra having a small number of constraints and variables, the NewPolka library of Apron is more efficient than our approach using PLP. When the size of polyhedra is larger, for instance in Figure 6.16 and Figure 6.17 the algorithm of PLP solver has better performance. We obtain a similar result to the experiments on projection: our algorithm of PLP solver has advantage for the polyhedra having a large number of constraints and variables.

Table 6.5 presents the results of experiments on the random selected programs from SV-COMP benchmarks. Our PLP algorithm is slower than both Apron and ELINA on these benchmarks, in which the polyhedra have a small number of constraints and variables.

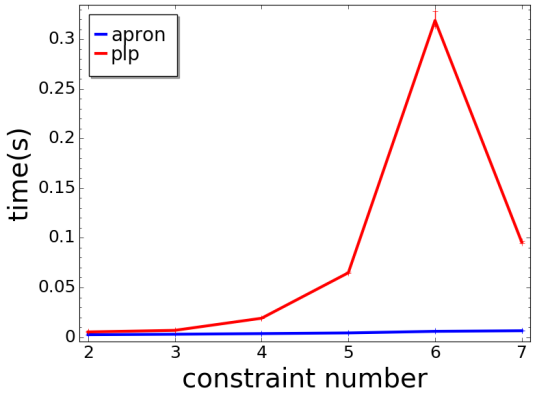


Figure 6.14 – $CN1$=4, $CN2$=[2,7], V=4.

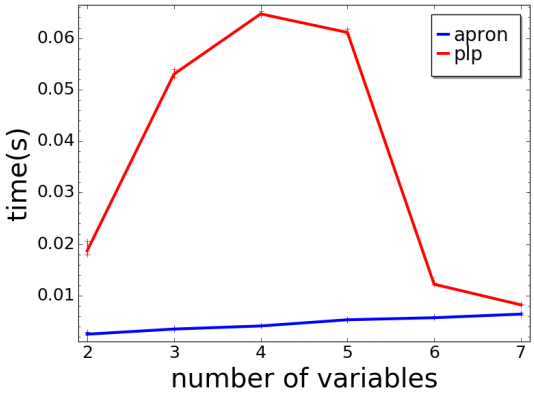


Figure 6.15 – $CN1$=4, $CN2$=5, V=[2,7].

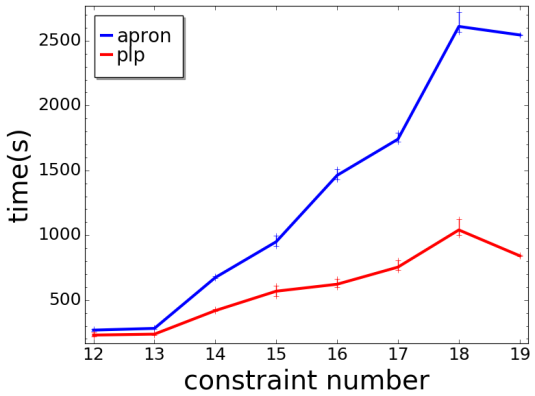


Figure 6.16 – $CN1$=15, $CN2$=[12,19], V=6.

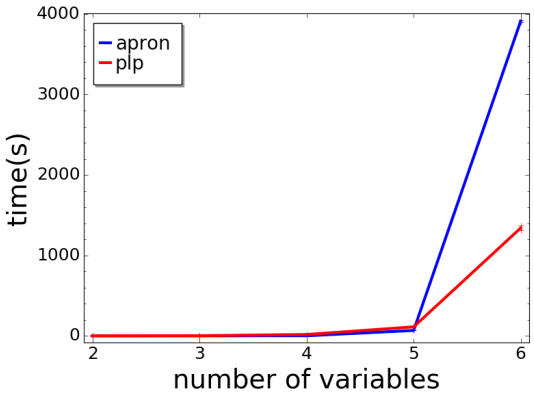


Figure 6.17 – $CN1$=20, $CN2$=15, V=[2,6].

Program	Apron (ms)	ELINA (ms)	PLP (ms)	ACN	AVN	AR
array-memsafety	0.28	0.18	1.98	2.63, 2.63	2.5	6.75
busybox-1.22.0	0.34	0.21	0.73	3.5, 2.5	7	3.5
bitvector	0.31	0.19	0.5	2.0, 1.5	4.5	1.5
loops	0.28	0.19	0.9	2.88, 2.88	3.0	4.88
ldv-linux-3.0-atm	0.43	0.26	1.36	4.5, 4.5	7.5	5.5
loop-invgen	0.38	0.24	9.89	5.8, 5.9	3.4	21.5

Table 6.5 – Performance PLP algorithm on convex hull on SV-COMP benchmarks.

6.5 Parallel PLP

In this section we report some experiments of our parallel algorithm of the PLP solver on projection. First we performed experiments on randomly generated benchmarks. Secondly we used the selected SV-COMP benchmarks.

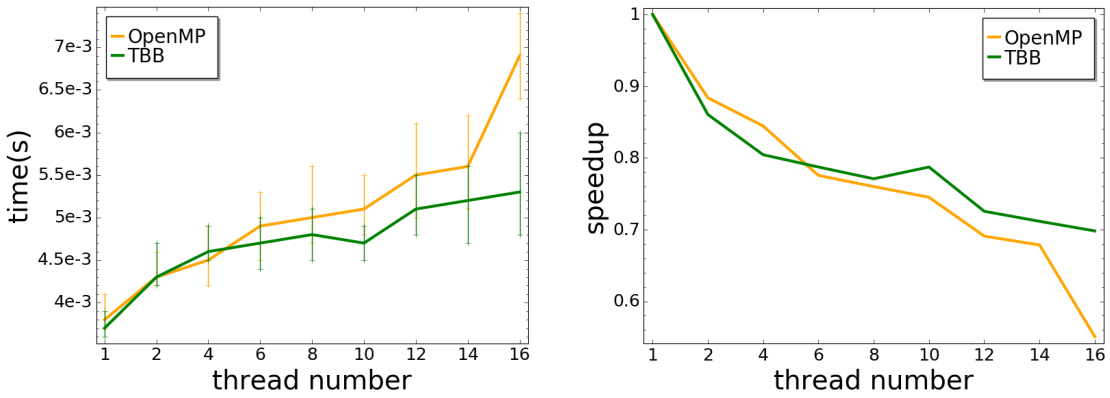
6.5.1 Randomly generated benchmarks

6.5.1.1 Performance on various numbers of constraints and dimensions

In the first experiment, we consider polyhedra in very low dimensions. We project out 1 variable from polyhedra with 4 constraints and 2 variables. Figure 6.18 shows that the execution time increases slowly with the number of threads. As we sum up the running time of 10 polyhedra with the same parameters, the average running time of one polyhedron is only 0.35 to 0.7 milliseconds. Our algorithm takes time to construct the PLP and to preprocess the constraints (Gaussian elimination, eliminating implicit constraints, etc.) which are sequential. The parallelized approaches then solve the LP problems and reconstruct the rational results. When the LP problems are small, the parallel steps take a low proportion of the total execution time, and thus the parallelism cannot speed up the algorithm significantly. Besides, the average number of regions of the projected polyhedra is 2, and thus there are not enough tasks for all the threads. With the increase of the number of threads, there is extra cost for scheduling the tasks and managing the shared data. Thus the performance becomes worse when the number of threads gets larger.

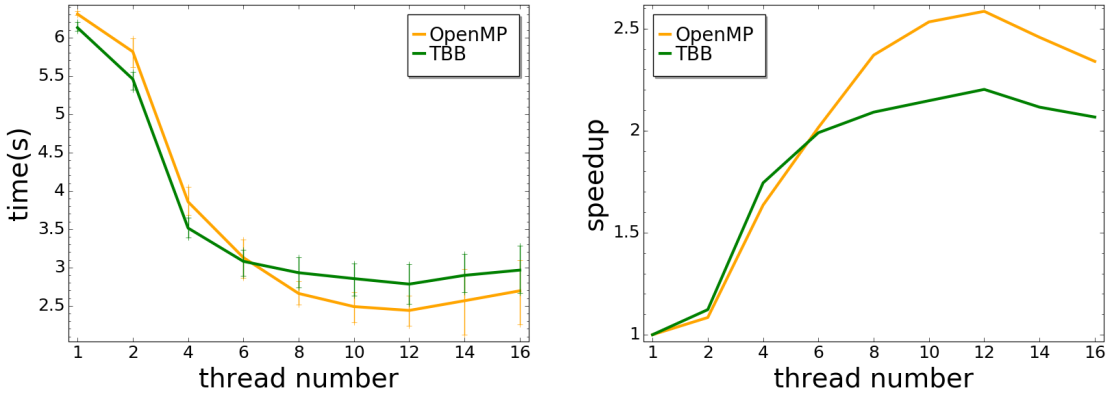
Now let us see the results of performance on the polyhedra having 25 constraints in 15 dimensions. We can see that in Figure 6.19, in both versions the performance basically remains the same when the number of threads is greater than 8. Our understanding is that when there are not enough tasks, increasing thread number cannot improve the performance.

The results on polyhedra with 100 constraints and 50 variables are shown in Figure 6.20. The two versions have similar performance. With 12 threads, the speedup factor reaches 4. The performance of parallelism is mainly limited by the total number of regions (or the number of tasks in the work list) and



(a) Performance. (b) Speedup.

Figure 6.18 – Variation of redundancy, C=4, V=2, D=0%, eliminating 1 variable. The average number of regions is 2.



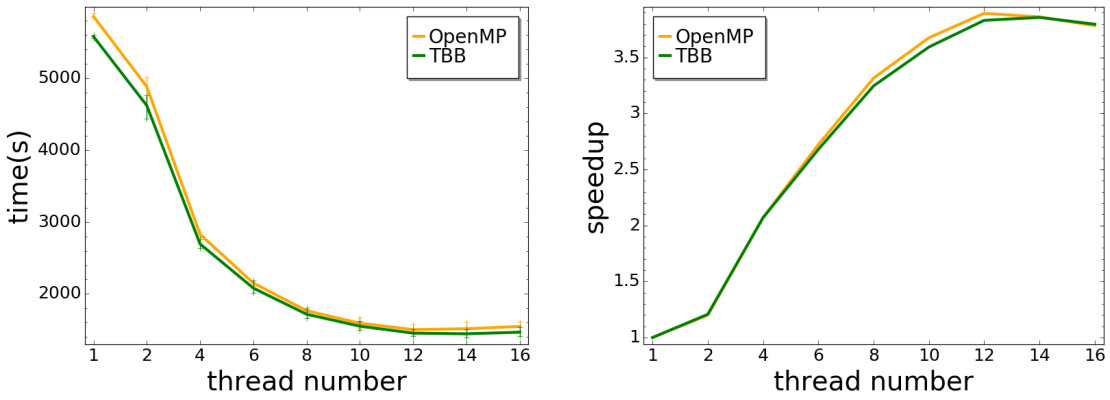
(a) Performance. (b) Speedup.

Figure 6.19 – C=25, V=15, D=20%, eliminating 1 variable. The average number of regions is 172.5.

computations of degenerated regions. If the number of tasks is less than that of threads, there will be some threads waiting for tasks, and thus launching more threads will not improve performance. The degenerated regions, which are regions corresponding to the same optimal function, are computed by one thread in our approach (shown in Algorithm 4). When there are a great amount of degenerated regions, the one thread computing on them may take more time than other threads, i.e., this thread may complete its tasks much later than the others. In this case the total execution time depends on the thread computing the degenerated regions.

6.5.1.2 Generation graph of regions

Let us see the generation of regions for analyzing the impact factors of the parallelism. We perform the experiments on a polyhedron having 12 constraints in 10 dimension. The performance and speedup are shown in Figure 6.21. We can see that the performance improves while the number of threads increases



(a) Performance.

(b) Speedup

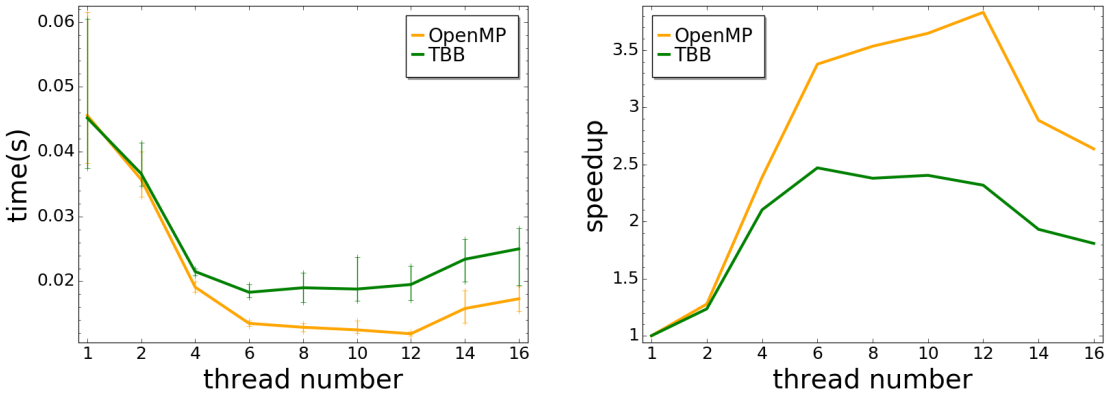
Figure 6.20 – C=100, V=50, D=20%, eliminating 1 variable. Average region number is 2928.4.

from 1 to 6. Beyond that the execution time tends to stabilize. The execution time even slightly increases when the number of threads is greater than 12.

In the generation graphs in Figure 6.22 to Figure 6.33, a node represents one region. A region is generated from the region above it, and the two regions are connected by a solid or dotted line. The solid line represents the region below is computed by the task point generated by the corresponding region above. The degenerated regions are connected by dotted lines. Please note that the numbers labeling do not matter, and they are just assigned according to the generation order. What is of interest is the shape of the graph, which depends on the geometry of the regions (how many adjacent regions do they have), and on the selection of the next exploration point in the work list. The width of the graph captures the possibility to keep the threads busy.

Figure 6.22 and Figure 6.23 illustrate the generation graph with 1 thread. It is likely that TBB uses a last-in-first-out strategy to manage the tasks, and OpenMP uses first-in-first-out strategy. In TBB version a region is computed with a point generated by the latest obtained region, and in OpenMP version the threads tend to take the oldest tasks, i.e., the ones generated by the region 0.

From 1 thread to 6 threads (Figure 6.22 to Figure 6.26, Figure 6.29 to Figure 6.31), it is likely that all the threads are busy. In Figure 6.27 and Figure 6.32 the region 0 generated 11 task points, and one of the 12 threads needs to wait for a task. In the case of using more threads(Figure 6.28, Figure 6.33), there will be more threads waiting for tasks. On one hand not all the threads are used when there are not enough tasks, and on the other hand TBB/OpenMP needs to manage the shared data, concurrent data structures, etc, which takes time. The speedup of both versions are likely to be limited by the computation of degenerated regions when they use 16 threads, which are shown in Figure 6.28 and Figure 6.33.



(a) Performance.

(b) Speedup.

Figure 6.21 – C=12, V=10, D=20%, eliminating 1 variable.

Figure 6.22 – C=12, V=10, D=20%, running with 1 thread, TBB.

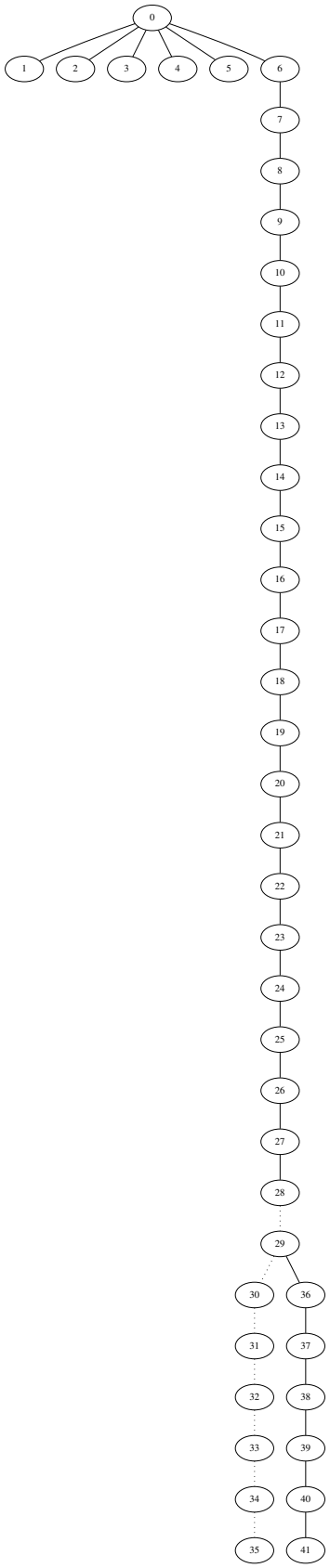


Figure 6.23 – C=12, V=10, D=20%, running with 1 thread, OpenMP.

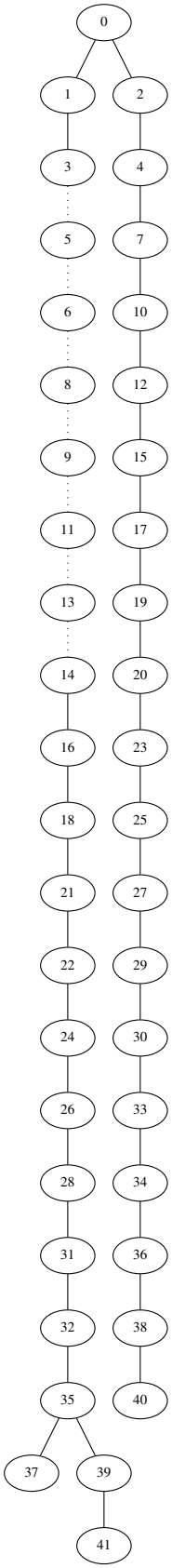


Figure 6.24 – C=12, V=10, D=20%, running with 2 threads, TBB.

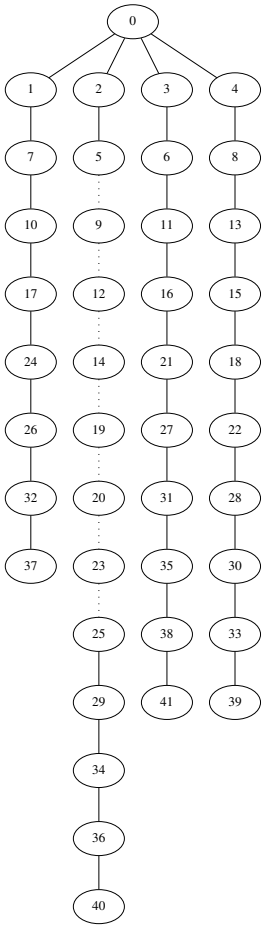


Figure 6.25 – C=12, V=10, D=20%, running with 4 threads, TBB.

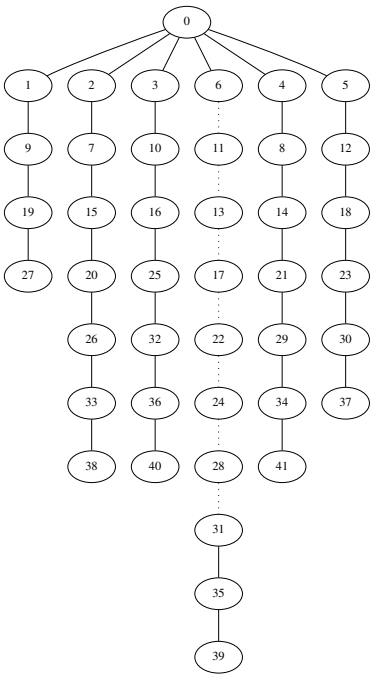


Figure 6.26 – C=12, V=10, D=20%, running with 6 threads, TBB.

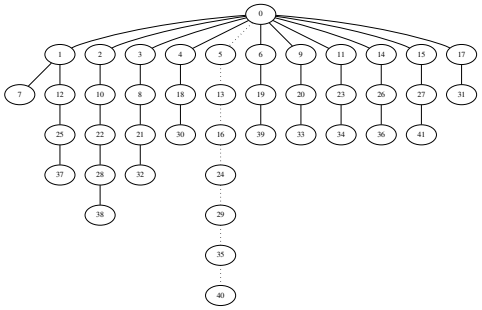


Figure 6.27 – C=12, V=10, D=20%, running with 12 threads, TBB.

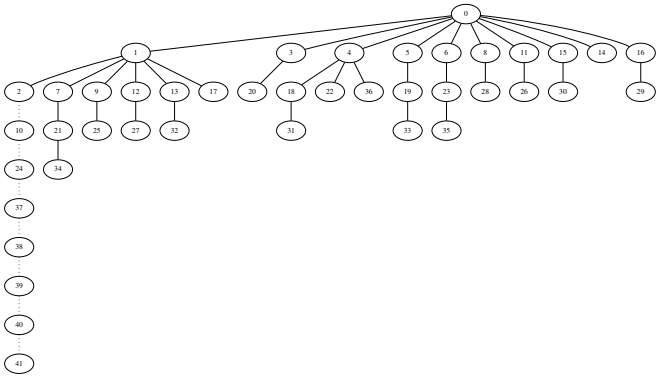


Figure 6.28 – C=12, V=10, D=20%, running with 16 threads, TBB.

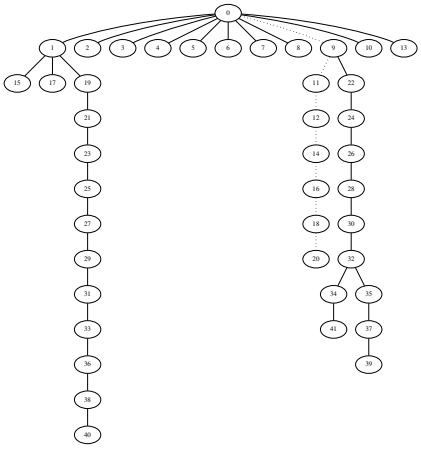


Figure 6.29 – C=12, V=10, D=20%, running with 2 threads, OpenMP.

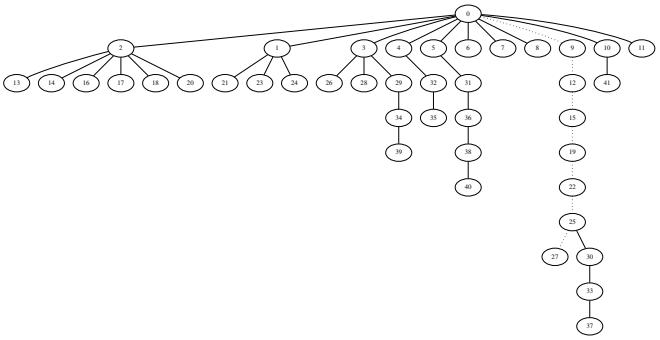


Figure 6.30 – C=12, V=10, D=20%, running with 4 threads, OpenMP.

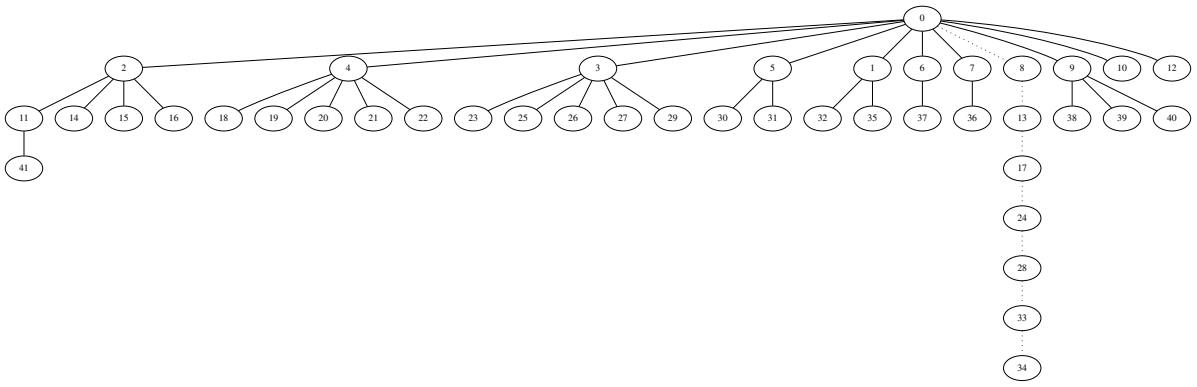


Figure 6.31 – C=12, V=10, D=20%, running with 6 threads, OpenMP.

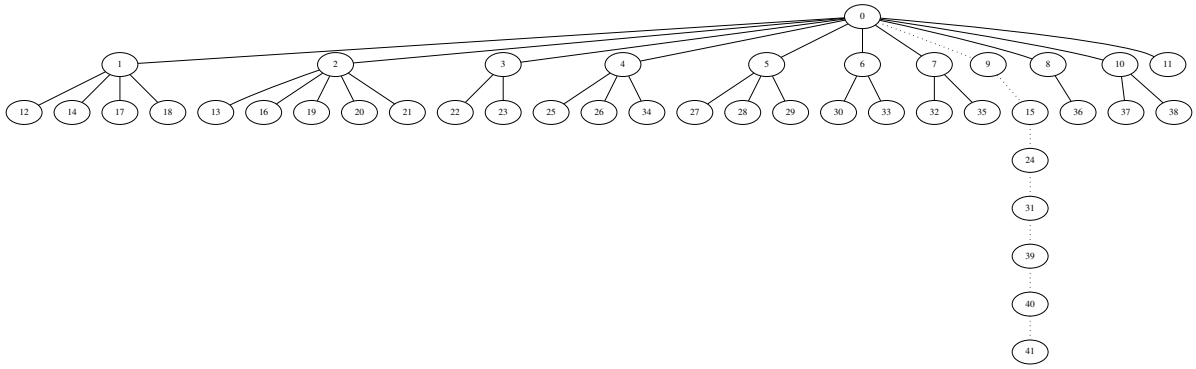


Figure 6.32 – C=12, V=10, D=20%, running with 12 threads, OpenMP.

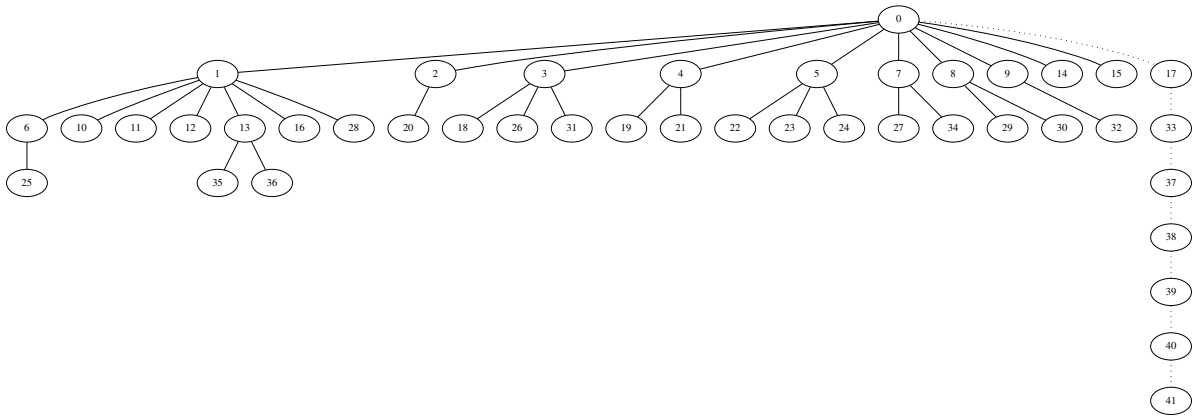
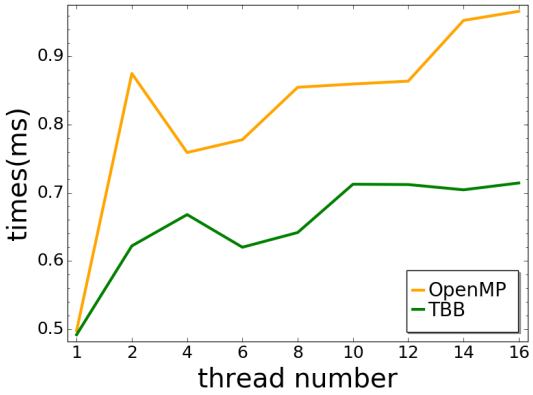


Figure 6.33 – C=12, V=10, D=20%, running with 16 threads, OpenMP.

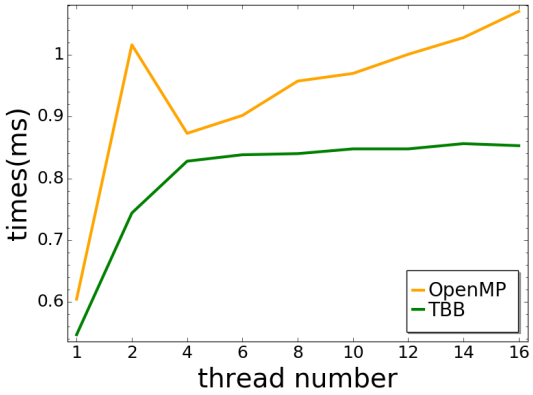
6.5.2 SV-COMP benchmarks

We used the same selected SV-COMP benchmarks as for the experiments of the sequential algorithm for computing projection. In this experiment, we show the average execution time of 5 executions on the same polyhedron, i.e. $\frac{\text{total running time}}{\text{number of polyhedron} \times 5}$.

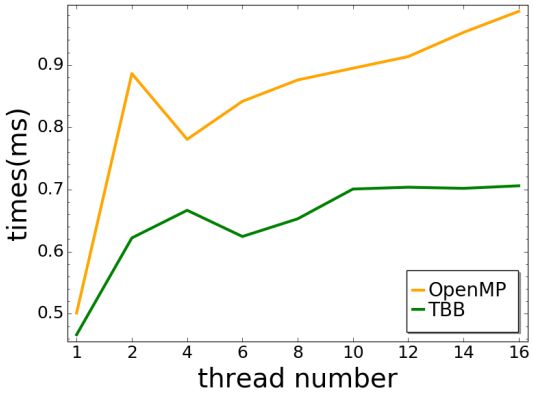
Most of the polyhedra in these benchmarks exhibit a small number of regions, as it is shown in Table 6.4, for most of them the performance becomes worse as the number of threads increases (some of them are slightly improved at first) (shown in Figure 6.34(a) to Figure 6.34(i)). These results are similar to our experiment on randomly generated polyhedra which obtained 2 projected regions on average (Figure 6.18). The parallelism on ldv-linux-3.0-bluetooth (Figure 6.34(j)) shows good performance, as the projected polyhedra on these programs have a significant number of regions.



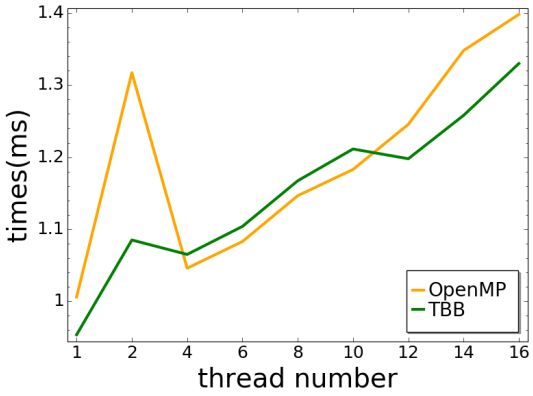
(a) pthread-complex-buffer



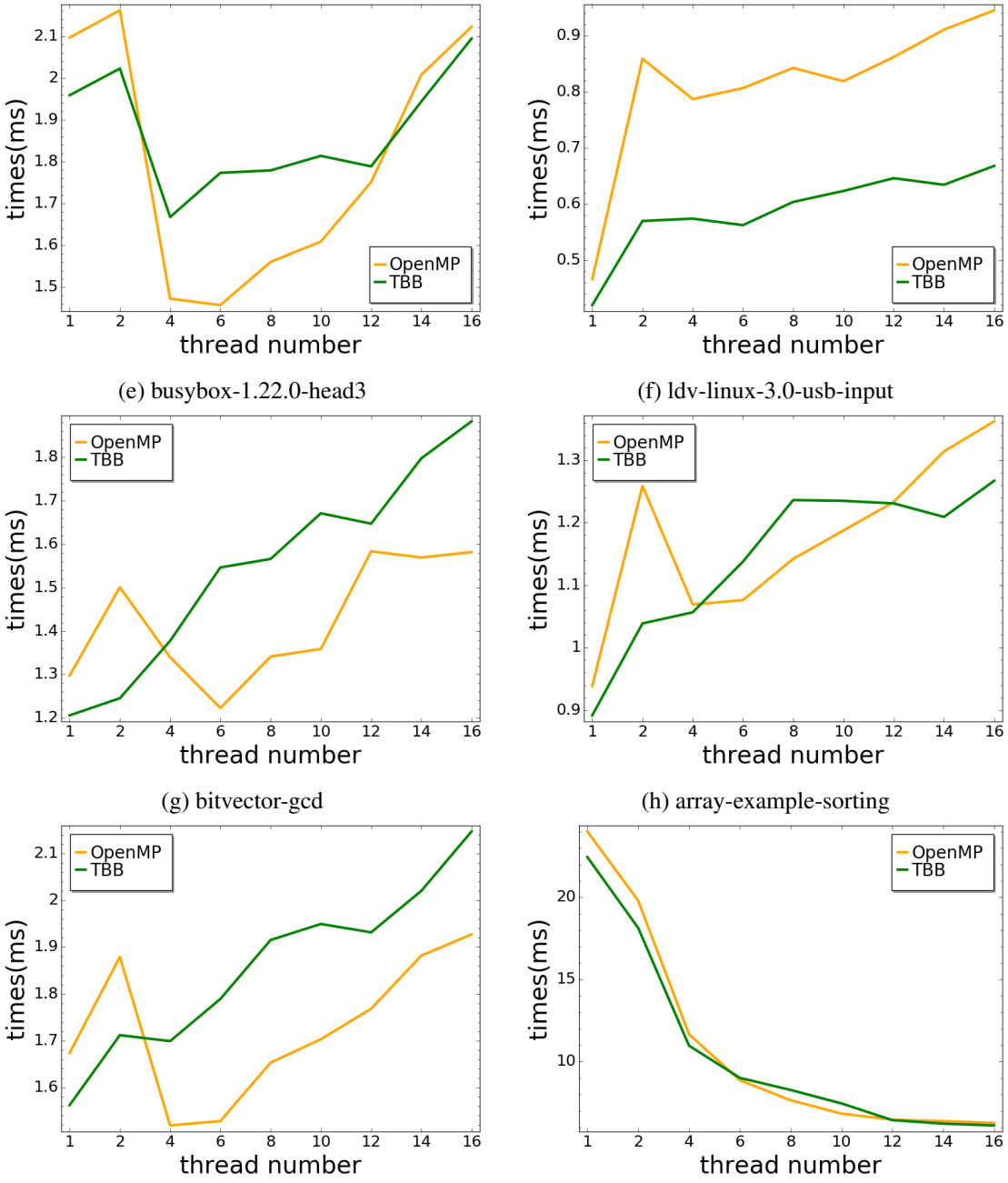
(b) ldv-linux-3.0-module-loop



(c) ssh-clnt-01



(d) ldv-consumption-firewire



(e) busybox-1.22.0-head3

(f) ldv-linux-3.0-usb-input

(g) bitvector-gcd

(h) array-example-sorting

(i) ssh-srvr-01

(j) ldv-linux-3.0-bluetooth

Figure 6.34

6.6 Conclusion

We did a set of experiments on minimization, projection and convex hull. The raytracing minimization using floating point arithmetic has a big advantage over using rational numbers. When the redundancy ratio is low, raytracing minimization is more efficient. We use it in our PLP algorithm for minimizing the obtained regions, as they contain more irredundant constraints than redundant ones.

Our PLP solver improved the efficiency significantly by using floating point arithmetic instead of rational computations. Compared to NewPolka library which used double description of polyhedra, our approach are more efficient when solving problems in high dimensions. Our approach also has an advantage for solving big problems which contains a large number of constraints. Our algorithm is less efficient for computing small sized polyhedra.

For solving the problems which are not very small, the parallelism improved the performance obviously. But it is not a good strategy to use parallel algorithm for solving small problems, which means when the number of obtained regions is small, as there will be not enough tasks for all the threads and considerable time will be spent on scheduling the tasks and managing the shared data. Thus we conclude that it is a good strategy to use our PLP solver for computing the projection/convex hull of the polyhedra having a large number of constraints and/or variables. It is better to use the parallel algorithm in this case, as when the constraint number/variable number is large, the obtained polyhedron tends to have a large number of regions.

Chapter 7

Future Work

In this chapter we talk about the future work. We will first explain some possible improvements and supplements to the current version of the PLP solver. Then we will show that it is possible to implement a pure floating point PLP solver for computing projection and convex hull, if the rational result is not required. Without rational procedures the soundness of the result can be guaranteed, i.e., the resulting polyhedron is over-approximate, but the precision could be lower. For other operators, the pure floating point solver cannot obtain a sound result, and we need new algorithms to compute these operators using pure floating point arithmetic.

7.1 Improving the current algorithm of the PLP solver

In this section we explain the possible improvements on our PLP solver. First the method of checking adjacency could be more efficient. Second the PLP solver could be adapted for solving more general PLP problems which do not have the normalization constraint.

7.1.1 The approach for checking adjacency

We currently store the regions that have been explored into an unstructured array; checking whether an optimization direction is covered by an existing region is done by a linear search. This could be improved in two ways: i) regions corresponding to the same optimum (primal degeneracy) could be merged into a single region; ii) regions could be stored in a structure allowing fast search. For instance, we could use a binary tree where each node is labeled with a hyperplane, and each path from the root corresponds to a conjunction of half-spaces; then each region is stored only in the paths such that the associated half-spaces intersects the region.

7.1.2 Dual degeneracy in general PLP problems

For more general cases, i.e., PLP problems without the normalization constraint, we need to deal with the dual degeneracy. We have two strategies: i) applying perturbation method to the objective function;

ii) transforming the problem into its dual form and apply our method to solve primal degeneracy to the dual problem.

Perturbing the objective function The perturbated objective function is shown in Equation 7.1. We can choose a small value for ε_i, and then the LP problem with instantiated objective function can be solved by GLPK. But this will lead to numerical problem. Another possibility is to deal with $\boldsymbol{\varepsilon}$ as "infinitesimal" and compare the ratio in lexico-order, as what we did for perturbing the constraints of the PLP. But the disadvantage is that we cannot use GLPK to solve the LP problems and we need to add the perturbation terms to the rational matrix, as GLPK does not support the perturbation method.

There is a third method. When the dual degeneracy occurs, one or multiple coefficients of the nonbasic variables in the objective function equal to zero. We can pick up these variables, say $x_i, \cdots x_j$, and optimize the obtained objective function furthermore in each direction of $x_i, \cdots x_j$ one by one. Each optimization is based on the result of the previous one. Then by solving several optimization problems, a unique optimal vertex will be found.

$$Z(\boldsymbol{\lambda}, \boldsymbol{x}) = \sum_{i=1}^{m} (\sum_{j=1}^{n} a_{ij} x_j - b_i + \varepsilon_i) \lambda_i \tag{7.1}$$

Dual problem Given an LP problem in Problem 7.2, the corresponding dual problem is shown in Problem 7.3. The primal problem does not have dual degeneracy if its dual problem does not have primal degeneracy. Hence we can add perturbation terms to the right-hand side of the constraints of the dual problem for avoiding primal degeneracy, and thus the original problem can get rid of dual degeneracy.

$$\begin{aligned} \text{maximize} \quad & Z(\boldsymbol{\lambda}) = \sum_{j=1}^{n} o_j \lambda_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} \lambda_j \leq b_i \quad \forall i \in \{1, \cdots m\} \\ \text{and} \quad & \boldsymbol{\lambda} \geq 0 \end{aligned} \tag{7.2}$$

$$\begin{aligned} \text{minimize} \quad & Z(\boldsymbol{y}) = \sum_{i=1}^{m} b_i y_i \\ \text{subject to} \quad & \sum_{i=1}^{m} a_{ij} y_i \geq o_i \quad \forall i \in \{1, \cdots m\} \\ \text{and} \quad & \boldsymbol{y} \geq 0 \end{aligned} \tag{7.3}$$

7.2 Towards a pure floating point algorithm of PLP solver

In this section we talk about the possibility of implementing a pure floating point PLP solver for solving projection and convex hull without impact on soundness.

For the situation where the exact solution is not required, it is possible to get rid of the rational approaches and obtain a sound floating point solution, i.e., the resulting polyhedron contains all the possible states that can be reached by the program in running time. We remove the rational processes from the flowchart in Figure 3.1 and obtain the flowchart of pure floating point algorithm, which is shown in Figure 7.1.

To obtain a sound result, the obtained polyhedron could lose some faces, but it should not contain any face that narrows the correct polyhedron. We have seen that in the raytracing minimization, some irredundant constraints could be removed. Thus the witness point of the corresponding constraint will never be added, and there will be a risk of missing regions, i.e. the obtained polyhedron may lose some faces. Even thought the obtained polyhedron is sound.

Two processes in the PLP solver which were computed in rational numbers need to be replaced by floating point arithmetic: checking adjacency and verifying the feasibility of the result obtained from GLPK. The criterion of checking adjacency is that: i) the regions which are not adjacent could be judged as adjacent; ii) the regions which are adjacent should never be judged as nonadjacent. If the nonadjacent regions are misjudged as adjacent, we will end up with missing the region between them, and the final result will be larger but sound. But if the adjacent regions are misjudged as nonadjacent, we will always try to add an extra task point between them, and the algorithm may loop forever.

The objective function of the PLP problem is a non-negative combination of a set of constraints, and we expected to obtain an over-approximated polyhedron. However, as what we already said, GLPK may obtain a solution which is in fact infeasible. In this case some of the variables have negative value, and thus the obtained polyhedron is not a non-negative combination anymore, which results in an unsound result. Therefore, we need to guarantee the feasibility of the solution from GLPK using a floating point checker.

7.3 Other operators in the polyhedra domain using floating point arithmetic

We saw that the results of minimization, projection and convex hull can be guaranteed to be sound using pure floating point computations. Now we talk about the inclusion and equality test.

By our raytracing method and PLP solver, we can guarantee the soundness of eliminating redundancy, meet, join and projection operators. Our approach does not implement the widening operator, so we will not talk about it. Let us see the other operators.

The inclusion operator can be implemented by minimization: considering two polyhedra $\mathcal{P}$ and $\mathcal{P}'$, $\mathcal{P} \sqsubseteq \mathcal{P}'$ if $\forall \mathcal{C}'_i \in \mathcal{P}'$, $\mathcal{C}'_i$ is redundant with respect to $\mathcal{P}$. But we cannot guarantee the soundness

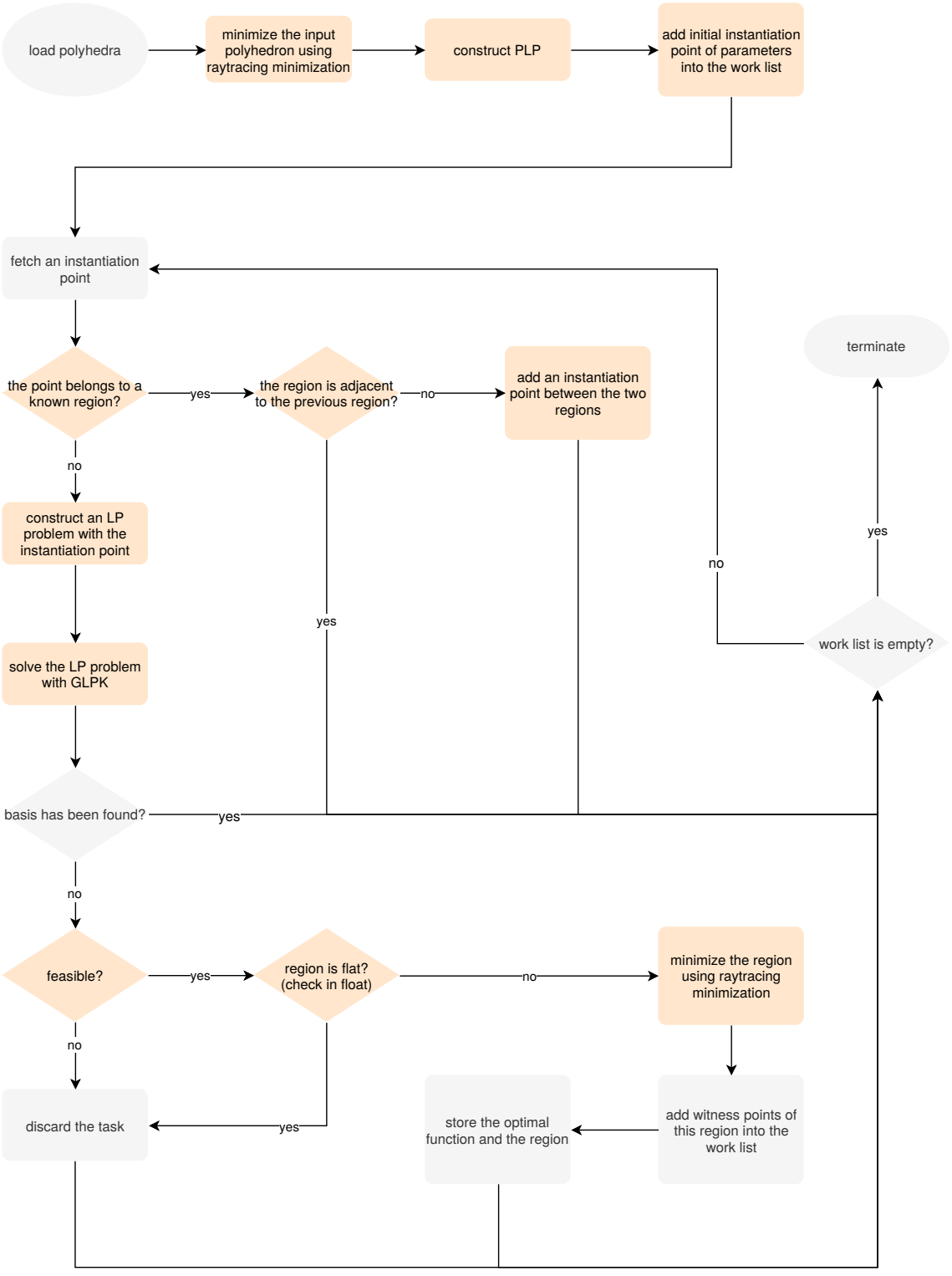


Figure 7.1 – Flowchart of pure floating point PLP procedure

of the inclusion operator computed by our floating point minimization, as our strategy is to report an undetermined constraint as redundant to ensure the soundness of minimization. When we test if $\mathcal{P}$ is included in $\mathcal{P}'$ by our strategy, the approach may report $\mathcal{P} \sqsubseteq \mathcal{P}'$ when in fact $\mathcal{P} \not\sqsubseteq \mathcal{P}'$. An analyzer using polyhedra would consider that $\mathcal{P}'$ is a sound approximation of the reachable states. Thus some states of $\mathcal{P}$ which are not included in $\mathcal{P}'$ could be reached at running time.

The equality test can be implemented by double inclusion test. Similarly, we may report two polyhedra are equal, which are in fact unequal, and thus it is unsound.

As a conclusion, we cannot guarantee the soundness of inclusion and equality test using floating point minimization with the current strategy. We need to explore new methods for these operators.

Acknowledgements

First I would like to express my special thanks to my supervisors Dr. David Monniaux and Dr. Michaël Périn, who spare no effort to help me and inspire me during the three years. They are always so patient with my buggy code and the reports having annoying structures. They not only teach me the knowledge, but also show me the method to present the work and the way to do research. I also want to present my great thank to Dr. Laurent Mounier, who helped me so much with my first attempt to teach. Then I want to show my special thank to my families, who always support me at any time, and my two cats, who always sit beside my laptop and try to type some sentences when I was writing my thesis in the night. I would like to thank all my friends and colleague in Verimag for their help and kindness.

Bibliography

[1] Alexandre Maréchal. *New Algorithmics for Polyhedral Calculus via Parametric Linear Programming*. Theses, UGA - Université Grenoble Alpes, December 2017.

[2] David Monniaux. On using floating-point computations to help an exact linear arithmetic decision procedure. In *International Conference on Computer Aided Verification*, pages 570–583. Springer, 2009.

[3] Tim King, Clark Barrett, and Cesare Tinelli. Leveraging linear and mixed integer programming for smt. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, pages 139–146. FMCAD Inc, 2014.

[4] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, 4.5 edition, November 2015.

[5] Dirk Beyer. Automatic verification of c and java programs: Sv-comp 2019. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 133–155. Springer, 2019.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[7] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.

[8] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[9] Nicolas Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. PhD thesis, Université Scientifique et Médicale de Grenoble & Institut National Polytechnique de Grenoble, March 1979.

[10] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, 1976.

[11] Roberto Bagnara, Patricia M Hill, and Enea Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1):3–21, 2008.

[12] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *International Conference on Computer Aided Verification*, pages 661–667. Springer, 2009.

[13] Gagandeep Singh, Markus Püschel, and Martin Vechev. Fast polyhedra abstract domain. In *ACM SIGPLAN Notices*, volume 52, pages 46–59. ACM, 2017.

[14] NV Chernikova. Algorithm for discovering the set of all the solutions of a linear programming problem. *USSR Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.

[15] Hervé Le Verge. A note on Chernikova's algorithm. Technical Report 635, IRISA, 1992.

[16] Alexis Fouilhé. *Revisiting the abstract domain of polyhedra: constraints-only representation and formal proof.* PhD thesis, Université Grenoble Alpes, 2015.

[17] George B Dantzig. Fourier-motzkin elimination and its dual. Technical report, STANFORD UNIV CA DEPT OF OPERATIONS RESEARCH, 1972.

[18] Colin N Jones, Eric C Kerrigan, and Jan M Maciejowski. On polyhedral projection and parametric programming. *Journal of Optimization Theory and Applications*, 138(2):207–220, 2008.

[19] Jacob M Howe and Andy King. Polyhedral analysis using parametric objectives. In *International Static Analysis Symposium*, pages 41–57. Springer, 2012.

[20] Alexandre Maréchal, David Monniaux, and Michaël Périn. Scalable minimizing-operators on polyhedra via parametric linear programming. In *International Static Analysis Symposium*, pages 212–231. Springer, 2017.

[21] Nicolas Halbwachs, David Merchat, and Laure Gonnord. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods in System Design*, 29(1):79–95, 2006.

[22] Liqian Chen, Antoine Miné, and Patrick Cousot. A sound floating-point polyhedra abstract domain. In *Asian Symposium on Programming Languages and Systems*, pages 3–18. Springer, 2008.

[23] Nicholas J Higham. *Accuracy and stability of numerical algorithms*, volume 80. Siam, 2002.

[24] George B Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Activity analysis of production and allocation*, 13:339–347, 1951.

[25] George B Dantzig and Mukund N Thapa. *Linear programming 2: theory and extensions*. Springer Science & Business Media, 2006.

[26] Branko Grünbaum, Victor Klee, Micha A Perles, and Geoffrey Colin Shephard. Convex polytopes. 1967.

[27] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[28] Vasek Chvatal, Vaclav Chvatal, et al. *Linear programming*. Macmillan, 1983.

[29] Alexandre Maréchal and Michaël Périn. Efficient elimination of redundancies in polyhedra by ray-tracing. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 367–385. Springer, 2017.

[30] Hang Yu and David Monniaux. An efficient parametric linear programming solver and application to polyhedral projection. In *International Static Analysis Symposium*, pages 203–224. Springer, 2019.

[31] Leonid Khachiyan. Fourier–motzkin elimination method. *Encyclopedia of Optimization*, pages 1074–1077, 2009.

[32] Florence Benoy, Andy King, and Frédéric Mesnard. Computing convex hulls with a linear solver. *Theory and Practice of Logic Programming*, 5(1-2), 2005.

[33] George B Dantzig and Mukund N Thapa. *Linear programming 1: introduction*. Springer Science & Business Media, 2006.

[34] Tomas Gal. *Postoptimal Analyses, Parametric Programming, and Related Topics: degeneracy, multicriteria decision making, redundancy*. Walter de Gruyter, 2010.

[35] Robert G Bland. New finite pivoting rules for the simplex method. *Mathematics of operations Research*, 2(2):103–107, 1977.

[36] George B Dantzig. Application of the simplex method to a transportation problem. *Activity Analysis and Production and Allocation*, 1951.

[37] George B Dantzig, Alex Orden, Philip Wolfe, et al. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

[38] Horst A Eiselt and C-L Sandblom. *Linear programming and its applications*. Springer Science & Business Media, 2007.

[39] Colin N Jones, Eric C Kerrigan, and Jan M Maciejowski. Lexicographic perturbation for multiparametric linear programming with applications to control. *Automatica*, 43(10):1808–1816, 2007.

[40] Camille Coti, David Monniaux, and Hang Yu. Parallel parametric linear programming solving, and application to polyhedral computations. In *International Conference on Computational Science*, pages 566–572. Springer, 2019.

[41] Julien Henry, David Monniaux, and Matthieu Moy. Pagai: A path sensitive static analyser. *Electronic Notes in Theoretical Computer Science*, 289:15–25, 2012.

Index

Abstract

VPL (Verified Polyhedra Library) is an abstract polyhedra domain using constraint-only description. All main operators boiled down to polyhedral projection, which can be computed using Fourier-Motzkin elimination [20]. This method generates many redundant constraints which must be eliminated at a high cost. A novel algorithm was implemented in VPL for computing the polyhedral projection using *parametric linear programming* (PLP) [34], which can replace Fourier-Motzkin elimination. This PLP solver can also be used for computing the join operator (convex hull). Our work focuses on improving the efficiency of the algorithm of PLP solver.

In prior work, PLP [39, 1] was done in arbitrary precision rational arithmetic. In this work, we present an approach where most of the computations are performed in floating-point arithmetic, and then exact rational results are reconstructed. The result obtained from our approach is guaranteed to be sound. We also propose a workaround for a difficulty called *degeneracy*, which plagued previous attempts at using PLP for computations on polyhedra: in general the (parametric) linear programming problems are degenerated, resulting in redundant computations and duplicates in geometric descriptions. The algorithm of the PLP solver is intrinsically parallelable, however it was developed in VPL with OCaml, which does not well support parallelism programming. In our approach, which is implemented with C++, we proposed a task-based scheme for parallelizing it. Two parallel implementations have been realized using two different parallel mechanisms: Intel's Thread Building Blocks (TBB)[1] and OpenMP tasks [4].

Résumé

VPL (Verified Polyhedra Library) est un domaine de polyhèdres abstraits utilisant une description uniquement par contrainte. Tous les opérateurs principaux se résumaient à une projection polyédrique pouvant être calculée à l'aide de l'élimination de Fourier-Motzkin. Cette méthode génère de nombreuses contraintes redondantes qui doivent être éliminées à un coût élevé. Un nouvel algorithme a été mis en œuvre dans VPL pour calculer la projection polyédrique à l'aide de la programmation linéaire paramétrique (PLP), qui peut remplacer l'élimination de Fourier-Motzkin. Ce solveur PLP peut également être utilisé pour calculer l'opérateur de jointure (coque convexe). Notre travail est axé sur l'amélioration de l'efficacité de l'algorithme du solveur PLP.

Dans des travaux antérieurs, le PLP était effectué en arithmétique rationnelle de précision arbitraire. Dans ce travail, nous présentons une approche dans laquelle la plupart des calculs sont effectués en arithmétique en virgule flottante, puis les résultats rationnels exacts sont reconstruits. Le résultat obtenu grâce à notre approche est assuré d'être solide. Nous proposons également une solution de contournement à une difficulté appelée dégénérescence, qui entachait les tentatives précédentes d'utilisation de PLP pour les calculs sur polyèdres : en général, les problèmes de programmation linéaire (paramétrique) sont

[1] https://www.threadingbuildingblocks.org/

dégénérés, ce qui entraîne des calculs redondants et des doublons dans les descriptions géométriques. L'algorithme du solveur PLP est intrinsèquement parallèle, mais il a été développé en VPL avec OCaml, qui ne prend pas bien en charge la programmation en parallélisme. Dans notre approche, qui est implémentée avec C ++, nous avons proposé un schéma basé sur les tâches pour le paralléliser. Deux implémentations parallèles ont été réalisées en utilisant deux mécanismes parallèles différents : les blocs de construction de threads (TBB) d'Intel et les tâches OpenMP.