Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

12-11-2018 2:30 PM

Complexity Results for Fourier-Motzkin Elimination

Delaram Talaashrafi, *The University of Western Ontario*

Supervisor: Moreno Maza, Marc, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science
© Delaram Talaashrafi 2018

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Part of the Theory and Algorithms Commons

Recommended Citation
Talaashrafi, Delaram, "Complexity Results for Fourier-Motzkin Elimination" (2018). *Electronic Thesis and Dissertation Repository*. 5981.
https://ir.lib.uwo.ca/etd/5981

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.

Abstract

In this thesis, we propose a new method for removing all the redundant inequalities generated by Fourier-Motzkin elimination. This method is based on Kohler's work and an improved version of Balas' work. Moreover, this method only uses arithmetic operations on matrices. Algebraic complexity estimates and experimental results show that our method outperforms alternative approaches based on linear programming.

Keywords: polyhedron, projection, Fourier-Motzkin elimination, linear algebra, polyhedral cone, parametric linear programming

Acknowledgments

First and foremost, I would like to offer my sincerest gratitude to my supervisor Professor Marc Moreno Maza, I am very thankful for his great advice and support. It is my honor to have Professor Graham Denham, Professor Roberto Solis-Oba, and Professor Kaizhong Zhang as the examiners. I am grateful for their insightful comments and questions. I would like to thank the members of Ontario Research Center for Computer Algebra and the Computer Science Department of the University of Western Ontario. Specially, I am thankful to my colleague Dr. Robert Moir for proofreading chapters of my thesis. Finally, I am very thankful to my family and friends for their endless support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computations with polyhedral sets play an important role in the analysis, transformation and scheduling of for-loops of computer programs. We refer to the following articles as entry points to this subject [2, 3, 4, 9, 10, 15, 33]. Of prime importance are the following operations on polyhedral sets: (i) representation conversion of polyhedral sets, between H-representation and V-representation, (ii) projection of polyhedral sets, namely Fourier-Motzkin elimination and block elimination, (iii) redundancy removal, for which most works use linear programming.

Fourier-Motzkin elimination is an algorithmic tool for projecting a polyhedral set on a linear subspace. It was proposed independently by Joseph Fourier and Theodore Motzkin, in 1827 and in 1936. The naive version of this algorithm produces large amounts of redundant inequalities and has a double exponential algebraic complexity. Removing all these redundancies is equivalent to give a minimal representation of the projected polyhedron. Leonid Khachiyan explained in [24] how linear programming (LP) could be used to remove all redundant inequalities, then reducing the cost of Fourier-Motzkin elimination to singly exponential time; however, Khachiyan does not give any running time estimate.

Instead of using linear programming, we are hoping to use matrix arithmetic operations in order to increase theoretical and practical efficiency of Fourier-Motzkin elimination while maintaining the requirement of an irredundant representation of the projected polyhedron.

As we have mentioned above, the so-called *block elimination method* is another algorithmic tool to project a polyhedral set. This method needs to enumerate the extreme rays of a cone. Many authors have been working on this topic, see Nataĺja V. Chernikova [7], Hervé Le Verge [26] and Komei Fududa [13]. Other algorithms aiming at projecting polyhedral sets remove some (but not all) redundant inequalities with the help of extreme rays: see the work of David A. Kohler [25]. As observed by Jean-Louis Imbert in [18], the method he proposed in this paper and that of Sergei N. Chernikov in [6] are equivalent. These methods are very effective in practice, but none of them can remove all redundant inequalities generated by Fourier-Motzkin Elimination. Egon Balas in [1] proposed a method to overcome that latter limitation. However, we found flaws in both his construction and its proof.

In this thesis, we show how to remove all the redundant inequalities generated by Fourier-Motzkin Elimination combining Kohler's work and an improved version of Balas' work. Moreover, our method has a better algebraic complexity estimate than the approaches using linear programming.

Our main contributions are:

1. Based on Kohler's and Balas' works, we propose an efficient method for removing all redundancies in Fourier-Motzkin elimination;
2. We use simple matrix operations and avoid computing all the extreme rays of the so-called *test cone*, this latter task being time-consuming.
3. We give a single exponential complexity time for our method.
4. We propose a method for removing the redundant inequalities of the input system as well.
5. We implemented our method in the C language, within the BPAS library, and our experimental results show that our method outperforms competitive implementations.

This thesis is organized as follows. Chapter 2 provides background materials. Chapter 3 explains our algorithm. To be more specific, Sections 3.1.3 and 3.3 presents Kohler's and improved Balas' works respectively. We will see that Kohler's method is effective but can not guarantee to remove all the redundant inequalities. Meanwhile, Balas's method consumes more computing resources but can remove all the redundant inequalities. The main algorithm and its complexity are presented in Section 3.4. Experimental results are reported in Chapter 4. In Chapter 5, we discuss related work together with the flaws of the construction and the proof in the original work of Balas. Chapter 6 shows an application of Fourier-Motzkin elimination: solving parametric linear programming (PLP) problems, which is a core routine in the analysis, transformation and scheduling of for-loops of computer programs.

This thesis is a joint work with Rui-Juan and Marc Moreno Maza, see the preprint [21].

Chapter 2

Background

In this chapter, we review the basics of polyhedral geometry. Section 2.1 is dedicated to the notions of polyhedral sets and polyhedral cones Section 2.2 states the double description method and Fourier-Motzkin elimination, which are the two of the most important algorithms operating on polyhedral sets. Finally, we conclude this background chapter with the cost model that we shall use for complexity analysis, see Section 2.3. We omit most proofs in this chapter. For more details please refer to [13, 30, 32].

2.1 Polyhedral cones and polyhedral sets

Notation 1 *We use bold letters, e.g.* $\mathbf{v}$*, to denote vectors and we use capital letters, e.g.* A*, to denote matrices. Also, we assume vectors are column vectors. For row vectors, we use the transpose notation, that is,* A^t *for the transposition of matrix* A*. For a matrix* A *and an integer* k*,* A_k *is the row of index* k *in* A*. Also, if* K *is a set of integers,* A_K *denotes the sub-matrix of* A *with indices in* K*.*

We begin this section with the fundamental theorem of linear inequalities.

Theorem 2.1 *Let* $\mathbf{a}_1, \cdots, \mathbf{a}_m$ *be a set of linearly independent vectors in* $\mathbb{R}^n$*. Also, let and* $\mathbf{b}$ *be a vector in* $\mathbb{R}^n$*. Then, exactly one of the following holds:*

(i) *the vector* $\mathbf{b}$ *is a non-negative linear combination of* $\mathbf{a}_1, \ldots, \mathbf{a}_m$*. In other words, there exist positive numbers* $y_1, \ldots, y_m$ *such that we have* $\mathbf{b} = \sum_{i=1}^{m} y_i \mathbf{a}_i$*, or,*

(ii) *there exists a vector* $\mathbf{d} \in \mathbb{R}^n$*, such that both* $\mathbf{d}^t\mathbf{b} < 0$ *and* $\mathbf{d}^t\mathbf{a}_i > 0$ *hold for all* $1 \leq i \leq m$*.*

Definition 2.1 (Convex cone) *A subset of points* $C \subseteq \mathbb{R}^n$ *is called a* cone *if for each* $\mathbf{x} \in C$ *and each real number* $\lambda \geq 0$ *we have* $\lambda\mathbf{x} \in C$*. A cone* $C \subseteq \mathbb{R}^n$ *is said* convex *if for all* $\mathbf{x}, \mathbf{y} \in C$*, we have* $\mathbf{x} + \mathbf{y} \in C$*. If* $C \subseteq \mathbb{R}^n$ *is a convex cone, then its elements are called the* rays *of* C*. For two rays* $\mathbf{r}$ *and* $\mathbf{r}'$ *of* C*, we write* $\mathbf{r}' \simeq \mathbf{r}$ *whenever there exists* $\lambda \geq 0$ *such that we have* $\mathbf{r}' = \lambda\mathbf{r}$*.*

Definition 2.2 (Hyperplane) *Let* H *be a subset of* $\mathbb{R}^n$*. It is called a hyperplane if* $H = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{a}^t\mathbf{x} = 0\}$ *for some non-zero vector* $\mathbf{a} \in \mathbb{R}^n$*.*

Definition 2.3 (Half-space) *A* half-space *is a set of the form* $\{x \in \mathbb{R}^n \mid \mathbf{a}^t x \leq 0\}$ *for a some vector* $\mathbf{a} \in \mathbb{R}^n$.

Definition 2.4 (Polyhedral cone) *A cone* $C \subseteq \mathbb{R}^n$ *is a polyhedral cone if it is the intersection of finitely many half-spaces, that is,* $C = \{x \in \mathbb{R}^n \mid Ax \leq 0\}$ *for some matrix* $A \in \mathbb{R}^{m \times n}$.

Example 2.1 *The set* $C = \{-2\,x + y + 11\,z \leq 0, x - 5\,y + 7\,z \leq 0, x + y + z \leq 0, x + 2\,y + 3\,z \leq 0\}$ *defines a polyhedral cone and Figure 2.1 is the illustration of this cone. As it can be seen in the Figure 2.1 all hyperplanes defining the cone, are intersecting at the origin.*

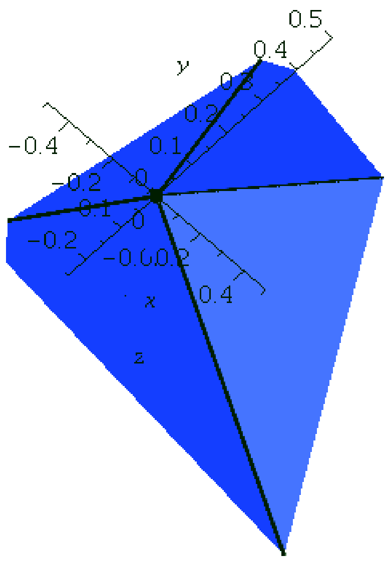

Figure 2.1: Example of a polyhedral cone

Definition 2.5 (Finitely generated cone) *Let* $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ *be a set of vectors in* $\mathbb{R}^n$. *The* cone generated *by this set, denoted by* $\mathsf{Cone}(\mathbf{x}_1, \cdots, \mathbf{x}_m)$, *is the smallest convex cone containing those vectors. In other words, we have* $\mathsf{Cone}(\mathbf{x}_1, \ldots, \mathbf{x}_m) = \{\lambda_1 \mathbf{x}_1 + \cdots + \lambda_m \mathbf{x}_m \mid \lambda_1 \geq 0, \ldots, \lambda_m \geq 0\}$. *A cone obtained in this way is called a* finitely generated cone.

Example 2.2 *The cone in Figure 2.1 can be generated by:*

$$G = \{[1, \frac{-1}{2}, \frac{-1}{2}], [-1, -\frac{25}{62}, -\frac{9}{62}], [\frac{-1}{2}, 1, \frac{-1}{2}], [-1, \frac{17}{19}, -\frac{5}{19}]\}$$

With the following lemma, which is a consequence of the fundamental Theorem of linear inequalities, we can say that the two concepts of polyhedral cones and finitely generated cones are equivalent, see [30]

Theorem 2.2 (Minkowski-Weyl theorem) *A convex cone is polyhedral if and only if it is finitely generated.*

Definition 2.6 (Convex polyhedron) *A set of vectors $P \subset \mathbb{R}^n$ is called a* convex polyhedron *if $P = \{\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b}\}$, for a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $\mathbf{b} \in \mathbb{R}^m$. Moreover, the polyhedron P is called a* polytope *if P is bounded,*

Example 2.3 *The set*

$$P = \{-10\,x + y - 2\,z \leq 6, x + 2\,y + z \leq 5, 3\,x - 5\,y + z \leq 15, 6\,x + 15\,y - 9\,z \leq 14, 21\,x - 17\,y + 7\,z \leq 84\}$$

defines a convex polyhedron and Figure 2.2 illustrates it.

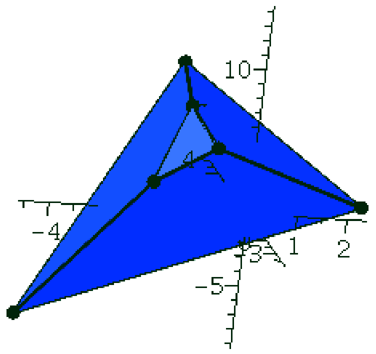

Figure 2.2: Example of a convex polyhedron

Definition 2.7 (Representation of a polyhedron) *Given a polyhedron $P = \{\mathbf{x} \in \mathbb{R}^n \mid A\mathbf{x} \leq \mathbf{c}\}$, the system of linear inequalities $\{A\mathbf{x} \leq \mathbf{c}\}$ is the* representation *of P.*

Definition 2.8 (Minkowski sum) *For two subsets P and Q of $\mathbb{R}^n$, their* Minkowski sum, *denoted by $P + Q$, is the subset of $\mathbb{R}^n$ defined as $\{p + q \mid (p, q) \in P \times Q\}$.*

The following lemma, which is another consequence of the fundamental theorem of linear inequalities, helps us to determine the relation between polytopes and polyhedrons. The proof can be found in [30]

Lemma 2.1 (Decomposition theorem for convex polyhedra) *A subset P of $\mathbb{R}^n$ is a convex polyhedron if and only if it can be written as the Minkowski sum of a finitely generated cone and a polytope.*

Another consequence of the fundamental theorem of inequalities, is the famous Farkas lemma. This lemma has different variants. Here we only mention a variant from [30], which is applicable in the next chapters and algorithms.

Lemma 2.2 (Farkas' lemma) *Let $A \in \mathbb{R}^{m \times n}$ be a matrix and $\mathbf{b} \in \mathbb{R}^m$ be a vector. Then, there exists a vector $\mathbf{t} \in \mathbb{R}^n$, $\mathbf{t} \geq \mathbf{0}$ satisfying $A\mathbf{t} = \mathbf{b}$ if and if $\mathbf{y}^t\mathbf{b} \geq 0$ holds for each vector $\mathbf{y} \in \mathbb{R}^m$ such that we have $\mathbf{y}^t A \geq 0$.*

An important consequence of Farkas' lemma is the following lemma which gives a criterion to test whether an inequality $\mathbf{bx} \leq b_0$ is *redundant* w.r.t. a polyhedron representation $A\mathbf{x} \leq \mathbf{c}$, that is, whether $\mathbf{bx} \leq b_0$ is implied by $A\mathbf{x} \leq \mathbf{c}$.

Lemma 2.3 (Redundancy test criterion) *Let $\mathbf{b} \in \mathbb{R}^n$, $b_0 \in \mathbb{R}$, $A \in \mathbb{R}^{m \times n}$ and $\mathbf{c} \in \mathbb{R}^m$. Then, the inequality $\mathbf{bx} \leq b_0$ is redundant w.r.t. the system of linear inequalities $A\mathbf{x} \leq \mathbf{c}$ if and only if there exists a vector $\mathbf{t} \geq \mathbf{0}$ and a number $\lambda \geq 0$ satisfying $\mathbf{b}^t = \mathbf{t}^t A$ and $b_0 = \mathbf{t}^t\mathbf{c} + \lambda$.*

From now on, we assume that $P = \{\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b}\}$ is a polyhedron in $\mathbb{R}^n$, with $A \in \mathbb{R}^{m \times n}$.

Definition 2.9 (Implicit equation) *An inequality $\mathbf{a}^t\mathbf{x} \leq b$ (with $\mathbf{a} \in \mathbb{R}^n$ and $b \in \mathbb{R}$) is an implicit equation of the inequality system $A\mathbf{x} \leq \mathbf{b}$ if $\mathbf{a}^t\mathbf{x} = b$ holds for all $\mathbf{x} \in P$.*

Example 2.4 *On Figure 2.3, the brown polyhedron is entirely on one side of the blue plane. This means that the blue plane defines two half-spaces, one of which is redundant for the inequality system defining the brown polyhedron.*

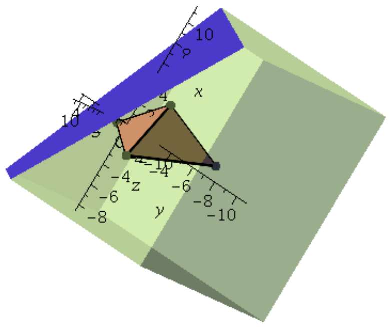

Figure 2.3: Redundant inequality illustration

Definition 2.10 (Minimal representation) *A representation of a polyhedron is minimal if there is no redundant inequality in it, that is, if no inequality of that representation is implied by the other inequalities of that representation.*

Definition 2.11 (Characteristic (recession) cone of a polyhedron) *The* characteristic cone *of P, denoted by $\mathsf{CharCone}(P)$, is the polyhedral cone defined by*

$$\mathsf{CharCone}(P) = \{\mathbf{y} \in \mathbb{R}^n \mid \mathbf{x} + \mathbf{y} \in P, \ \forall \mathbf{x} \in P\} = \{\mathbf{y} \mid A\mathbf{y} \leq \mathbf{0}\}.$$

Definition 2.12 (Lineality space and pointed polyhedron) *The* lineality space *of the polyhedron P is denoted by $\mathsf{LinearSpace}(P)$ and is the linear space*

$$\mathsf{CharCone}(P) \cap -\mathsf{CharCone}(P) = \{\mathbf{y} \mid A\mathbf{y} = \mathbf{0}\},$$

where $-\mathsf{CharCone}(P)$ *is the negation of points in* $\mathsf{CharCone}(P)$.
The polyhedron P is pointed *if its lineality space is zero.*

Lemma 2.4 (Pointed polyhedron criterion) *The polyhedron P is pointed if and only if the matrix A is full column rank.*

Remark 2.1 *When the cone* $C = \{\mathbf{x} \mid A\mathbf{x} \leq \mathbf{0}\}$ *has implicit equations, it can be written as* $C = \{\mathbf{x} \in \mathbb{R}^n \mid A_1\mathbf{x} = \mathbf{0}, A_2\mathbf{x} \leq \mathbf{0}\}$, *where* $A_1 \in \mathbb{R}^{m_1 \times n}$ *and* $A_2 \in \mathbb{R}^{m_2 \times n}$ *and* $\{A_2\mathbf{x} \leq \mathbf{0}\}$ *has no implicit equations. In this case, the cone C is pointed if and only if we have* $\mathrm{rank}(\begin{bmatrix} A_1 \\ A_2 \end{bmatrix}) = n$.

Definition 2.13 (Dimension of a polyhedron) *The dimension of the polyhedron P, denoted by* $\dim(P)$, *is* $n - r$, *where n is dimension*[1] *of the ambient space (that is,* $\mathbb{R}^n$*) and r is the maximum number of implicit equations defined by linearly independent vectors. We say that P is* full-dimensional *whenever* $\dim(P) = n$ *holds. In another words, P is full-dimensional if and only if it does not have any implicit equations.*

Definition 2.14 (Face of a polyhedron) *A subset F of the polyhedron P is called a* face *of P if and only if F equals* $\{\mathbf{x} \in P \mid A_t\mathbf{x} = b_t\}$ *for a sub-matrix* A_t *of A and a sub-vector* $\mathbf{b}_t$ *of* $\mathbf{b}$.

Remark 2.2 *It is obvious that every face of a polyhedron is also polyhedron. Moreover, the intersection of two faces* F_1 *and* F_2 *of P is another face F, which is either* F_1, *or* F_2, *or a face with a dimension less than* $\min(dim(F_1), \dim(F_2))$. *Note that P and the empty set are faces of P.*

Definition 2.15 (Facet of a polyhedron) *A face of P, distinct from P and of maximal dimension is called a* facet *of P.*

Remark 2.3 *It follows from the previous remark that P has at least one facet and that the dimension of any facet of P is equal to* $\dim(P) - 1$. *When P is full-dimensional, there is a one to one correspondence between the inequalities in a minimal representation of P and the facets of P. From this latter observation, we deduce that the minimal representation of a full dimensional polyhedron is unique up to multiplying each of the defining inequality by a positive constant.*

Definition 2.16 (Minimal face) *A non-empty face that does not contain any other face of a polyhedron is called a* minimal face *of that polyhedron. Specifically, if the polyhedron P is pointed each minimal face is just a point and is called a extreme point or vertex of the polyhedron.*

[1]Of course, this notion of dimension coincides with the topological one, that is, the maximum dimension of a ball contained in P.

Definition 2.17 (Extreme rays) *Let C be a cone such that* $\dim(\mathsf{LinearSpace}(C)) = t$. *Then, a face of C of dimension $t+1$ is called a minimal proper face of C. In the special case of a pointed cone, that is, whenever $t = 0$ holds, the dimension of a minimal proper faces is 1 and such a face is called an* extreme ray .

We call extreme ray *of the polyhedron P any extreme ray of its characteristic cone* $\mathsf{CharCone}(P)$.

We say that two extreme rays $\mathbf{r}$ and $\mathbf{r}'$ of the polyhedron P are equivalent, *and denote it by $r \simeq r'$, if one is a positive multiple of the other. When we consider the set of all extreme rays of the polyhedron P (or the polyhedral cone C) we will only consider one ray from each equivalence class.*

Lemma 2.5 (Generating a cone from its extreme rays) *A pointed cone C can be generated by its extreme rays, that is:*

$$C \;=\; \{\mathbf{x} \in \mathbb{R}^n \mid (\exists \mathbf{c} \geq \mathbf{0})\ \mathbf{x} = R\mathbf{c}\},$$

where the columns of R are the extreme rays of C.

Remark 2.4 *From the previous definitions and lemmas, we derive the following observations:*
1. *the number of extreme rays of each cone is finite,*
2. *the set of all extreme rays is unique up to multiplication by a scalar, and,*
3. *all members of a cone are positive linear combination of extreme rays.*

We denote by $\mathsf{ExtremeRays}(C)$ the set of extreme rays of the cone C. Recall that all cones considered here are polyhedral.

The following lemma is helpful in the analysis of algorithms manipulating extreme rays of cones and polyhedra.

Lemma 2.6 (Maximum number of extreme rays [27] and [32]) *Let $E(C)$ be the number of extreme rays of a polyhedral cone $C \in \mathbb{R}^n$ with m facets. Then, we have:*

$$E(C) \leq \binom{m-\lfloor\frac{n+1}{2}\rfloor}{m-1} + \binom{m-\lfloor\frac{n+2}{2}\rfloor}{m-n} \leq m^{\lfloor\frac{n}{2}\rfloor}.$$

From Remark 2.4, it appears that extreme rays are important characteristics of polyhedral cones. Therefore, two algorithms have been developed in [13] to check whether a member of a cone is an extreme ray or not. For explaining these algorithms, we need the following definition.

Definition 2.18 (Zero set of a cone) *For a cone $C = \{\mathbf{x} \in \mathbb{R}^n \mid A\mathbf{x} \leq \mathbf{0}\}$ and $\mathbf{t} \in C$, we define the* zero set $\zeta_A(\mathbf{t})$ *as the set of row indices i such that $A_i\mathbf{t} = 0$, where A_i is the i-th row of A. For simplicity, we use $\zeta(\mathbf{t})$ instead of $\zeta_A(\mathbf{t})$ when there is no ambiguity.*

Consider a cone $C = \{\mathbf{x} \in \mathbb{R}^n \mid A'\mathbf{x} = \mathbf{0},\ A''\mathbf{x} \leq \mathbf{0}\}$ where A' and A'' are two matrices such that the system $A''\mathbf{x} \leq \mathbf{0}$ has no implicit equations. The proofs of the following lemmas are straightforward and can be found in [13] and [32].

Lemma 2.7 (Algebraic test for extreme rays) *Let $\mathbf{r} \in C$. Then, the ray $\mathbf{r}$ is an extreme ray of C if and only if we have* $\mathrm{rank}(\begin{bmatrix} A' \\ A''_{\zeta(r)} \end{bmatrix}) = n - 1$.

Lemma 2.8 (Combinatorial test for extreme rays) *Let* $\mathbf{r} \in C$. *Then, the ray* $\mathbf{r}$ *is an extreme ray of* C *if and only if for any ray* $\mathbf{r}'$ *of* C *such that* $\zeta(\mathbf{r}) \subseteq \zeta(\mathbf{r}')$ *holds we have* $\mathbf{r}' \simeq \mathbf{r}$.

Definition 2.19 (Polar cone) *For the given polyhedral cone* $C \subseteq \mathbb{R}^n$, *the* polar cone *induced by* C *is denoted* C^* *and given by:*

$$C^* = \{\mathbf{y} \in \mathbb{R}^n \mid \mathbf{y}^t\mathbf{x} \leq \mathbf{0}, \forall \mathbf{x} \in C\}.$$

The following lemma shows an important property of the polar cone of a polyhedral cone. The proof can be found in [30].

Lemma 2.9 (Polarity property) *For a give cone* $C \in \mathbb{R}^n$, *there is a one to one correspondence between the faces of* C *of dimension* k *and the faces of* C^* *of dimension* $n-k$. *In particular, there is a one to one correspondence between facets of* C *and extreme rays of* C^* *and vice versa.*

Each polyhedron P can be embedded in a higher-dimensional cone, called the *homogenized cone* associated with P.

Definition 2.20 (Homogenized cone of a polyhedron) *The homogenized cone of the polyhedron* $P = \{\mathbf{x} \in \mathbb{R}^n \mid A\mathbf{x} \leq \mathbf{d}\}$, *denoted by* $\mathsf{hom}(P)$, *is defined by:*

$$\mathsf{hom}(P) = \{(\mathbf{x}, x_{last}) \in \mathbb{R}^{n+1} \mid C[\mathbf{x}, x_{last}] \leq 0\},$$

where

$$C = \begin{bmatrix} A & -\mathbf{d} \\ \mathbf{0}^t & 1 \end{bmatrix}$$

is an $(m+1) \times (n+1)$*-matrix, if* A *is an* $(m \times n)$*-matrix.*

Theorem 2.3 (Extreme rays of the homogenized cone) *Every extreme ray of the homogenized cone* $\mathsf{hom}(P)$ *associated with the polyhedron* P *is either of the form* $(\mathbf{x}, 0)$ *where* $\mathbf{x}$ *is an extreme ray of* P, *or* $(\mathbf{x}, 1)$ *where* $\mathbf{x}$ *is an extreme point of* P.

Lemma 2.10 (H-representation correspondence) *An inequality* $A_i\mathbf{x} \leq d_0$ *is redundant in* P *if and only if the corresponding inequality* $A_i\mathbf{x} - b_i x_{last} \leq 0$ *is redundant in* $\mathsf{hom}(P)$.

Corollary 2.1 *The minimal representation of* $\mathsf{hom}(P)$ *has one more facet than the minimal representation of* P *and that facet is* $\mathbf{x}_{last} \leq 0$.

2.2 Polyhedral computations

In this section, we review two of the most important algorithms for polyhedral computations: the double description algorithm (DD for short) and the Fourier-Motzkin elimination algorithm (FME for short).

A polyhedral cone C can be represented either as an intersection of finitely many half-spaces (thus using the so-called H-representation of C) or as by its extreme rays (thus using the so-called V-representation of C); the DD algorithm produces one representation from the other.

We shall explain the version of the DD algorithm which takes as input the H-representation of C and returns as output the V-representation of C.

The FME algorithm performs a standard projection of a polyhedral set to lower dimension subspace. In algebraic terms, this algorithm takes as input a polyhedron P given by a system of linear inequalities (thus an H-representation of P) in n variables $x_1 < x_2 < \cdots < x_n$ and computes the H-representation of the projection of P on $x_1 < \cdots < x_k$ for some $1 \leq k < n$.

2.2.1 The Double description method

We know from Theorem 2.2 that any polyhedral cone $C = \{\mathbf{x} \in \mathbb{R}^n \mid A\mathbf{x} \leq \mathbf{0}\}$ can be generated by finitely many vectors, say $\{\mathbf{x}_1, \ldots, \mathbf{x}_q\} \in \mathbb{R}^n$. Moreover, from Lemma 2.5 we know that if C is pointed, then it can be generated by its extreme rays, that is, $C = \mathsf{Cone}(R)$ where $R = [\mathbf{x}_1, \ldots, \mathbf{x}_q]$. Therefore, we have two possible representations for the pointed polyhedral cone C:

H-representation: as the intersection of finitely many half-spaces, or equivalently, with a system of linear inequalities $A\mathbf{x} \leq \mathbf{0}$;

V-representation: as a linear combination of finitely many vectors, namely $\mathsf{Cone}(R)$, where R is a matrix, the columns of which are the extreme rays of the cone C.

We say that the pair (A, R) is *Double Description Pair* or simply a *DD pair*. We call A a *representation matrix* of C and R a *generating matrix* of C. We call R (resp. A) a *minimal generating (resp. representing) matrix* when no proper sub-matrix of R (resp. A) is generating (resp. representing) C.

It is important to notice that, for some queries in polyhedral computations, the output can be calculated in polynomial time using one representation (either a representation matrix or a generating matrix) while it would require exponential time using the other representation.

For example, we can compute in polynomial time the intersection of two cones when they are in H-representation but the same problem would be harder to solve when the same cones are in V-representation. Therefore, it is important to have a procedure to convert between these two representations, which is the focus of the articles [7] and [32].

We will explain this procedure, which is known as the *double description method* as well as *Chernikova's algorithm*. This algorithm takes a cone in H-representation as input and returns a V-representation of the same cone as output. In other words, this procedure finds the extreme rays of a polyhedral cone, given by its representation matrix. It has been proven that this procedure runs in single exponential time. To the best of our knowledge, the most practically efficient variant of this procedure has been proposed by Fukuda in [13] and is implemented in the CDD library. We shall explain his approach here and analyze its algebraic complexity. Before presenting Fukuda's algorithm, we need a few more definitions and results. In this section, we assume that the input cone C is pointed.

The *double description method* works in an incremental manner. Denoting by $H_1, \ldots, H_m$ the half-spaces corresponding to the inequalities of the H-representation of C, we have $C = H_1 \cap \cdots \cap H_m$. Let $1 < i \leq m$ and assume that we have computed the extreme rays of the cone $C^{i-1} := H_1 \cap \cdots \cap H_{i-1}$. Then the i-th iteration of the DD method deduces the extreme rays of C^i from those of C^{i-1} and H_i.

Assume that the half-spaces $H_1, \ldots, H_m$ are numbered such that H_i is given by $A_i\mathbf{x} \le 0$, where A_i is the i-th row of the representing matrix A. We consider the following partition of $\mathbb{R}^n$:

$$H_i^+ = \{\mathbf{x} \in \mathbb{R}^n \mid A_i\mathbf{x} > 0\}$$
$$H_i^0 = \{\mathbf{x} \in \mathbb{R}^n \mid A_i\mathbf{x} = 0\}$$
$$H_i^- = \{\mathbf{x} \in \mathbb{R}^n \mid A_i\mathbf{x} < 0\}$$

Assume that we have found the DD-pair (A^{i-1}, R^{i-1}) of C^{i-1}. Let J be the set of the column indices of R^{i-1}. We use the above partition $\{H_i^+, H_i^0, H_i^-\}$ to partition J as follows:

$$J_i^+ = \{j \in J \mid \mathbf{r}_j \in H^+\}$$
$$J_i^0 = \{j \in J \mid \mathbf{r}_j \in H^0\}$$
$$J_i^- = \{j \in J \mid \mathbf{r}_j \in H^-\}$$

where $\{\mathbf{r}_j \mid j \in J\}$ is the set of the columns of R^{i-1}, thus the set of the extreme rays of C^{i-1}. For future reference, let us denote by $\mathsf{partition}(\mathsf{J}, \mathsf{A_i})$ the function which returns J^+, J^0, J^- as defined above. The following lemma explains how to obtain (A^i, R^i) from (A^{i-1}, R^{i-1}), where A^{i-1} (resp. A^i) is the sub-matrix of A consisting of its first $i-1$ (resp. i) rows. The proof can be found in [13].

Lemma 2.11 (Double description method) *Let $J' := J^+ \cup J^0 \cup (J^+ \times J^-)$. Let R^i be the $(n \times |J'|)$-matrix consisting of*

- *the columns of R^{i-1} with index in $J^+ \cup J^0$, followed by*
- *the vectors $\mathbf{r}'_{(j,j')}$ for $(j, j') \in (J^+ \times J^-)$, where*

$$\mathbf{r}'_{(j,j')} = (A_i\mathbf{r}_j)\mathbf{r}_{j'} - (A_i\mathbf{r}_{j'})\mathbf{r}_j,$$

Then, the pair (A^i, R^i) is a DD pair of C^i.

The most efficient way to start the incremental process is to choose the largest sub-matrix of A with linearly independent rows; we call this matrix A^0. Indeed, denoting by C^0 the cone with A^0 as representation matrix, the matrix A^0 is invertible and its inverse gives the extreme rays of C^0, that is:

$$\mathsf{ExtremeRays}(C^0) = (A^0)^{-1}.$$

Therefore, the first DD-pair that the above incremental step should take as input is $(A^0, (A^0)^{-1})$.

The next key point towards a practically efficient DD method is to observe that most of the vectors $\mathbf{r}'_{(j,j')}$ in Lemma 2.11 are redundant. Indeed, Lemma 2.11 leads to a construction of a generating matrix of C (in fact, this would be Algorithm 2 where Lines 13 and 16 are suppressed) producing a double exponential number of rays (w.r.t. the ambient dimension n) whereas Lemma 2.6 guarantees that the number of extreme rays of a polyhedral cone is singly exponential in its ambient dimension. To deal with this issue of redundancy, we need the notion of *adjacent* extreme rays.

Definition 2.21 (Adjacent extreme rays) *Two distinct extreme rays r and r' of the polyhedral cone C are called adjacent if they span a two dimensional face of C.* [2]

[2]We do not use the minimal face, as it used in the main reference because it makes confusion.

The following lemma shows how we can test whether two extreme rays are adjacent or not. The proof can be found in [13].

Proposition 2.1 (Adjacency test) *Let* $\mathbf{r}$ *and* $\mathbf{r}'$ *be two distinct rays of* C*. Then, the following statements are equivalent:*

1. $\mathbf{r}$ *and* $\mathbf{r}'$ *are adjacent extreme rays,*
2. $\mathbf{r}$ *and* $\mathbf{r}'$ *are extreme rays and* $\operatorname{rank}(A_{\zeta(\mathbf{r})\cap\zeta(\mathbf{r}')}) = n-2$*,*
3. *if* $\mathbf{r}''$ *is a ray of* C *with* $\zeta(\mathbf{r})\cap\zeta(\mathbf{r}')\subseteq\zeta(\mathbf{r}'')$*, then* $\mathbf{r}''$ *is a positive multiple of either* $\mathbf{r}$ *or* $\mathbf{r}'$*.*

It should be noted that the second statement is related to algebraic test for extreme rays while the third one is related to the combinatorial test.

Based on Proposition 2.1, we have Algorithm 1 for testing whether two extreme rays are adjacent or not.

Algorithm 1 adjacencyTest

```
1: Input: $(A, \mathbf{r}, \mathbf{r}')$, where $A \in \mathbb{Q}^{m\times n}$ is the representation matrix of cone $C$, $\mathbf{r}$ and $\mathbf{r}'$ are two
   extreme rays of $C$
2: Output: true if $\mathbf{r}$ and $\mathbf{r}'$ are adjacent, false otherwise
3: $\mathbf{s} := A\mathbf{r},\ \mathbf{s}' := A\mathbf{r}'$
4: let $\zeta(\mathbf{r})$ and $\zeta(\mathbf{r}')$ be set of indices of zeros in $\mathbf{s}$ and $\mathbf{s}'$ respectively
5: $\zeta := \zeta(\mathbf{r})\cap\zeta(\mathbf{r}')$
6: if $\operatorname{rank}(A_\zeta) = n-2$ then
7:     return(true)
8: else
9:     return(false)
10: end if
```

Lemma 2.12 (Strengthened lemma for the DD method) *As above, let* (A^{i-1}, R^{i-1}) *be a DD-pair and denote by* J *be the set of indices of the columns of* R^{i-1}*. Assume that* $\operatorname{rank}(A^{i-1}) = n$ *holds. Let* $J' := J^+ \cup J^0 \cup \mathrm{Adj}$*, where* Adj *is the set of the pairs* $(j, j') \in J^+ \times J^-$ *such that* $\mathbf{r}_j$*, and* $\mathbf{r}_{j'}$ *are adjacent as extreme rays of* C^{i-1}*, the cone with* A^{i-1} *as representing matrix. Let* R^i *be the* $(n\times|J'|)$*-matrix consisting of*

- *the columns of* R^{i-1} *with index in* $J^+\cup J^0$*, followed by*
- *by the vectors* $\mathbf{r}'_{(j,j')}$ *for* $(j, j')\in(J^+\times J^-)$*, where*

$$\mathbf{r}'_{(j,j')} = (A_i\mathbf{r}_j)\mathbf{r}_{j'} - (A_i\mathbf{r}'_j)\mathbf{r}_j,$$

Then, the pair (A^i, R^i) *is a DD pair of* C^i*. Furthermore, if* R^{i-1} *is a minimal generating matrix for the representation matrix* A^{i-1}*, then* R^i *is also a minimal generating matrix for the representation matrix* A^i*.*

Using Proposition 2.1 and Lemma 2.12 we can obtain Algorithm 2 [3] for computing the extreme rays of a cone.

[3]In this algorithm, A^i shows the representation matrix in step i

Algorithm 2 DDmethod

```
1: Input: a matrix $A \in \mathbb{Q}^{m\times n}$, a representation matrix of a pointed cone $C$
2: Output: $R$, the minimal generating matrix of $C$
3: let $K$ be the set of indices of $A$'s independent rows
4: $A^0 := A_K$
5: $R^0 := (A^0)^{-1}$
6: let $J$ be set of column indices of $R^0$
7: while $K \neq \{1, \cdots, m\}$ do
8:     select a $A$-row index $i \notin K$
9:     $J^+,\ J^0,\ J^- := partition(J, A_i)$
10:    add vectors with indices in $J^+$ and $J^0$ as columns to $R^i$
11:    for $\mathbf{r}_p \in J^+$ do
12:        for $\mathbf{r}_n \in J^-$ do
13:            if $adjacencyTest(A^{i-1}, \mathbf{r}_p, \mathbf{r}_n) = true$ then
14:                $r_{new} := (A_i\mathbf{r}_n)\mathbf{r}_p - (A_i\mathbf{r}_p)\mathbf{r}_n$
15:                add $\mathbf{r}_{new}$ as columns to $R^i$
16:            end if
17:        end for
18:    end for
19:    let $J$ be set of indices in $R^i$
20: end while
```

2.2.2 Fourier-Motzkin elimination

Definition 2.22 (Projection of a polyhedron) *Let $A \in \mathbb{R}^{m\times p}$ and $B \in \mathbb{R}^{m\times q}$ be matrices. Let $\mathbf{c} \in \mathbb{R}^m$ be a vector. Consider the polyhedron $P \subseteq \mathbb{R}^{p+q}$ defined by $P = \{(\mathbf{u}, \mathbf{x}) \in \mathbb{R}^{p+q} \mid A\mathbf{u} + B\mathbf{x} \leq \mathbf{c}\}$. We denote by $\mathsf{proj}_{\mathbf{x}}(P)$ the projection of P on $\mathbf{x}$, that is, the subset of $\mathbb{R}^q$ defined by*

$$\mathsf{proj}_{\mathbf{x}}(P) = \{\mathbf{x} \in \mathbb{R}^q \mid \exists\, \mathbf{u} \in \mathbb{R}^p,\ (\mathbf{u}, \mathbf{x}) \in P\}.$$

Example 2.5 *Consider the polyhedron defined by $\{x + 2y - z \leq 2, 2x - 3y + 6z \leq 2, -2x + 3y + 4z \leq 20\}$. Its projection on $[y, z]$ is the polyhedron represented by $\{z \leq \frac{11}{5},\ y + \frac{2}{7}z \leq \frac{24}{7}\}$. Figure 2.4 shows this polyhedron and its projection.*

Fourier-Motzkin elimination (FME for short) is an algorithm computing the projection of a polyhedron in the sense of Definition 2.22. The key point of that algorithm is that projecting a polyhedron to lower dimension is equivalent to eliminating variables from its H-representation. Hence, FME works by successively eliminating the $\mathbf{u}$-variables (following the notations introduced in Definition 2.22 from the inequality system $A\mathbf{u} + B\mathbf{x} \leq \mathbf{c}$ to finally get the desired projection. This process shows that $\mathsf{proj}_{\mathbf{x}}(P)$ is also a polyhedron. Before explaining the main theorem and the algorithm, we need some definitions.

Definition 2.23 (Inequality combination) *Let ℓ_1 and ℓ_2 be two inequalities $a_1x_1 + \cdots + a_nx_n \leq d_1$ and $b_1x_1 + \cdots + b_nx_n \leq d_2$. Let $1 \leq i \leq n$ such that the coefficients a_i and b_i of x_i in ℓ_1 and ℓ_2 are respectively positive and negative. We define the (positive)* combination *of ℓ_1 and ℓ_2 with respect to x_i as:*

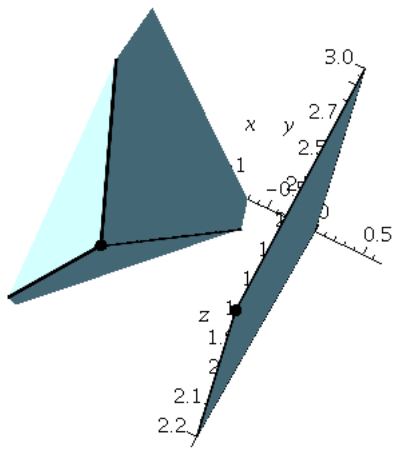

Figure 2.4: Illustration of a polyhedron and its projection

$$\mathsf{Combine}(\ell_1, \ell_2, x_i) = -b_i(a_1x_1 + \cdots + a_nx_n) + a_i(b_1x_1 + \cdots + b_nx_n) \le -b_id_1 + a_id_2.$$

In order to explain the algorithms in the next chapter, we need to assign a so-called *history set* to each inequality occurring during the execution of the FME algorithm. We denote the history set of the inequality ℓ by $\eta(\ell)$. Initially, that is, before the FME algorithm starts, the history set of an inequality is equal to its corresponding row index in the representing matrix of the input polyhedron. Now, let ℓ be the inequality resulting from the combination of the inequalities ℓ_1 and ℓ_2 in the sense of Definition 2.23. Then we define $\eta(\ell) = \eta(\ell_1) \cup \eta(\ell_2)$.

Theorem 2.4 (Fourier-Motzkin theorem [25]) *Let $A \in \mathbb{R}^{m\times n}$ be a matrix and let $\mathbf{c} \in \mathbb{R}^m$ be a vector. Consider the polyhedron $P = \{\mathbf{x} \in \mathbb{R}^n \mid A\mathbf{x} \le \mathbf{c}\}$. Let S be the set of inequalities defined by $A\mathbf{x} \le \mathbf{c}$. Also, let $1 \le i \le n$. We partition S according to the sign of the coefficient of x_i:*

$$S^+ = \{\ell \in S \mid coeff(\ell, x_i) > 0\}$$
$$S^- = \{\ell \in S \mid coeff(\ell, x_i) < 0\}$$
$$S^0 = \{\ell \in S \mid coeff(\ell, x_i) = 0\}.$$

We construct the following system of linear inequalities:

$$S' = \{\mathsf{Combine}(s_p, s_n, x_i) \mid (s_p, s_n) \in S^+ \times S^-\} \ \cup \ S^0.$$

Then, S' is a representation of $\mathsf{proj}_{x_i}(P)$.

Proof of this theorem is straightforward and can be found in[25].

Using Theorem 2.4, we can develop algorithm Algorithm 3 for projecting polyhedron.

Algorithm 3 Original FME

```
1: Input: $(S_0, \mathbf{u})$, where $S_0$ is the representation of $P = \{(\mathbf{u}, \mathbf{x}) \mid A\mathbf{u} + B\mathbf{x} \le \mathbf{c}\}$ and $\mathbf{u}$ is the vector of $p$ elements of variables to be eliminated
2: Output: $S_p$, a representation of $\mathsf{proj}_{\mathbf{x}}(P)$;
3: for $k$ from 1 to $p$ do
4:     (Partition) make $S^+_{k-1}$, $S^-_{k-1}$, $S^0_{k-1}$, subsets of $S_{k-1}$ consisting of inequalities with positive, negative and zero coefficient w.r.t. $u_k$, respectively.
5:     for $\ell_p \in S^+$ do
6:         for $\ell_n \in S^-$ do
7:             $\ell := \mathsf{Combine}(\ell_p, \ell_n, u_k)$
8:             add $\ell$ to $S'_k$
9:         end for
10:    end for
11:    $S_k := S'_k \cup S^0_{k-1}$
12: end for
13: return $(S_p)$.
```

Example 2.6 *Consider the system S_0 to be*

$$
\begin{aligned}
&2x_1 + 2x_2 - 2x_3 + 2x_4 - 2x_5 + x_6 - x_8 + x_{10} \le -2\\
&-2x_1 - x_2 - 2x_3 + x_4 - 2x_6 - 2x_8 + x_9 + 2x_{10} \le 0\\
&-x_2 - x_3 - 2x_4 + 2x_6 + 2x_7 + x_8 - x_{10} \le -2
\end{aligned}
$$

Eliminating x_1 partitions S_0 to:

$$
\begin{aligned}
S_1^+ &= \{2x_1 + 2x_2 - 2x_3 + 2x_4 - 2x_5 + x_6 - x_8 + x_{10} \le -2\}\\
S_1^- &= \{-2x_1 - x_2 - 2x_3 + x_4 - 2x_6 - 2x_8 + x_9 + 2x_{10} \le 0\}\\
S_1^0 &= \{-x_2 - x_3 - 2x_4 + 2x_6 + 2x_7 + x_8 - x_{10} \le -2\}
\end{aligned}
$$

Combining inequalities in S_1^+ and S_1^- we obtain:

$$S_1 = \{x_2 - 4x_3 + 3x_4 - 2x_5 - x_6 - 3x_8 + x_9 + 3x_{10} \le -2, -x_2 - x_3 - 2x_4 + 2x_6 + 2x_7 + x_8 - x_{10} \le -2\}$$

From Algorithm 3, we know that, in the worst case, each of the positive and negative sets contains half of the inequalities. That is, if the initial inequality set has m inequalities, in the process of eliminating the first variable, each of the positive and negative sets counts $\frac{m}{2}$ inequalities. Therefore, the representation of the projection is given by $m + (\frac{m}{2})^2$ inequalities. Continuing this process, after eliminating p variables, the projection would be given by $O((\frac{m}{2})^{2^d})$ inequalities. Thus, the algorithm is *double exponential* in d. On the other hand, from [28] and [20], we know that the maximum number of facets of the projection on $\mathbb{R}^{n-p}$ of a polyhedron in $\mathbb{R}^n$ with m facets is $O(m^{\lfloor n/2 \rfloor})$. Hence, it can be concluded that most of the generated inequalities by Algorithm 3 are *redundant*. Eliminating these redundancies is the main subject of the subsequent chapters.

2.3 Cost Model

We will use the following notations and concepts for analyzing the algebraic complexity of algorithms.

Let R be an Euclidean domain. In practice, the domain R is either the ring $\mathbb{Z}$ of integers or the field $\mathbb{Q}$ of rational numbers. For all $a,\ b \in R$, we will need the following operations:

- $\mathrm{Arith}_{+,-,\star,=}(a,b)$ which returns $a+b$, $a-b$, $a \cdot b$, true if $a=0$ and false otherwise;
- $\mathrm{Quo}(a,b)$ which, for $b \neq 0$, returns $q \in \mathbb{R}$ such that $0 \leq a - qb < |b|$;
- $\mathrm{Div}(a,b)$ which, for $b \neq 0$ dividing a, returns $v \in R$ such that $bv = a$ holds (if $a = 0$ choose $v = 0$).

Consider the important special case $R = \mathbb{Z}$. Let k be a non-negative integer. We denote by $\mathcal{M}(k)$ an upper bound for the number of bit operations required for performing any of the basic operations of type Arith and Quo on input $a, b \in \mathbb{Z}$ with $|a|, |b| < 2^k$. Using the multiplication algorithm of Arnold Schönhage and Volker Strassen [29] one can choose $\mathcal{M}(k) \in O(k \log k \log\log k)$.

We also need complexity estimates for some matrix operations. For positive integers a, b, c, Let us denote by $\mathcal{MM}(a,b,c)$ an upper bound for the number of arithmetic operations (on the coefficients) required for multiplying an $(a\times b)$ matrix by an $(b\times c)$ matrix. In the case of square matrices of order n, we simply write $\mathcal{MM}(n)$ instead of $\mathcal{MM}(n,n,n)$. We denote by θ the exponent of linear algebra, that is, the smallest real positive number such that $\mathcal{MM}(n) \in O(n^{\theta})$.

For any rational number $\frac{a}{b}$, with $b \neq 0$, we define the *height* of $\frac{a}{b}$, denoted as $\mathsf{height}(\frac{a}{b})$ as $\mathsf{height}(\frac{a}{b}) = \max(|a|,|b|)$. In particularly, we define $\mathsf{height}(0) = \mathsf{height}(0/1) = 1$. For a given matrix $A \in \mathbb{Q}^{m\times n}$, we define the height of A, denoted by $\mathsf{height}(A)$, as the maximal height of a coefficient in A.

In the following, we give complexity estimates in terms of $\mathcal{M}(k) \in O(k \log k \log\log k)$ and $\mathcal{B}(k) = \mathcal{M}(k)\log k \in O(k(\log k)^2 \log\log k)$.

We replace every term of the form $(\log k)^p (\log\log k)^q (\log\log\log k)^r$, (where p, q, r are positive real numbers) with $O(k^{\epsilon})$ where ϵ is a (positive) infinitesimal.

Furthermore, in the complexity estimates of algorithms operating on matrices and vectors over $\mathbb{Z}$, we use a parameter β, which is a bound on the magnitude of the integers occurring during the algorithm. Our complexity estimates are measures in terms of either *word operations* or *bit operations*.

We conclude this chapter with some complexity estimates for which we refer to [31].

Lemma 2.13 (Matrix matrix multiplication complexity [31]) *Let $A \in \mathbb{Z}^{m\times n}$ and $B \in \mathbb{Z}^{n\times p}$. Then, the product of A by B can be computed within $O(\mathcal{MM}(m,n,p)(\log\beta) + (mn + np + mp)\mathcal{B}(\log\beta))$ word operations, where $\beta = n\,\|A\|\,\|B\|$ and $\|A\|$ (resp. $\|B\|$) denotes the maximum absolute value of a coefficient in A (resp. B). Neglecting log factors, this complexity estimate becomes $O(\max(m,n,p)^{\theta}\max(h_A, h_b))$ where $h_A = \mathsf{height}(A)$ and $h_B = \mathsf{height}(B)$.*

Lemma 2.14 (Gaussian elimination complexity [31]) *For a matrix $A \in \mathbb{Z}^{m\times n}$, an estimate of Gauss-Jordan transform is $O(nmr^{\theta-2}(\log\beta) + nm(\log r)\mathcal{B}(\log\beta))$ word operations, where r is the rank of the input matrix A and $\beta = (\sqrt{r}\|A\|)^r$.*

Corollary 2.2 (Matrix rank and inverse computation complexity) *For a matrix $A \in \mathbb{Z}^{m\times n}$, with height h, an upper bound estimate of*

1. *computing the rank of A is done within* $O(mn^{\theta+\epsilon}h^{1+\epsilon})$ *word operations,*
2. *computing the inverse of A (when this matrix is invertible over* $\mathbb{Q}$ *and* $m = n$*) is done within* $O(m^{\theta+1+\epsilon}h^{1+\epsilon})$ *word operations.*

Lemma 2.15 (matrix inverse height bound) *Let* $A \in \mathbb{Z}^{n\times n}$ *be an integer matrix, which is invertible over* $\mathbb{Q}$*. Then, the absolute value of any coefficient in* A^{-1} *(inverse of A) can be bounded over by* $(\sqrt{n-1}||A||^{(n-1)})$.

Proof Because the matrix is over $\mathbb{Z}$, a lower bound of its determinant is 1. On the other hand, by Hadamard's inequality, we know that an upper bound for the determinant of any minors of A is $(\sqrt{n-1}||A||^{(n-1)})$. Therefore, with Cramer's rule, every entry of A^{-1} can be bounded over by $(\sqrt{n-1}||A||^{(n-1)})$.

Chapter 3

Minimal Representation for the Projection of a Polyhedron

As it was explained in the previous chapter, the algebraic complexity of the original FME algorithm is double exponential in the number of variables being eliminated The reason is the large number of redundant inequalities generated by this algorithm. In this chapter, we will develop a method which is capable of detecting these redundant inequalities yielding an algorithm which is single exponential in the number of variables (or polynomial in the number of inequalities when the dimension of the ambient space is regarded as constant).

With Section 3.1, we will begin by first explaining the *projection theorem* and then we will review three methods, *block elimination*, *Chernikov's algorithm* and *Kohler's algorithm*, based on this theorem. These three methods are able to detect some of the redundant inequalities and they can reduce the whole process of FME to single exponential running time. Although these methods are very effective in detecting and eliminating redundancies, their output is not the minimal representation of the projection of the polyhedron. Therefore, after reviewing them, we will discuss methods for computing a minimal representation of the projection of the polyhedron, that is, methods producing an output with no redundant inequalities.

The first method, see Section 3.2, is based on linear programming (LP) and this is a popular strategy implemented in various software, like the `PolyhedralSets` package in the computer algebra system Maple. Although this LP-based method can obtain a minimal representation of the projection of the polyhedron, both the theoretical efficiency and practical efficiency of that method can be outperformed by an other alternative approach. Then, in Section 3.3, we will discuss a corrected version of Balas' algorithm. This algorithm is not based on LP and it can detect all redundancies, but its complexity is double exponential in the number of input inequalities. Finally, in Section 3.4, we will introduce the algorithm that we have developed, based on the works of Balas and Kohler. This latter can generate a minimal representation and run in singly exponential time in the number of variables.

We will also analyze the complexity of all algorithms. Although the methods that we mention here are correct over the field $\mathbb{R}$ of real numbers, we shall assume, for simplicity of complexity analysis, that our input polyhedrons are over the field $\mathbb{Q}$ of rational numbers.

Throughout this chapter, we assume that Q is a polyhedron in $\mathbb{Q}^n$, defined by m inequalities and that we want to compute its standard projection on the $\mathbf{x}$-variables. Without loss of generality, we can write

$$Q = \{(\mathbf{u}, \mathbf{x}) \in \mathbb{Q}^p \times \mathbb{Q}^q \mid A\mathbf{u} + B\mathbf{x} \le \mathbf{c}\},$$

where $A \in \mathbb{Q}^{m\times p}$ and $B \in \mathbb{Q}^{m\times q}$ are matrices, $\mathbf{c} \in \mathbb{Q}^m$ is a vector and we have $p + q = n$. Then, the projection of the polyhedron, denoted by $\mathsf{proj}_{\mathbf{x}}(Q)$, is given by

$$\mathsf{proj}_{\mathbf{x}}(Q) = \{\mathbf{x} \in \mathbb{Q}^q \mid (\exists \mathbf{u} \in \mathbb{Q}^p)\ (\mathbf{u}, \mathbf{x}) \in Q\}.$$

3.1 Representations computable in single exponential time

We begin this section with a lemma known as the *projection lemma.*

Lemma 3.1 (Projection lemma [6]) *The projection $\mathsf{proj}_{\mathbf{x}}(Q)$ of the polyhedron Q can be represented by*

$$S = \{\mathbf{x} \in \mathbb{Q}^q \mid \mathbf{y}^t B\mathbf{x} \le \mathbf{y}^t\mathbf{c}\ (\forall \mathbf{y} \in \mathsf{extr(C)})\},$$

where $C = \{\mathbf{y} \in \mathbb{Q}^m \mid \mathbf{y}^t A = 0$ and $\mathbf{y} \ge \mathbf{0}\}$ and $\mathsf{extr(C)}$ is the set of extreme rays of C. The cone C defined in this lemma is called the projection cone.

Proof We should prove that $\mathsf{proj}_{\mathbf{x}}(Q) = S$ holds.

First, we show that $\mathsf{proj}_{\mathbf{x}}(Q) \subseteq S$. Let $\overline{\mathbf{x}} \in \mathsf{proj}_x(Q)$. Therefore, there exists $\overline{\mathbf{u}} \in \mathbb{Q}^p$ such that $(\overline{\mathbf{u}}, \overline{\mathbf{x}}) \in Q$. Thus $A\overline{\mathbf{u}} + B\overline{\mathbf{x}} \le \mathbf{c}$. Because the members of the cone C are all positive, we can multiply both sides of the inequalities in the representation of the polyhedron Q, by their transpose (it is equivalent to multiplying both sides of inequalities by a positive number). Therefore, for all $\mathbf{y} \in C$ (specifically extreme rays) we have $\mathbf{y}^t A\overline{\mathbf{u}} + \mathbf{y}^t B\overline{\mathbf{x}} \le \mathbf{y}^t\mathbf{c}$. Thus, $\mathbf{y}^t B\overline{\mathbf{x}} \le \mathbf{y}^t\mathbf{c}$, which implies that $\overline{\mathbf{x}} \in S$.

Then, we show $S \subseteq \mathsf{proj}_x(Q)$. We prove this by contradiction. Assume there exists $\overline{\mathbf{x}} \in S$ and $\overline{\mathbf{x}} \notin \mathsf{proj}_{\mathbf{x}}(Q)$. This means that there exists $\overline{\mathbf{x}}$ such that $\mathbf{v}^t B\overline{\mathbf{x}} \le \mathbf{v}^t\mathbf{c}$ for all $\mathbf{v} \in C$ and there is no $\mathbf{u} \in Q^p$ such that $A\mathbf{u} + B\overline{\mathbf{x}} \le \mathbf{c}$. Equivalently, there exists no $\mathbf{u}$ such that $A\mathbf{u} \le \mathbf{c} - B\overline{\mathbf{x}}$. By Farkas' lemma, there exists $\mathbf{w} \ge \mathbf{0}$ such that $\mathbf{w}^t A = \mathbf{0}$ and $\mathbf{w}^t(\mathbf{c} - B\overline{\mathbf{x}}) < \mathbf{0}$ both hold. Thus, we have $\mathbf{w}^t B\overline{\mathbf{x}} > \mathbf{w}^t\mathbf{c}$, which contradicts the assumption and the claim follows.

3.1.1 Block Elimination

In the *block elimination* algorithm below, see Algorithm 4, we first find the projection cone of the input polyhedron and then, find its extreme rays. By Lemma 3.1, we know that multiplying extreme rays by the representation matrix results in the desired projection of the polyhedron.

Lemma 3.2 (Block elimination algorithm correctness) *Algorithm 4 is correct.*

Proof This follows easily from the projection lemma, Lemma 3.1.

In order to analyze the complexity of block elimination, we need the following lemmas.

Lemma 3.3 (Height bound of extreme rays) *Let $S = \{A\mathbf{x} \le \mathbf{0}\}$ be a representation of a cone $C \in \mathbb{Q}^n$, where $A \in \mathbb{Q}^{m\times n}$. Then, the height of each extreme rays of the cone C is no more than $(n-1)^n \|A\|^{2(n-1)}$.*

Algorithm 4 Block Elimination

```
1: Input: $(S, \mathbf{u})$, where $S = \{A\mathbf{u} + B\mathbf{x} \le \mathbf{c}\}$ is the representation for input polyhedron $Q$ and
   $\mathbf{u}$ is set of variables to eliminate
2: Output: projection of $Q$ on $\mathbf{x}$, $\mathsf{proj}_{\mathbf{x}}(Q)$
3: $PS := \{ \}$
4: extract the projection cone representation from $S$ and call it $S_c$
5: $R := DDmethod(S_c)$
6: for column $\mathbf{r}_j$ in $R$ do
7:     $\ell_{new} := \mathbf{r}_j^t B$
8:     add $\ell_{new}$ to $PS$
9: end for
10: return $(PS)$
```

Proof By Lemma 2.7, we know that when $\mathbf{r}$ is an extreme ray, there exists a sub-matrix $A' \in \mathbb{Q}^{(n-1)\times n}$ of A, such that $A'\mathbf{r} = 0$. This means that r is in the null-space of A'. Thus, the claim follows by proposition 6.6 of [31].

Corollary 3.1 *Algorithm 2 requires $O(m^{(n+2)} n^{\theta+\epsilon} h^{1+\epsilon})$ bit operations, where h is the height of A in the input system.*

Proof Let $h_A = \mathsf{height}(A)$. To analyze the complexity of the DD method after adding t inequalities, with $n \le t \le m$, the first step is to partition the extreme rays in step $t-1$, with respect to the new inequality being considered. This step consists of n multiplication of numbers of size at most $(n-1)^n \|A\|^{2(n-1)}$ (Lemma 3.3), for $(t-1)^{\lfloor \frac{n}{2} \rfloor}$ vectors (Lemma 2.6). Hence, this step needs at most $C_1 := (t-1)^{\lfloor \frac{n}{2} \rfloor} \times n \times \mathcal{M}(\log((n-1)^n \|A\|^{2(n-1)})) \le O(t^{\lfloor \frac{n}{2} \rfloor} n^{2+\epsilon} h_A^{1+\epsilon})$ bit operations. After partitioning the extreme rays, the adjacency check is run. The cost of this step is equivalent to computing the rank of a sub-matrix $A' \in \mathbb{Q}^{(t-1)\times n}$ of A. This should be done for $\frac{t^n}{4}$ pairs of vectors. This step needs at most $C_2 := \frac{t^n}{4} \times O((t-1) n^{\theta+\epsilon} h_A^{1+\epsilon}) \le O(t^{n+1} n^{\theta+\epsilon} h_A^{1+\epsilon})$ bit operations. By Lemma 2.6, we know there are at most $t^{\lfloor \frac{n}{2} \rfloor}$ pairs of adjacent extreme rays. The next step is to combine every pair of adjacent vectors in order to obtain a new extreme ray. This step consists of n multiplication of numbers of size at most $(n-1)^n \|A\|^{2(n-1)}$ (Lemma 3.3) and it should be done for $t^{\lfloor \frac{n}{2} \rfloor}$ vectors. Therefore, the bit complexity of this step, is no more than $C_3 := t^{\lfloor \frac{n}{2} \rfloor} \times n \times \mathcal{M}(\log((n-1)^n \|A\|^{2(n-1)})) \le O(t^{\lfloor \frac{n}{2} \rfloor} n^{2+\epsilon} h_A^{1+\epsilon})$. Finally, the complexity of step t of the algorithm is $C := C_1 + C_2 + C_3$. The claim follows after simplification.

The complexity of the DD method highly depends on the number of extreme rays of the cones occurring in the intermediate steps. Those intermediate cones depend on the order in which the inequalities are selected in the algorithm. There are some methods for choosing the inequalities in order to decrease the number of extreme rays in the intermediate cones. In Corollary 3.1 we have considered the basic DD algorithm.

Lemma 3.4 (Complexity of block elimination) *Algorithm 4 needs at most $O(m^{\lfloor \frac{m}{2} \rfloor + \theta + \epsilon + 2} h_A)$ bit operations, where $h_A = \mathsf{height}(A)$.*

Proof The first step in block elimination is finding the extreme rays of the projection cone. For the polyhedron Q defined in the beginning of this chapter, the projection cone for eliminating p variables is a cone $C \in Q^m$ defined by p equations and m inequalities. We can use equations to substitute $p' \leq p$ variables and reduce the number of variables to $m - p'$, using Gaussian elimination. We neglect the complexity of this step. After this, the cone can be defined by m inequalities and $m - p'$ variables. Using Corollary 3.1, the complexity of the DD method for this is $O(m^{(m-p'+2)}(m-p')^{\theta+\epsilon}h_A^{1+\epsilon})$. Also, by Lemma 3.3, $||\mathbf{r}|| \leq (m-p'-1)^{m-p'}||A||^{2(m-p'-1)}$, where $\mathbf{r}$ is an extreme ray.

After finding extreme rays, we should compute the product of their transpose with the input representation matrix. By Lemma 2.6, the maximum number of extreme rays is $O(m^{\lfloor \frac{m-p'}{2} \rfloor})$. The complexity of this step can be shown to be $O((m^{\lfloor \frac{m-p'}{2} \rfloor})\mathcal{MM}(1, m, (n-p))) \leq m^{\lfloor \frac{m-p'}{2} \rfloor} m^{\theta} \log((m-p'-1)^{m-p'}||A||^{2(m-p'-1)})$ and the claim follows by simplifying this expression.

As it is shown in Lemma 3.4, finding the extreme rays of a projection cone is an exponential algorithm. Therefore, this method is not efficient in most cases. The reason is that usually the number of inequalities is much more than the dimension of the space (number of variables). We will explain how Chernikov and Kohler overcome the drawbacks of the block elimination algorithm by not computing extreme rays explicitly.

In other words, given the polyhedron $Q = \{(\mathbf{u}, \mathbf{x}) \in \mathbb{R}^p \times \mathbb{R}^q \mid A\mathbf{u} + B\mathbf{x} \leq \mathbf{c}\}$, Lemma 3.1 shows how to obtain a representation of the projection $\mathsf{proj}_{\mathbf{x}}(Q)$ of the polyhedron Q by enumerating the extreme rays of the projection cone $C = \{\mathbf{y} \in \mathbb{R}^m \mid \mathbf{y}A = \mathbf{0}, \mathbf{y} \geq \mathbf{0}\}$. To be more specific, we have $\mathsf{proj}_{\mathbf{x}}(Q) = \{\mathbf{x} \mid \forall \mathbf{y} \in \mathsf{ExtremeRays}(C), \mathbf{y}^t B\mathbf{x} \leq \mathbf{y}^t\mathbf{c}\}$. Instead of computing all the extreme rays of the cone C, Kohler's and Chernikov's methods take advantage of the notion of history set in order to check whether newly generated inequality belongs to the set $\{\mathbf{y}B\mathbf{x} \leq \mathbf{y}\mathbf{c} \mid \mathbf{y} \in \mathsf{ExtremeRays}(C)\}$ or not.

3.1.2 Chernikov's Method

Chernikov proposes a method in [6] that generates all inequalities in the FME process and then only keeps the ones that are extreme rays of the polyhedron's projection cone. The algorithm uses a combinatorial method to check whether the coefficient vector of an inequality is an extreme ray of the projection cone or not. At the end of each elimination, the algorithm goes thorough the history sets of inequalities and eliminates those ones which have a history set that is a super-set of another inequality's history set.

3.1.3 Kohler's method

In contrast with Chernikov's method that uses the combinatorial test, Kohler's algorithm uses the algebraic test for checking extreme rays of the projection cone. We found out that this method is more efficient in terms of running time and memory usage.

Lemma 3.5 (Kohler's theorem) *After eliminating p variables, the coefficient vector of an inequality ℓ is an extreme ray of the projection cone if and only if we have:*

$$\operatorname{rank}(A_{\eta(\ell)}) = ||\eta(\ell)|| - 1.$$

Algorithm 5 KohlerCheck

```
1: Input $(S_0, p, \ell, \eta(\ell))$, where $S_0$ is the representation of a polyhedron $Q$ and $S_0 = \{A\mathbf{u} + B\mathbf{x} \le \mathbf{c}\}$ for a matrix $A \in \mathbb{R}^{m\times p}$ and a matrix $B \in \mathbb{R}^{m\times q}$, $p$ is the number of variables that have been eliminated, $\ell$ is a newly generated inequality $\eta(\ell)$ is the history set of $\ell$
2: Output true if the coefficient vector of $\ell$ is an extreme ray of the projection cone, false otherwise
3: if the number of elements in $\eta(\ell)$ is greater than $p + 1$ then
4:     return false
5: end if
6: Let $A_{\eta(\ell)}$ be the sub-matrix of $A$ consisting of the rows with indices in $\eta(\ell)$;
7: if the number of elements in $\eta(\ell)$ is equal to $\text{rank}(A_{\eta(\ell)}) + 1$ then
8:     return true
9: else
10:    return false
11: end if
```

Corollary 3.2 *After eliminating p variables, an extreme ray can not have more than $p + 1$ elements in its historical set.*

Proposition 3.1 *Algorithm 5 is correct.*

Proof This proposition follows from Lemma 3.5 and Corollary 3.2.

Lemma 3.6 (Kohler check complexity) *Algorithm 5 can be performed within $O(p^{1+\theta+\epsilon}h^{1+\epsilon})$ bit operations.*

Proof From Lines 3 to 5 of Algorithm 5, we know $\eta(\ell)$ contains at most $p+1$ elements, which means that $A_{\eta(\ell)} \in \mathbb{Q}^{d_r\times p}$ holds for some non-negative integer $d_r \le p + 1$. Let $h = \mathsf{height}(A)$. By Corollary 2.2, computing the rank of $A_{\eta(\ell)}$ needs at most $O(pd_r^{\theta+\epsilon}h_A^{1+\epsilon}) \le O(p^{1+\theta+\epsilon}h^{1+\epsilon})$ bit operations.

Denote by S_p the set of inequalities representing the projection of the polyhedron after eliminating p variables. Denote by $S_p^{(K)}$ the set of inequalities in S_p which pass Algorithm 5, that is

$$S_p^{(K)} = \{\ell \in S_p \mid \text{KohlerCheck}(S_0, p, \ell, \eta(\ell)) = \text{true}\}.$$

Lemma 3.7 (Maximum number of inequalities and height bound) *There are at most $O(m^{p+1})$ inequalities in $S_p^{(K)}$. Moreover, the height for a coefficient of any inequality in $S_p^{(K)}$ can be bounded above by $(p + 1)h$, where h is the height of A in the input system S_0.*

Proof From Corollary 3.2, we know $\eta(\ell)$ contains at most $p + 1$ elements. That is, $S_p^{(K)}$ has at most $\binom{m}{1} + \cdots + \binom{m}{p+1} \le O(m^{p+1})$ inequalities. The second claim follows easily from Corollary 2 of [20].

With these explanations, we develop Algorithm 6, which uses Kohler's method to improve the FME algorithm and yields a single exponential representation of the projection of the polyhedron.

Algorithm 6 ImprovedFMEWithKohler

```
1: Input: $(S_0, \mathbf{u})$, where $S_0$ is the representation of $Q$ and $S_0 = \{A\mathbf{u} + B\mathbf{x} \leq \mathbf{c}\}$ for a matrix
   $A \in \mathbb{R}^{m\times p}$ and a matrix $B \in \mathbb{R}^{m\times q}$, all inequalities are assigned a history set and $\mathbf{u}$ is the
   vector of variables to be eliminated
2: Output: $S_p$, a single exponential representation of $\mathsf{proj}_{\mathbf{x}}(P)$;
3: for $k$ from 1 to $p$ do
4:   (Partition) Make $S^+_{k-1}$, $S^-_{k-1}$, $S^0_{k-1}$, subsets of $S_{k-1}$ consisting of inequalities with posi-
     tive, negative and zero coefficient w.r.t. $u_k$, respectively.
5:   for $\ell_p \in S^+$ do
6:     for $\ell_n \in S^-$ do
7:       $\eta := \eta(\ell_p) \cup \eta(\ell_n)$
8:       $\ell_{new} :=$ Combine$(\ell_p, \ell_n, u_k)$
9:       if KohlerCheck$(\mathsf{S_0}, \mathsf{k}, \ell_{\mathsf{new}}, \eta)$ is true then
10:        add $\ell$ to $S'_k$
11:        $\eta(\ell_{new}) := \eta$
12:      end if
13:    end for
14:  end for
15:  $S_k := S'_k \cup S^0_{k-1}$
16: end for
17: return$(S_p)$
```

Since block elimination, FME with Chernikov's method and FME with Kohler's method algorithms find extreme rays of the projection cone, their outputs are equivalent up to multiplication by a scalar. As we mentioned in the beginning of this section, block elimination, Chernikov's method and Kohler's method are effective in eliminating redundancies. The important point is that they only can remove *some* of the redundancies, not all of them, and their output is not the minimal representation. The reason for this is that, based on their structure, they only can detect redundancies due to the coefficient matrices of variables that are being eliminated and they do not consider redundancies due to the coefficient matrices of variables that remain.

The current method for obtaining a minimal representation, is based on linear programming (LP). We will explain it briefly in the next section.

3.2 Minimal representation via linear programming

The most common way for detecting redundant inequalities from a system is using Linear Programming methods (e.g. simplex [30]). This method is based on the following lemma [12].

Lemma 3.8 (Redundancy elimination with LP) *Let $A \in \mathbb{R}^{m \times n}$, $\mathbf{c} \in \mathbb{R}^m$ and $S = \{\mathbf{x} \in \mathbb{R}^n \mid A\mathbf{x} \leq \mathbf{c}\}$. An arbitrary inequality $\mathbf{p}^t\mathbf{x} \leq a$ is redundant in S if and only if the solution to the LP system S^* is less than or equal to a, where S^* is given by*

$$\begin{array}{rl} \textit{maximize:} & \mathbf{p}^t\mathbf{x} \\ \textit{subject to:} & A\mathbf{x} \leq \mathbf{c}, \\ & \mathbf{p}^t\mathbf{x} \leq a + 1 \end{array}$$

According to Lemma 3.8, we can detect all redundancies from a linear inequality system by considering each inequality (in turn) as an objective function and solve the corresponding LP problem. Although this method is proved to detect all redundant inequalities, it neither has a good theoretical complexity, nor is it effective in practice. The main reason for the low performance of this method is its dependence on LP solvers. While LP solvers are highly effective at solving even extremely large LP problems, making a large number of calls to a LP solver cannot have a negligible cost in the process of FME. Since, apart from redundancy testing, FME algorithms are essentially an adaptation of Gaussian elimination, it is desirable to achieve the redundancy via linear algebra instead of using LP. In the following sections, we will develop some methods that are able to detect all redundant inequalities and do not require an LP solver.

3.3 A revised version of Balas' algorithm

In this section, we will explain Balas' idea [1] for developing an algorithm which can detect all redundant inequalities from the projection of a pointed polyhedron. In contrast with currently used methods, this method does not need any LP solving and it only uses basic linear algebra operations. It should be noted that although we are using his idea, we have found some flaws in his paper. In this chapter, we will explain the corrected form.

We use $Q = \{(\mathbf{u}, \mathbf{x}) \in \mathbb{Q}^p \times \mathbb{Q}^q \mid A\mathbf{u} + B\mathbf{x} \leq \mathbf{c}\}$ as the input polyhedron. Here, we assume Q is pointed, that is, $\mathrm{rank}([A, B]) = p + q$.

Balas observed in [1] that if B is an invertible matrix, then, we can find a cone such that its extreme rays are in one-to-one correspondence with the facets of the projection of the polyhedron (the proof is similar to the proof which will come later Theorem 3.1). Using this fact, he developed an algorithm to find all redundant inequalities for all cases, including the cases where B is singular.

The basic idea is to find another polyhedron, which has a projection similar to the original one, and with an invertible matrix of coefficients for the $\mathbf{x}$ variables. To achieve this, we need to lift the polyhedron Q to a space with higher dimension as follows:

- Construct B_0:

  Assume that the first q rows of B, denoted as B_1, are independent. Denote the last $m - q$ rows of B as B_2. Add $m - q$ columns, $\mathbf{e}_{q+1}, \ldots, \mathbf{e}_m$, to B, where $\mathbf{e}_i$ is the i-th canonical basis in $\mathbb{Q}^m$ with 1 in the i-th position and 0's anywhere else. B_0 has the following form:

$$B_0 = \begin{bmatrix} B_1 & \mathbf{0} \\ B_2 & I_{m-q} \end{bmatrix}.$$

It also can be shown as:

$$\begin{array}{ccc} & \begin{array}{cc} \overbrace{\qquad\qquad\quad}^{q} & \overbrace{\qquad\quad}^{m-q} \end{array} & \\ m\left\{\vphantom{\begin{array}{c}a\\a\\a\\a\\a\end{array}}\right. & \left[\begin{array}{c:c} & 0 \\ B & \\ \hdashline & I \\ & \end{array}\right] & \left.\vphantom{\begin{array}{c}a\\a\\a\end{array}}\right\} m-q \end{array}$$

- To keep the consistency of the symbols, let $A_0 = A$, $\mathbf{c}_0 = \mathbf{c}$.
- Construct Q^0:

$$Q^0 = \{(\mathbf{u}, \mathbf{x}') \in \mathbb{Q}^p \times \mathbb{Q}^m \mid A_0\mathbf{u} + B_0\mathbf{x}' \leq \mathbf{c}_0 \ , \ x_{q+1} = \cdots = x_m = 0\}.$$

  Here and after, we use $\mathbf{x}'$ to represent the vector $\mathbf{x} \in \mathbb{Q}^q$, augmented with $m - q$ variables $(x_{q+1}, \ldots, x_m)$. Since the extra variables, $(x_{q+1}, \ldots, x_m)$, are zeros, $\mathsf{proj}_{\mathbf{x}}(Q)$ and $\mathsf{proj}_{\mathbf{x}'}(Q^0)$ are "isomorphic" with the bijection Φ:

$$\begin{aligned} \Phi : \mathsf{proj}_{\mathbf{x}}(Q) &\to \mathsf{proj}_{\mathbf{x}'}(Q^0) \\ (x_1, \ldots, x_q) &\mapsto (x_1, \ldots, x_q, 0, \ldots, 0) \end{aligned}$$

  In the following, we will treat $\mathsf{proj}_{\mathbf{x}}(Q)$ and $\mathsf{proj}_{\mathbf{x}'}(Q^0)$ as the same polyhedron when there is no ambiguity.

For simplicity, we denote the above process as $(A_0, B_0, \mathbf{c}_0) = \mathsf{ConstructQ^0(Q)}$.
Now, we use B_0 to construct the cone W^0 as

$$W^0 = \{(\mathbf{v}, \mathbf{w}, v_0) \in \mathbb{Q}^q \times \mathbb{Q}^{m-q} \times \mathbb{Q} \mid (\mathbf{v}, \mathbf{w})^t B_0^{-1} A_0 = 0, -(\mathbf{v}, \mathbf{w})^t B_0^{-1}\mathbf{c}_0 + v_0 \geq 0, (\mathbf{v}, \mathbf{w})^t B_0^{-1} \geq 0\}.$$

This construction of W^0 is slightly different from the one in Balas' work [1]: we change $-(\mathbf{v}, \mathbf{w})^t B_0^{-1}\mathbf{c}_0 + v_0 = 0$ to $-(\mathbf{v}, \mathbf{w})^t B_0^{-1}\mathbf{c}_0 + v_0 \geq 0$. Similar to the discussion in Balas' work, the extreme rays of the projected cone $\mathsf{proj}_{(\mathbf{v}, v_0)}(W^0)$ are used to construct the minimal representation of the projection of the polyhedron, $\mathsf{proj}_{\mathbf{x}}(Q)$. To prove this relation, we need some preparations first.

Lemma 3.9 *For the polyhedron Q, the operations "computing the characteristic cone" and "computing projections" commute. To be precise, we have:*

$$\mathsf{CharCone}(\mathsf{proj}_{\mathbf{x}}(Q)) = \mathsf{proj}_{\mathbf{x}}(\mathsf{CharCone}(Q)).$$

Proof By the definition of the characteristic cone, we have $\mathsf{CharCone}(Q) = \{(\mathbf{u}, \mathbf{x}) \mid A\mathbf{u} + B\mathbf{x} \le \mathbf{0}\}$, whose representation has the same left-hand side as the one of Q. The lemma is valid if we can show that the representation of $\mathsf{proj}_{\mathbf{x}}(\mathsf{CharCone}(Q))$ has the same left-hand side as $\mathsf{proj}_{\mathbf{x}}(Q)$. This is obvious with the Fourier-Motzkin elimination procedure.

Theorem 3.1 *The polar cone of* $\mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))$ *is equal to*

$$\mathsf{proj}'_{(\mathbf{v},v_0)}(W^0) := \{(\mathbf{v}, -v_0) \mid (\mathbf{v}, v_0) \in \mathsf{proj}_{(\mathbf{v},v_0)}(W^0)\}.$$

Proof By the definition of the polar cone in Definition 2.19,

$$(\mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q)))^* = \{(\mathbf{y}, y_0) \mid (\mathbf{y}, y_0)^t(\mathbf{x}, x_{\text{last}}) \le 0, \forall\, (\mathbf{x}, x_{\text{last}}) \in \mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))\}.$$

First, we show $\mathsf{proj}'_{(\mathbf{v},v_0)}(W^0) \subseteq (\mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q)))^*$. For any $(\overline{\mathbf{v}}, -\overline{v_0}) \in \mathsf{proj}'_{(\mathbf{v},v_0)}(W^0)$, we have $(\overline{\mathbf{v}}, \overline{v_0}) \in \mathsf{proj}_{(\mathbf{v},v_0)}(W^0)$. There exists $\overline{\mathbf{w}}$ such that

$$(\overline{\mathbf{v}}, \overline{\mathbf{w}})^t B_0^{-1} A_0 = 0, -(\overline{\mathbf{v}}, \overline{\mathbf{w}})^t B_0^{-1}\mathbf{c}_0 + \overline{v_0} \ge 0, (\overline{\mathbf{v}}, \overline{\mathbf{w}})^t B_0^{-1} \ge 0.$$

To show that $(\overline{\mathbf{v}}, -\overline{v_0}) \in \mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))^*$, we need to show that all vectors $(\mathbf{x}, x_{\text{last}}) \in \mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))$ satisfy $(\overline{\mathbf{v}}, -\overline{v_0})^t(\mathbf{x}, x_{\text{last}}) \le 0$. It is enough to prove the claim only for the extreme rays of $\mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))$. By Theorem 2.3 we know that extreme rays either have the form $(\mathbf{s}, 1)$ or $(\mathbf{s}, 0)$.

For the form $(\mathbf{s}, 1)$, we know that $\mathbf{s}$ is an extreme point of $\mathsf{proj}_x(Q)$, therefore, we have $\mathbf{s} \in \mathsf{proj}_{\mathbf{x}}(Q)$, that is, there exists $\overline{\mathbf{u}} \in \mathbb{Q}^p$, such that $A\overline{\mathbf{u}} + B\mathbf{s} \le \mathbf{c}$. By the construction of Q^0, we have $A_0\overline{\mathbf{u}} + B_0\mathbf{s}' \le \mathbf{c}_0$, where $\mathbf{s}' = [\mathbf{s}, s_{q+1}, \ldots, s_m]$ with $s_{q+1} = \cdots = s_m = 0$. Thus, by construction of W^0, we have $(\overline{\mathbf{v}}, \overline{\mathbf{w}})^t B_0^{-1} A_0\overline{\mathbf{u}} + (\overline{\mathbf{v}}, \overline{\mathbf{w}})^t B_0^{-1} B_0\mathbf{s}' \le (\overline{\mathbf{v}}, \overline{\mathbf{w}})^t B_0^{-1}\mathbf{c}_0$. Therefore, we have $\overline{\mathbf{v}}^t\mathbf{s} = (\overline{\mathbf{v}}, \overline{\mathbf{w}})^t\mathbf{s}' \le (\overline{\mathbf{v}}, \overline{\mathbf{w}})^t B_0^{-1}\mathbf{c}_0 \le \overline{v_0}$.

For the form $(\mathbf{s}, 0)$, we need to show $\overline{\mathbf{v}}\mathbf{s} \le 0$. Since $(\mathbf{s}, 0)$ is an extreme ray of $\mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))$, we have $\mathbf{s} \in \mathsf{CharCone}(\mathsf{proj}_{\mathbf{x}}(Q))$. By Lemma 3.9, there exists $\overline{\mathbf{u}} \in \mathbb{Q}^p$ such that $A_0\overline{\mathbf{u}} + B_0\mathbf{s}' \le \mathbf{0}$. Again by construction of W^0, we have $(\overline{\mathbf{v}}, \overline{\mathbf{w}})^t B_0^{-1} A_0\overline{\mathbf{u}} + (\overline{\mathbf{v}}, \overline{\mathbf{w}})^t B_0^{-1} B_0\mathbf{s}' \le (\overline{\mathbf{v}}, \overline{\mathbf{w}})^t B_0^{-1}\mathbf{0}$. Therefore, $\overline{\mathbf{v}}\mathbf{s} = (\overline{\mathbf{v}}, \overline{\mathbf{w}})\mathbf{s}' \le (\mathbf{v}, \mathbf{w}) B_0^{-1}\mathbf{0} = 0$. Obviously we have $(\overline{\mathbf{v}}, -\overline{v_0}) \in \mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))^*$.

Next, we show the reversed inclusion, that is, $(\mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))^* \subseteq \mathsf{proj}'_{(\mathbf{v},v_0)}(W^0)$. We need to show that for any $(\overline{\mathbf{y}}, \overline{y_0}) \in \mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))^*$, we have $(\overline{\mathbf{y}}, \overline{y_0}) \in \mathsf{proj}'_{(\mathbf{v},v_0)}(W^0)$, or equivalently, we have $(\overline{\mathbf{y}}, -\overline{y_0}) \in \mathsf{proj}_{(\mathbf{v},v_0)}(W^0)$. From $(\overline{\mathbf{y}}, \overline{y_0}) \in \mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))^*$, we conclude that for all $(\mathbf{x}, x_{\text{last}}) \in \mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))$, $(\overline{\mathbf{y}}, \overline{y_0})^t(\mathbf{x}, x_{\text{last}}) \le 0$ holds. Specifically, this is true for $(\mathbf{x}, 1) \in \mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))$. This implies that in this case, $\overline{\mathbf{y}}^t\mathbf{x} \le -\overline{y_0}$.

On the other hand, we know that in this case, $\mathbf{x} \in \mathsf{proj}_{\mathbf{x}}(Q)$. Therefore, $\overline{\mathbf{y}}^t\mathbf{x} \le -\overline{y_0}$, for all $\mathbf{x} \in \mathsf{proj}_{\mathbf{x}}(Q)$, which makes the inequality $\overline{\mathbf{y}}^t\mathbf{x} \le -\overline{y_0}$ redundant in the system $\{A\mathbf{u} + B\mathbf{x} \le \mathbf{c}\}$. By Farkas' Lemma (Lemma 2.3), there exists $\mathbf{p} \ge \mathbf{0}, \mathbf{p} \in \mathbb{Q}^m$ and $\lambda \ge 0$ such that $\mathbf{p}^t A = \mathbf{0}$, $\mathbf{y} = \mathbf{p}^t B$, [1]$y_0 = \mathbf{p} \cdot \mathbf{c} + \lambda$.

Remember that $A_0 = A$, $B_0 = [B, B']$, $\mathbf{c}_0 = \mathbf{c}$. Here $(B_0)_q$ is the first q columns of B_0. Let B' be the last $m - q$ columns of B_0 we augmented and let $\mathbf{w} = \mathbf{p}^t B'$. We will then have

$$\{\mathbf{p}^t A_0 = \mathbf{0},\ [\mathbf{y}, \mathbf{w}] = \mathbf{p}^t B_0, -y_0 \ge \mathbf{p}^t\mathbf{c}_0, \mathbf{p} \ge \mathbf{0}\},$$

[1]With the construction of W^0 in his paper [1], Balas tried to prove *"the inequality* $\mathbf{v}\mathbf{x} \le v_0$ *defines a facet of* $P_x(Q)$ *if and only if* $(\mathbf{v}, v_0)$ *is an extreme ray of the cone* $P_{\mathbf{v},v_0}(W^0)$*"*. While we found this is not true by experiments, we found the theoretical problem in his proof is he missed the non-negative number λ here.

which is equivalent to

$$\{\mathbf{p} = [\mathbf{y}, \mathbf{w}]^t B_0^{-1}, [\mathbf{y}, \mathbf{w}]^t B_0^{-1} A_0 = \mathbf{0}, -y_0 \geq [\mathbf{y}, \mathbf{w}]^t B_0^{-1} \mathbf{c}_0, [\mathbf{y}, \mathbf{w}]^t B_0^{-1} \geq \mathbf{0}\}.$$

Thus, $(\mathbf{y}, \mathbf{w}, -y_0) \in W^0$. Therefore, $(\mathbf{y}, -y_0) \in \mathsf{proj}_{\mathbf{v}, v_0}(W^0)$. From this, we deduce that $(\mathbf{y}, y_0) \in \mathsf{proj}'_{\mathbf{v}, v_0}(W^0)$ holds.

Corollary 3.3 *The following statements are equivalent:*

1. $(\mathbf{0}, 1)$ *is an extreme ray of cone* $\mathsf{proj}_{\mathbf{v}, v_0}(W^0)$*;*

2. $x_{\text{last}} \geq 0$ *is non-redundant in the representation of* $\mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))$.

Proof By Lemma 2.9, the extreme rays of the polar cone of $\mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))$ are in one-to-one correspondence with the facets, that is, the minimal representation, of $\mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))$. To be more specific, $\mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q)) = \{(\mathbf{y}, y_0)^t(\mathbf{x}, x_{\text{last}}) \leq 0\}$, where $(\mathbf{y}, y_0)$ runs through the extreme rays of $(\mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q)))^*$. Obviously, the ray $(\mathbf{0}, 1)$ is an extreme ray of cone $\mathsf{proj}_{\mathbf{v}, v_0}(W^0)$ if and only if the ray $(\mathbf{0}, -1)$ is an extreme ray of cone $\mathsf{proj}'_{\mathbf{v}, v_0}(W^0) = (\mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q)))^*$. Note that the ray $(\mathbf{0}, -1)$ is an extreme ray of the polar cone of $\mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))$ corresponds to the fact that $x_{\text{last}} \geq 0$ in the defining system of $\mathsf{hom}(\mathsf{proj}_{\mathbf{x}}(Q))$. The lemma is proved.

Remark 3.1 *Corollary 3.3 tells us that, by our construction of* W^0*, we may have at most one extra extreme ray* $(\mathbf{0}, 1)$ *of the cone* $\mathsf{proj}_{\mathbf{v}, v_0}(W^0)$ *which does not correspond to some facet of* $\mathsf{proj}_{\mathbf{x}}(Q)$*. However, with the construction of* W^0 *in Balas' paper, this extra extreme ray of* $\mathsf{proj}_{\mathbf{v}, v_0}(W^0)$ *can not be detected easily, which may lead to some redundant inequalities in the final output.*

Proposition 3.2 *The extreme ray of* $\mathsf{proj}_{\mathbf{v}, v_0}(W^0)$*, except* $(\mathbf{0}, 1)$*, defines exactly the facets of* $\mathsf{proj}_{\mathbf{x}}(Q)$*. To be more specific,*

$$\mathsf{proj}_{\mathbf{x}}(Q) = \{\mathbf{v}^t\mathbf{x} \leq v_0 \mid (\mathbf{v}, v_0) \textit{ runs through the extreme ray of } \mathsf{proj}_{\mathbf{v}, v_0}(W^0), \textit{ except } (\mathbf{0}, 1)\}.$$

Proof This proposition follows from Theorem 3.1, Corollary 3.3 and Theorem 2.3.

From Proposition 3.2, finding the extreme rays of $\mathsf{proj}_{\mathbf{v}, v_0}(W^0)$ is enough for finding a minimal representation of $\mathsf{proj}_{\mathbf{x}}(Q)$. This only requires the (not necessarily minimal) representation of $\mathsf{proj}_{\mathbf{v}, v_0}(W^0)$, which we will call as the *redundancy test cone* in what follows. Algorithm 7 shows how to find the redundancy test cone associated with a polyhedron, w.r.t. a variable set.

Algorithm 7 ConstructRedundancyTestCone

```
1: Input: $(S_0, \mathbf{u})$, where (i) $S_0$ is a representation of the polyhedron $Q$ and $S_0 = \{A\mathbf{u} + B\mathbf{x} \leq \mathbf{c}\}$ (ii) $\mathbf{u}$ is the list of variables to be eliminated
2: Output: a representation of the redundancy test cone $\mathsf{proj}_{\mathbf{v},v_0}(W^0)$
3: Construct $B_0$, $A_0$ and $\mathbf{c}_0$ as shown at the beginning of this subsection:
   $(A_0, B_0, \mathbf{c}_0) :=$ ConstructQ$^0(S)$
4: Let $\mathbf{v} := (v_1, \ldots, v_q)$, $\mathbf{w} := (w_1, \ldots, w_{m-q})$ and $v_0$ be a scalar variable
   $W^0 := \{(\mathbf{v}, \mathbf{w}, v_0) : (\mathbf{v}, \mathbf{w})B_0^{-1}A_0 = \mathbf{0},\ -(\mathbf{v}, \mathbf{w})B_0^{-1}\mathbf{c}_0 + v_0 \geq 0, (\mathbf{v}, \mathbf{w})B_0^{-1} \geq \mathbf{0}\}$
5: Solve the equation system $\{(\mathbf{v}, \mathbf{w})B_0^{-1}A_0 = \mathbf{0}\}$. W.l.o.g., let $\mathbf{w} = [\mathbf{w}', \mathbf{w}'']$, where $\mathbf{w}' \in \mathbb{Q}^{m-q-p'}$ and $\mathbf{w}'' \in \mathbb{Q}^{p'}$ (for some positive integer $p' \leq \min(p, m-q)$), are, respectively, the unsolved and solved variables
6: Substitute the solved variables $\mathbf{w}''$ into $W^0$, we obtain a new polyhedral cone $W_1^0$
7: Compute the projection cone $\mathsf{proj}_{\mathbf{v},v_0}(W_1^0)$, using block elimination
8: return the representation of $\mathsf{proj}_{\mathbf{v},v_0}(W^0)$
```

Lemma 3.10 (Complexity of constructing the redundancy test cone) *Algorithm 7 requires at most $O(m^{\lfloor \frac{p+q+1}{2} \rfloor + 2 + \epsilon}(p+q)^{\theta+\epsilon}q^{2+\epsilon}h^{1+\epsilon})$ bit operations, where h is the maximal height of A, B and $\mathbf{c}$ in the input system. Moreover, $\mathsf{proj}_{\mathbf{v},v_0}(W^0)$ can be represented by at most $O((p+q+1)^{\lfloor \frac{p+q+1}{2} \rfloor})$ inequalities, each with height bound no more than $O((p+q)^{1+\epsilon}(m^{\epsilon}q^{1+\epsilon} + q^{2+\epsilon}h))$.*

Proof Denote by h the maximal height of A, B and $\mathbf{c}$ in the input system. We analyze the complexity step by step.

Step 1 The complexity for this step can be neglected. However, we should notice the special structure of B_0: $\begin{pmatrix} B_1 & \mathbf{0} \\ B_2 & I_{m-q} \end{pmatrix}$, with $B_1 \in \mathbb{Q}^{q \times q}$ full rank and $B_2 \in \mathbb{Q}^{(m-q) \times q}$.

Step 2 Note that $B_0^{-1} = \begin{pmatrix} B_1^{-1} & \mathbf{0} \\ -B_2B_1^{-1} & I_{m-q} \end{pmatrix}$ and $\|B_1^{-1}\| \leq (\sqrt{q-1}\|B_1\|)^{q-1}$ by Lemma 2.15. Therefore,
$\|B_0^{-1}\| \leq q^{\frac{q+1}{2}}\|B\|^q$, $\quad \|B_0^{-1}A_0\| \leq q^{\frac{q+3}{2}}\|B\|^q\|A\| + (m-q)\|A\|$,
$\|B_0^{-1}\mathbf{c}_0\| \leq q^{\frac{q+3}{2}}\|B\|^q\|\mathbf{c}_0\| + (m-q)\|\mathbf{c}_0\|$.
That is, $\mathsf{height}(B_0^{-1}) \leq O(q^{1+\epsilon}h)$, $\mathsf{height}(B_0^{-1}A_0), \mathsf{height}(B_0^{-1}\mathbf{c}_0) \leq O(m^{\epsilon} + q^{1+\epsilon}h)$.
To give the complexity for this step, we need the following consecutive steps:

- Computing B_1^{-1} requires $O(q^{\theta+1+\epsilon}h^{1+\epsilon})$ bit operations;
- Computing B_0^{-1} requires
$$\begin{aligned} &O(q^{\theta+1+\epsilon}h^{1+\epsilon}) + O((m-q)q^2\mathcal{M}(\max(\mathsf{height}(B_2), \mathsf{height}(B_1^{-1})))) \\ \leq &O(mq^{\theta+1+\epsilon}h^{1+\epsilon}) \text{ bit operations;} \end{aligned}$$
- Constructing W^0 requires at most
$$\begin{aligned} C_1 := &O(m^{1+\epsilon}q^{\theta+1+\epsilon}h^{1+\epsilon}) + O((m-q)qp\mathcal{M}(\max(\mathsf{height}(A_0), \mathsf{height}(B_0^{-1}), \mathsf{height}(\mathbf{c}_0))) \\ &+ O((m-q)h) \leq O(m^{1+\epsilon}q^{\theta+\epsilon+1}ph^{1+\epsilon}) \text{ bit operations.} \end{aligned}$$

Step 3 The solutions to the equation system $(\mathbf{v}, \mathbf{w})B_0^{-1}A_0 = \mathbf{0}$ can be obtained by computing the Gaussian elimination of $B_0^{-1}A_0$, which has rank at most p. Thus, the bit complexity for this step is at most $C_2 := O(m^{1+\epsilon}p^{\theta+\epsilon}q^{1+\epsilon}h)$

Moreover, the solved variables $\mathbf{w}''$ can be expressed as a linear combination of $(\mathbf{w}', \mathbf{v})$, denoted as $\mathbf{w}'' = U_1\mathbf{w}' + U_2\mathbf{v}$ for some $U_1 \in \mathbb{Q}^{p' \times (m-q-p')}$ and $U_2 \in \mathbb{Q}^{p' \times q}$, where p' is the number of variables in $\mathbf{w}''$. The absolute value of a coefficient in U_1 and U_2 can be bounded above by $\max(\|U_1\|, \|U_2\|) \leq q(\sqrt{q-1}\|B_0^{-1}A_0\|)^{q-1}\|B_0^{-1}A_0\| \leq q^{\frac{q+1}{2}}\|B_0^{-1}A_0\|^q$. That is, $\mathsf{height}([U_1, U_2]) \leq O(m^\epsilon p + q^{1+\epsilon}ph)$.

Step 4 Substituting $\mathbf{w}''$ into the following inequality system, we obtain the cone W_1, which has form

$$W_1 = \{(\mathbf{w}', \mathbf{v}, v_0) \mid (\mathbf{v}, \mathbf{w}')\begin{pmatrix} \overline{B}_1 \\ \overline{B}_2 \end{pmatrix} + v_0 \geq 0, (\mathbf{v}, \mathbf{w}')\begin{pmatrix} \overline{B}_3 & \mathbf{0} \\ \overline{B}_4 & I_{q'} \end{pmatrix} \geq \mathbf{0}\},$$

where $q' = q + p', \overline{B}_1 \in \mathbb{Q}^{q \times 1}, \overline{B}_2 \in \mathbb{Q}^{q' \times 1}, \overline{B}_3 \in \mathbb{Q}^{q \times (m-q')}, \overline{B}_4 \in \mathbb{Q}^{q' \times (m-q')}$. This requires

$C_3 := O((m-p')q^2\mathcal{M}(\max(\mathsf{height}(U_1), \mathsf{height}(U_2), \mathsf{height}(B_0^{-1}\mathbf{c}_0), \mathsf{height}(B_0^{-1}))))$

$\leq O(m^{1+\epsilon}p^2q^{2+\epsilon}h^{1+\epsilon})$ bit operations.

Moreover, the absolute value of a coefficient in $\overline{B_i}, i = 1, \ldots, 4$ can be bounded by $q\max(\|B_0^{-1}\mathbf{c}_0\|, \|B_0^{-1}\|)\max(\|U_1\|, \|U_2\|)$, from which we deduce that: $\mathsf{height}(\overline{B_1}, \overline{B_2}, \overline{B_3}, \overline{B_4}) \leq O(m^\epsilon q + q^{2+\epsilon}h)$.

Step 5 We follow Lemma 3.1 to obtain the representation of $P_{\mathbf{v},v_0}(W_1^0)$, that is, we need to find the extreme rays of the projection cone

$$C = \{\mathbf{y} \in \mathbb{Q}^{m+1} \mid \mathbf{y}\begin{pmatrix} I_{q'} \\ (\overline{B}_4)^t \\ (\overline{B}_2)^t \end{pmatrix} = \mathbf{0}, \mathbf{y} \geq \mathbf{0}\}.$$

Note that $y_1, \ldots, y_{m-q'}$ can be solved by the system of equations in the representation of C. Therefore, finding the extreme rays of the cone C is equivalent to computing the extreme rays of

$$C' = \{\mathbf{y}' \in \mathbb{Q}^{q'+1} \mid \mathbf{y}'\overline{B}_5 \leq \mathbf{0}, \mathbf{y}' \geq \mathbf{0}\}, \text{ where } \overline{B}_5 = \begin{pmatrix} (\overline{B}_4)^t \\ (\overline{B}_2)^t \end{pmatrix} \in \mathbb{Q}^{(q'+1) \times m-q'}.$$

Applying Algorithm 2 to C', we can obtain all the extreme rays of C', and subsequently, extreme rays of C. This operation requires at most $C_4 := O(m^{p'+q}(p+q)^{\theta+\epsilon}\mathsf{height}(\overline{B_5})^{1+\epsilon} \leq O(m^{p+q+2+\epsilon}(p+q)^{\theta+\epsilon}q^{2+\epsilon}h^{1+\epsilon})$ bit operations.

Thus, the total bit complexity for Algorithm 7 will be

$$C_1 + C_2 + C_3 + C_4 \leq O(m^{p+q+2+\epsilon}(p+q)^{\theta+\epsilon}q^{2+\epsilon}h^{1+\epsilon}).$$

Moreover, by Lemma 3.3 and Lemma 2.6, cone C has at most $O(m^{\lfloor\frac{p+q}{2}\rfloor})$ distinct extreme rays, each with height no more than $O((m)^{1+\epsilon}(m^\epsilon q^{1+\epsilon} + q^{2+\epsilon}h))$. That is, $\mathsf{proj}_{\mathbf{v},v_0}(W^0)$ can be represented by at most $O(m^{\lfloor\frac{p+q+1}{2}\rfloor})$ inequalities, each with height bound no more than $O(m^{1+\epsilon}q^{2+\epsilon}h)$.

From Lemma 3.10 we know that the redundancy test cone has exponential number of facets in the worst case and finding its extreme rays is double exponential. In order to solve this problem, instead of finding extreme rays, we generate all inequalities and then use the algebraic test for extremes to check whether the coefficient vector of that inequality is an extreme ray of the redundancy test cone or not. Algorithm 8 shows this algorithm.

Lemma 3.11 *Algorithm 8 can be performed* $O(m^{\frac{p+q}{2}}(p+q)^{\theta+\epsilon}h^{1+\epsilon})$ *bit operations.*

Algorithm 8 RedundancyTest

```
1: Input: $(\mathsf{proj}_{\mathbf{v},v_0}(W^0), \ell)$, where (i) $\mathsf{proj}_{\mathbf{v},v_0}(W^0) = \{(\mathbf{v}, v_0) \mid M(\mathbf{v}, v_0) \leq \mathbf{0}\}$ is the redundancy
   test cone, (ii) $\ell : \mathbf{tx} \leq t_0$ is an inequality
2: Output: true if $\ell$ is not redundant, false otherwise
3: Let $M$ be the coefficient matrix of $\mathsf{proj}_{\mathbf{v},v_0}(W^0)$
4: Let $\mathbf{s} := M(\mathbf{t}, t_0)$
5: Let $\zeta(\mathbf{s})$ be the index set of zero elements
6: if rank$(M_{\zeta(\mathbf{s})}) = n - 1$ then
7:     return true
8: else
9:     return false
10: end if
```

Proof The first step is to multiply the matrix M and the vector $(\mathbf{t}, t_0)$. Let d_M and c_M be the number of rows and columns of M, respectively, thus $M \in \mathbb{Q}^{d_M \times c_M}$. We know that M is the coefficient matrix of $\mathsf{proj}_{(\mathbf{v},\mathbf{v}_0)}(W^0)$. Therefore, $c_M = q + 1$ and $d_M \leq m^{\frac{p+q}{2}}$. Also, $\mathsf{height}(M) \leq O(m^{1+\epsilon}q^{2+\epsilon}h)$. With these specifications, the multiplication step and the rank computation step need $O(m^{\frac{p+q}{2}}(p+q)^{2+\epsilon}h^{1+\epsilon})$ and $O(m^{\frac{p+q}{2}}(q+1)^{\theta+\epsilon}h^{1+\epsilon})$ bit operations, respectively, and the claim follows after simplification.

Using Algorithm 7 and Algorithm 8, we can find the minimal representation of projection of the polyhedron in singly exponential time.

3.4 Efficient algorithm for minimal representation

In this section, we present our algorithm for removing all the redundant inequalities computed during the process of Fourier-Motzkin elimination, where our goal is to eliminate all variables one-by-one and get an equivalent representation of the input inequality system. For convenience, we rewrite the input polyhedron as

$$Q = \{\mathbf{y} \in \mathbb{Q}^n \mid \mathbf{Ay} \leq \mathbf{c}\}, \tag{3.1}$$

where $\mathbf{A} = [A, B] \in \mathbb{Q}^{m \times n}$, $n = p + q$ and $\mathbf{y} = [\mathbf{u}^t, \mathbf{x}^t]^t \in \mathbb{Q}^n$. Combining Kohler's method (Algorithm 5) and the revised and improved Balas' method (as explained in the previous section, using Algorithms 7 and 8), Algorithm 9 produces the minimal representation of $\mathsf{proj}_{\mathbf{x}}(Q)$ from the representation of $\mathsf{proj}_{\mathbf{u},\mathbf{x}}(Q)$, by eliminating variable $\mathbf{u}$ from the system. It should be noted that although we can obtain the minimal representation by using Algorithms 7 and 8 only, we choose to use Algorithm 5 as well in order to make our proposed algorithm more efficient.

For the polyhedron Q, given a variable order $y_1 > \cdots > y_n$, we want to obtain its projections onto the coordinate space $\mathbf{y}_i$ for any $i : 1 \leq i \leq n$, where $\mathbf{y}_i = [y_i, \ldots, y_n]$. To achieve this, we use *the projected representation* as defined in [19].

Definition 3.1 (Projected representation) *Given the polyhedron Q represented above, and the variable order $y_1 > \cdots > y_n$, we denote by $Q^{(y_1)}$ the inequalities in the representation of Q*

Algorithm 9 One-level Fourier-Motzkin elimination ($\mathrm{FM}_{\mathrm{inner}}$)

```
1: Input: $(S_0, u, \mathbf{x}, S, \eta)$, where (i) $S_0 = \{\mathbf{Ay} \le \mathbf{c}\}$ is the representation of the input polyhedron, (ii) $u$ is the variable to be eliminated, (iii) $\mathbf{x}$ is the coordinates for the space that we want to project on, (iv) $S$ is the representation of $\mathsf{proj}_{u,\mathbf{x}}(Q)$, (v) $\eta = \{\eta(\ell) : \text{for all } \ell \in \mathbf{S}\}$ is the set of history sets corresponding to each inequality in $S$.
2: Output: $(S^{\neq 0}, S', \eta')$, where (i) $S^{\neq 0}$ is the subset of $S$ consisting of inequalities with nonzero coefficient w.r.t. $u$; (ii) $S'$ is the minimal representation of $\mathsf{proj}_{\mathbf{x}}(Q)$; (iii) $\eta'$ is the set of history sets corresponding to all inequalities in $S'$.
3: Let $S^+$, $S^-$ and $S^0$ be the subsets of $S$ consisting of inequalities with positive, negative and zero coefficients w.r.t. $u$ respectively
4: $S^{\neq 0} := S^+ \cup S^-$
5: $S' := S^0$
6: Let $\eta'$ be the set of history sets of the inequalities in $S^0$,
7: if $S^+$ or $S^-$ is empty then
8:     return $(S^{\neq 0}, S', \eta')$,
9: end if
10: Let $\mathbf{u} := \mathbf{y} \setminus \mathbf{x}$
11: $\mathsf{proj}_{\mathbf{v},v_0}(W^0)$ := ConstructRedundancyTestCone($S_0, \mathbf{u}$)
12: for $\ell_p$ in $S^+$ do
13:     for $\ell_n$ in $S^-$ do
14:         $\ell_{\text{new}} := -\text{coeff}(\ell_n, u) \times \ell_p + \text{coeff}(\ell_p, u) \times \ell_n$
15:         $\eta(\ell_{\text{new}}) := \eta(\ell_p) \cup \eta(\ell_n)$
16:         if KohlerCheck($S_0, |\mathbf{y}| - |\mathbf{x}|, \ell_{\text{new}}, \eta(\ell_{\text{new}})$) = true then
17:             if RedundancyTest($\mathsf{proj}_{\mathbf{v},v_0}(W^0), \ell_{\text{new}}$) = true then
18:                 Add $\ell_{new}$ to $S'$
19:                 Add $\eta(\ell_{\text{new}})$ to $\eta'$
20:             end if
21:         end if
22:     end for
23: end for
24: return $(S^{\neq 0}, S', \eta')$
```

whose largest variable is y_1. We call this the projected representation *of Q w.r.t. the variable order $y_1 > \cdots > y_n$ and denote by* $\mathsf{ProjRep}(Q; y_1 > \cdots > y_n)$ *the linear system given by $Q^{(y_1)}$ if $n = 1$ and by the conjunction of $Q^{(y_1)}$ and* $\mathsf{ProjRep}(\mathsf{proj}_{\mathbf{y}_2}(Q); y_2 > \cdots > y_n)$ *otherwise. To be more specific,* $\mathsf{ProjRep}(Q; y_1 > \cdots > y_n) := \bigcup_{1\le i\le n} \mathsf{proj}_{\mathbf{y}_i}(Q)^{(y_i)}$, *where* $\mathsf{proj}_{\mathbf{y}_1}(Q) = Q$ *and* $\mathsf{proj}_{\mathbf{y}_i}(Q)^{(y_i)}$ *consists of the set of inequalities in the representation of* $\mathsf{proj}_{\mathbf{y}_i}(Q)$ *whose largest variable is y_i for $i : 1 \le i \le n$. We call* $\bigcup_{1\le i\le n} \mathsf{proj}_{\mathbf{y}_i}(Q)^{(y_i)}$ *a* minimal projected representation *if all the inequalities in* $\mathsf{proj}_{\mathbf{y}_k}(Q)^{(y_k)}$ *are not redundant w.r.t.* $\bigcup_{k\le i\le n} \mathsf{proj}_{\mathbf{y}_i}(Q)^{(y_i)}$ *for $k : 1 \le k \le n$.*

Algorithm 10 Fourier-Motzkin elimination with removing all the redundant inequalities

```
1: Input: $(S_0, y_1 > \cdots > y_n)$ where (i) $S_0 = \{\mathbf{Ay} \le \mathbf{c}\}$ is a representation of input polyhedron,
   with $m$ inequalities and $n$ variables, (ii) $y_1 > \cdots > y_n$ is the variable order
2: Output: $\mathsf{ProjRep}(Q; y_1 > \cdots > y_n)$: the minimal projected representation of $Q$
3: Let $S := S_0$, $\mathbf{x} = \mathbf{y}$, $S_{\text{out}} := \{\ \}$
4: for $i$ from 1 to $m$ do
5:     $\eta(\ell_i) := \{i\}$, where $\ell_i$ is the $i$-th inequality in $S_0$
6: end for
7: $\eta := \{\eta(\ell_i) \mid 1 \le i \le m\}$;
8: for $i$ from 1 to $n$ do
9:     $\mathbf{x} := \mathbf{x} \setminus \{y_i\}$, $u := y_i$
10:    $(S^{\neq 0}, S, \eta) := \mathrm{FM}_{\text{inner}}(S_0, u, \mathbf{x}, S, \eta)$
11:    $S_{\text{out}} := S_{\text{out}} \cup S^{\neq 0}$
12: end for
13: return $S_{\text{out}}$
```

Proposition 3.3 *Algorithm 10 is correct and terminates.*

Proof We eliminate the variables one by one, using Fourier-Motzkin elimination. Since FME algorithm terminates and we only have finitely many variables, termination of Algorithm 10 is obvious.

Clearly, the output of Algorithm 10 has the form of the projected representation. Minimality, and hence the correctness, are guaranteed by Propositions 3.1 and 3.1.

Lemma 3.12 *Algorithm 10 can be performed within $O(m^{n+2+\epsilon}n^{\theta+\epsilon}h^{1+\epsilon})$ bit operations.*

Proof By Lemma 3.7, when p variables are eliminated, we will have $(\frac{m^{p+1}}{2})^2$ pairs of inequalities in S_p^+ and S^-. At most m^{p+1} of them pass the first condition of Kohler check and we need to compute ranks only for them. For each newly generated inequality, we need to apply the KohlerCheck (Algorithm 5) and RedundancyCheck(Algorithm 8) while we only need to compute the redundancy test cone once for each loop. By Lemmas 3.6, 3.10 and 3.11, the total complexity for Algorithm 10 will be

$$\sum_{p=0}^{n-1}(m^{p+1}O(p^{1+\theta+\epsilon}h^{1+\epsilon}) + O(m^{n+2+\epsilon}n^{\theta+\epsilon}(n-p)^{2+\epsilon}h^{1+\epsilon})$$
$$+ m^p(O(m^{\lfloor\frac{p+q+1}{2}\rfloor+1+\epsilon}q^{\theta+1+\epsilon}h^{1+\epsilon})) \le m^{n+2+\epsilon}n^{\theta+\epsilon}h^{1+\epsilon}$$

Chapter 4

Implementation and Experimentation

This chapter focuses on our implementation in the C language. We will begin by describing the libraries we have used, then we will explain the details of our implementation and, finally, conclude by examining our experimental results.

4.1 Supporting libraries

For the implementation of this algorithm, we have used GMP [14] library for arithmetic operations, FLINT [16] for matrix computations and CDD [11] for polyhedron functions. In this section, we will describe these libraries briefly.

4.1.1 GMP library

GNU Multi Precision (GMP for short) library, is a portable library written in the C language. The main goal of this library is to provide fast arbitrary precision arithmetic on integers, rationals and floating point numbers. Based on the official manual [14], it provides the fastest possible arithmetic for all applications that need higher precision than can be supported by the basic C language.

The emphasis of GMP library is speed. The speed is achieved by using sophisticated algorithms with highly optimized implementations, using assembly code for most of the parts.

We are using the GMP rational number type, mpq_t, and related arithmetic for storing coefficients of inequalities and for their arithmetic operations. The combine function significantly depends on GMP for its implementation. In this function numbers grow quickly and go beyond the the supported precision, as is described in Lemma 3.7.

4.1.2 FLINT library

The Fast LIbrary for Number Theory (FLINT for short), is a C functions library for doing number theory operations. Not only it is highly optimized, it also provides support for multi-core processors[17]. Also, FLINT supports the GMP library for multi precision arithmetic.

There are also linear algebra functions implemented in FLINT. These functions are optimized by using number theory algorithms and modular arithmetic.

As it was described in previous chapters, we need matrix operations in some functions. We rely on FLINT for these operations. To be specific, we have used FLINT for the function KohlerCheck in Algorithm 6, the function redundancyCheck for matrix-vector multiplication and matrix rank computation, and for the function constructRedundancyTestCone for matrix-matrix multiplication and to compute matrix inverses.

4.1.3 CDD library

The CDD library is an efficient implementation of the double description method in the C language. It can efficiently transforms of polyhedrons in the H-representation to the V-representation. It also has a GMPRATIONAL mode for using GMP arithmetic.

We need to find a projection cone's extreme rays in order to find redundancy test cone efficiently. For this function, we rely on CDD library. Specifically, we have used CDD library to find the extreme rays to project the cone in constructRedundancyTestCone.

4.2 Basic data-structure

The most basic data structure that we use for our algorithm is the inequality data structure. We assume that coefficients and constants of inequalities are over $\mathbb{Q}$ and we use the mpq_t type in GMP library for storing them.

Therefore, the inequality data structure consists of an mpq_t array for saving inequality's coefficients and an mpq_t element for saving the constant value. They are called coeff and constant in the program, respectively. Also, there is an integer array to keep the historical set of the inequality and it calls the history in the program. The size of coeff array is equal to the dimension of the space and the size of history array is equal to the number of inequalities in the representation of the input polyhedron. Listing 4.1 and Listing 4.2 show the C program for defining this structure and the function to initialize it.

```c
typedef struct {
  mpq_t constant;
  mpq_t* coef;
  int* history;
} inequality;
```

Listing 4.1: Inequality data structure

```c
void makeNewInequality(inequality* newIneq, int varNum, int ineqNum)
{
  mpq_init(newIneq->constant);

  newIneq->coef = (mpq_t *) malloc(varNum*sizeof(mpq_t));
  for (int i = 0; i < varNum; i++)
    mpq_init(newIneq->coef[i]);

  newIneq->history = (int *) malloc(ineqNum*sizeof(int));
  for (int i = 0; i < ineqNum; i++)
    newIneq->history[i] = 0;
}
```

Listing 4.2: Make new inequality function

4.3 Finding extreme rays

As was explained in previous chapter, in order to find the redundancy test cone efficiently, we need to compute the extreme rays of its lifted cone. Also, we explained that we have used the CDD library for finding extreme rays. Listing 4.3 shows the C function for this purpose. It gets the representation of a cone in form of a rational array and computes its extreme rays, using CDD library.

```c
#include "setoper.h"
#include "cdd.h"
#include "cddmp.h"
#include "linAlg.h"
#include <gmp.h>

mpq_t* extr(mpq_t * in, int row, int col, int * Grow, int * Gcol)
{
  //CDD structures initialization
  dd_MatrixPtr A, G;
  dd_PolyhedraPtr poly;
  dd_ErrorType err;

  dd_set_global_constants();
  A = dd_CreateMatrix(row, col);

  //initialize CDD input matrix
  for(int i = 0; i < col; i++)
    dd_set_d(A->matrix[i][i], 1);

  int c = 0;
  for(int i = col; i < row; i++)
    for(int j = 0; j < col; j++)
        dd_set(A->matrix[i][j], in[c++]);
  A->representation = dd_Inequality;

  //find extreme rays
  poly = dd_DDMatrix2Poly(A, &err); //<<<---- Main function
  G = dd_CopyGenerators(poly);

  int grow = G->rowsize;
  int gcol = G->colsize;

  //make up mpq_t output
  mpq_t * out = (mpq_t *) malloc(grow * grow * sizeof(mpq_t));

  for(int i = 0; i < grow * grow; i++)
    mpq_init(out[i]);

  c = 0;
  for(int i = 0; i <  grow ; i++)
    for(int j = 0; j < gcol; j++)
      mpq_set(out[c++], G->matrix[i][j]);
  return(out);
}
```

Listing 4.3: Find extreme rays of a cone

4.4 Implementation details

We went through two rounds of implementation using different ways of implementing systems of inequalities:

1. the first one uses a dense, two dimensional array where each row encodes an inequality, we call it the *array representation*,
2. the second one uses linked lists where each node stores a small two dimensional array and each row encodes an inequality, we call it the *unrolled linked list representation*.

In this section, we will explain these two approaches in detail.

4.4.1 The array representation

In the array representation, we store an inequality in each element of a one dimensional array. Each inequality consists of an array of coefficients and an array to store its historical set. Therefore, this representation can be seen as storing the coefficient matrix of the inequality system in a two dimensional array.

Example 4.1 *Consider the inequality system:* $\{x + 2y + 5z \leq 2, x + z \leq 0, \ \ y \leq 5$. *In the array representation, these inequalities can be represented as in Figure 4.1. In this figure, the first three columns in each row shows the inequalities, the fourth is the constant term and last three one are storing the historical set.*

1	2	5	2	1	0	0
1	0	1	0	0	1	0
0	0	1	5	0	0	1

Figure 4.1: Example of the array representation

In this representation, to eliminate each variable we have the following steps:

1. partition input inequality system with respect to the variable, using the `partition` function,
2. for all inequalities in the positive array and all in the negative array, combine them using the `combine` function,
3. check whether the newly generated inequality is redundant or not, using the `kohlerCheck` function,
4. if it passes Kohler's check, check it with the `balasCheck` function,
5. if it passes Balas's check, add it to an array,
6. after considering all inequalities in the positive array with all in the negative array, form the union with the zero set and go to the next variable.

Listing 4.4 shows the implementation of this process. Detailed implementation of functions KohlerCheck and balasCheck can be found in Chapter A.

```c
#include "inequality.h"
#include "kohler.h"

inequality* FMEOneStepNoRedundant(inequality* data, inequality* elimSet, int varNum
    , int ineqNum, int newIneqNum, int elimNum, int* finalSize,
    mpq_t* pw0, int sizepw0)
{
  int size = newIneqNum;

  inequality * newPos = (inequality *) malloc(sizeof(inequality));
  inequality * newIneq = (inequality *) malloc(sizeof(inequality));
  inequality * newIneqTmp = (inequality *) malloc(sizeof(inequality));

  int e = elimNum - 1;

  int * positives = (int *) malloc(size * sizeof(int));
  inequality * negatives = (inequality *) malloc(size * sizeof(inequality));

  //allocate memory for the worst case
  inequality * news = (inequality *) malloc((size / 2) * (size / 2) * sizeof(
    inequality));

  int sp = 0;
  int sn = 0;
  int sz = 0;

  partition(elimSet, positives, negatives, news, &sp, &sn, &sz, e, varNum, ineqNum,
     size);

  if (sp != 0 && sn != 0)
    for (int i = 0; i < sp; i++) //consider positive set
    {
      copyIneq(elimSet[positives[i]], newPos, varNum, ineqNum);
      for (int j = 0; j < sn; j++) //consider negative set
      {
        combine(*newPos, negatives[j], newIneq, e, varNum, ineqNum);
        if (kohlerCheck(*newIneq, data, elimNum, e, varNum, ineqNum))
          if (balasCheck(*newIneq, pw0, varNum, ineqNum, sizepw0,elimNum))
            //add irredundant inequality to new set
            copyIneq(*newIneq, &news[sz++], varNum, ineqNum);

      }
    }
  size = sz;
  (*finalSize) = sz;
  free(positives);
  free(negatives);
  return (news);
}
```

Listing 4.4: Eliminate one variable

Using this function, Listing 4.5 shows the function to find the minimal projective representation. This function reads the input data from file, then, for each variable, finds its redundancy test cone and calls FMEOneStepNoRedundant to get the minimal projection. The union of minimal representation with respect to each variable is the minimal projective representation.

```
inequality * project(char * fileName, int varNum, int ineqNum)
{
  int * size = (int *) malloc(sizeof(int));
  inequality * data = (inequality *) malloc(ineqNum * sizeof(inequality));
  inequality * inputData = (inequality *) malloc(ineqNum * sizeof(inequality));

  //get input data from the input file
  getFromFile(inputData, fileName, varNum, ineqNum);
  copyListOfInequalities(inputData, data, varNum, ineqNum);

  inequality * irredundantProject = (inequality *) malloc(sizeof(inequality));
  inequality * currentSet = (inequality *) malloc(sizeof(inequality));

  int * finalSize = (int *) malloc(sizeof(int));

  currentSet = data;
  (*size) = ineqNum;

  for (int j = 1; j < varNum; j++) //iterate over variables
  {
    inequality * W0;

    W0 = balasW0(data, j, varNum, ineqNum); //find cone W0 w.r.t. variable

    //find redundancy test cone
    mpq_t * mat = blockElimination(W0, j , varNum-j , ineqNum+1 , finalSize);

    //get irredundant projection
    irredundantProject = FMEOneStepNoRedundant(inputData, currentSet, varNum,
        ineqNum, *size, j, size, mat, *finalSize);

    currentSet = irredundantProject; //make the set for the next iteration
  }
  return(irredundantProject);
}
```

Listing 4.5: The minimal projective representation

Using arrays for storing inequality systems makes implementation simpler. Also, in comparison with linked lists, cache usage is more efficient because all inequalities are next to each other in the memory. On the other hand, because we do not know the number of non redundant inequalities in advance, we have to allocate memory for the worst case. Moreover, for the same reason, we need to store all the newly generated inequalities in the same array and call the partition function for each pass. We try to overcome these drawbacks in the second implementation.

4.4.2 The Unrolled linked list representation

Linked lists, like arrays, are data structures for storing more than one instance of a data structure. Unlike arrays, different elements of linked lists are not in consecutive locations in memory. Therefore, each node needs to store a pointer to the next node.

An unrolled linked list is a kind of linked list such that each node contains an array of the specified data structure rather than only one. Figure 4.2 shows an array, while Figure 4.3 shows the unrolled linked list corresponding to this array.

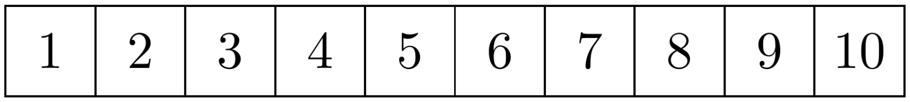

Figure 4.2: Illustration of an array

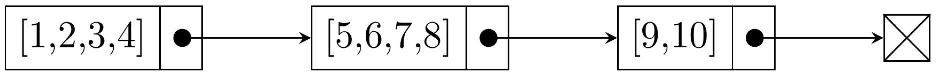

Figure 4.3: Illustration of the corresponding unrolled linked list

In our case of using unrolled linked lists, each node contains an array of the inequality data structure. Listing 4.6 and Listing 4.7 show the programs of the unrolledlinkedlist data structure and its basic functions for creating and adding new elements, respectively.

The main drawback of using linked lists is the high number of cache misses they have. We can solve this problem by using unrolled linked lists and carefully selecting the number of elements to be stored in each node. In our codes, the defined variable LL_SIZE is the number of inequalities we store in each node of the unrolled lined list. It should be selected such that the whole node fits in the cache memory. Therefore, the exact number of this variable depends on the specific hardware. Other than a data array and a pointer to the next node, we also need another variable to shows how many elements have been stored in each node. The fill variable shows this number. It is initialized to zero and it increases when we add a new inequality. When it reaches LL_SIZE, a new node is created.

```
typedef struct node
{
  inequality data[LL_SIZE]; //data array in the node
  int fill; //number of stored inequalities
  struct node * next; // pointer to the next node
}node;

typedef struct linkedList
{
  node * head; //the head node of list
  int number; //number of nodes in list
}linkedList;
```

Listing 4.6: Data structure of a node and an unrolled linked list

```
#include "inequality.h"

node* makeNode(int varNum , int ineqNum)
{
  node * new = (node *) malloc(sizeof(node));

  for(int i = 0 ; i < LL_SIZE ; i++)
    makeNewInequality(&(new->data[i]), varNum , ineqNum);

  new->fill = 0;
  new->next = NULL;

  return(new);
}

linkedList makeList(int varNum , int ineqNum)
{
  linkedList new;
  node * s = makeNode(varNum , ineqNum);
  new.head = s;
  new.head->fill = LL_SIZE; //head node does not store inequality
  new.number = 0;
  return (new);
}

void addList(linkedList * l , inequality d , int varNum , int ineqNum)
{
  node * current = l->head;

  while(current->next != NULL)
    current = current->next;

  if(current->fill < LL_SIZE) //current node is not full
  {
    for(int k = 0 ; k < varNum ; k++) //copy coefficients
      mpq_set(current->data[current->fill].coef[k] , d.coef[k]);

    for(int k = 0 ; k < ineqNum ; k++) //copy history set
      current->data[current->fill].history[k] = d.history[k];

    mpq_set(current->data[current->fill].constant , d.constant);

    current->fill++; //increase number of stored inequalities in the node
  }
  else //allocate new node to store new inequality
  {
    current->next = makeNode(varNum , ineqNum); //make new node

    for(int k = 0 ; k < varNum ; k++)  //copy coefficients
      mpq_set(current->next->data[0].coef[k], d.coef[k]);

    for(int k = 0 ; k < ineqNum ; k++) //copy history set
      current->next->data[0].history[k] = d.history[k];

          mpq_set(current->next->data[0].constant , d.constant);

    current->next->fill++; //increase number of stored inequalities in the new node
    l->number++; //increase number of nodes in the list
  }
}
```

Listing 4.7: Functions for creating and adding elements

In the implementation using unrolled linked list representation, to eliminate each variable we have the following steps:

1. for all inequalities in the positive list and all in the negative list, combine them using the combine function,
2. check whether the newly generated inequality is redundant or not, using the kohlerCheck function,
3. if it passes Kohler's check, check it with the balasCheck function,
4. if it passes Balas's check, add it to a new positive, negative or zero list with respect to its next variable's coefficient
5. after considering all inequalities in the positive list with all in the negative list, add inequalities in the zero list to the new positive, negative or zero list with respect to its next variable's coefficient

Listing 4.8 shows the implementation of this process. Detailed implementation of functions KohlerCheck and balashCheck can be found in Chapter B

```
void newFMEOneStepNoRedundant(linkedList positives, linkedList negatives,
    linkedList zeros, int v, int varNum, int ineqNum, linkedList * newpos,
    linkedList * newneg, linkedList * newzer, linkedList * inputList, mpq_t * pw0,
    int sizepw0)
{
  node * currentPos = positives.head->next;
  node * currentNeg = negatives.head->next;
  node * currentZer = zeros.head->next;
  inequality * newIneq = (inequality *) malloc (sizeof(inequality));
  makeNewInequality(newIneq, varNum, ineqNum);

  for (int i1 = 0; i1 < positives.number; i1++) //iterate over positive set nodes
  {
    //iterate over positive node inequalities
    for (int j1 = 0; j1 < currentPos->fill; j1++)
    {
      currentNeg = negatives.head->next;
      //iterate over negative set nodes
      for (int i2 = 0; i2 < negatives.number; i2++)
      {
        //iterate over negative node inequalities
        for (int j2 = 0; j2 < currentNeg->fill; j2++)
        {
          combine(currentPos->data[j1], currentNeg->data[j2], newIneq, v, varNum,
    ineqNum);
          if (kohlerCheck(*newIneq, inputList, v, varNum, ineqNum) == 1)
            if (balasCheck(*newIneq, pw0, varNum, ineqNum, sizepw0, v) == 1)
            {
              if (mpq_sgn(newIneq->coef[v + 1]) > 0)
                addList(newpos, *(newIneq), varNum, ineqNum);
              else if (mpq_sgn(newIneq->coef[v + 1]) < 0)
                addList(newneg, *(newIneq), varNum, ineqNum);
              else
                addList(newzer, *(newIneq), varNum, ineqNum);
            }
        }
        currentNeg = currentNeg->next;
      }
    }
  currentPos = currentPos->next;
  }
  //check inequalities with zero coefficient for redundancy
  for (int i3 = 0; i3 < zeros.number; i3++) //iterate over zero set nodes
  {
    //iterate over zero node inequalities
    for (int j3 = 0; j3 < currentZer->fill; j3++)
      if (balasCheck(currentZer->data[j3], pw0, varNum, ineqNum, sizepw0, v)== 1)
      {
        if (mpq_sgn(currentZer->data[j3].coef[v + 1]) > 0)
          addList(newpos, currentZer->data[j3], varNum, ineqNum);
        else if (mpq_sgn(currentZer->data[j3].coef[v + 1]) < 0)
          addList(newneg, currentZer->data[j3], varNum, ineqNum);
        else
          addList(newzer, currentZer->data[j3], varNum, ineqNum);
      }
    currentZer = currentZer->next;
  }
}
```

Listing 4.8: Eliminate one variable in list

Having this function, Listing 4.9 shows the function to find the minimal projective representation.

```
linkedList project(char * fileName, int varNum, int ineqNum)
{
  int * size = (int *) malloc(sizeof(int));
  inequality * data = (inequality *) malloc(ineqNum * sizeof(inequality));
  inequality * inputData = (inequality *) malloc(ineqNum * sizeof(inequality));

  //read data from file
  getFromFile(inputData, fileName, varNum, ineqNum);

  linkedList inputll;
  inputll = makeList(varNum , ineqNum);

  for(int i = 0 ; i < ineqNum ; i++)
    addList(&inputll , inputData[i] , varNum , ineqNum);

  linkedList datall = makeList(varNum , ineqNum);

  for (int i = 0; i < ineqNum; i++) //make list from input inequalities
    addList(&datall, inputData[i], varNum, ineqNum);

  linkedList outputll = makeList(varNum , ineqNum);

  linkedList pos = makeList(varNum, ineqNum);
  linkedList neg = makeList(varNum, ineqNum);
  linkedList zer = makeList(varNum, ineqNum);

  linkedList newPos = makeList(varNum, ineqNum);
  linkedList newNeg = makeList(varNum, ineqNum);
  linkedList newZer = makeList(varNum, ineqNum);

  int * finalSize = (int *) malloc(sizeof(int));

  //parition the input system
  partition(&(datall), &pos, &neg, &zer, 0, varNum, ineqNum);

  for (int j = 0; j < varNum - 1; j++)
  {
    //find W0 cone w.r.t. the variable
    linkedList w0ll = makeList(ineqNum + 1 , ineqNum + 1);
    balasW0(datall, j+1, varNum, ineqNum , &w0ll);

    //representation of redundancy test cone
    mpq_t * mat = blockElimination(w0ll, j+1 , varNum - j - 1, ineqNum + 1,
    finalSize);

    newPos = makeList(varNum, ineqNum);
    newNeg = makeList(varNum, ineqNum);
    newZer = makeList(varNum, ineqNum);

    //find minimal projection with fme
    newFMEOneStepNoRedundant(pos, neg , zer , j, varNum,
        ineqNum, &(newPos), &(newNeg), &(newZer), &(datall), mat,
        *finalSize);

    freeList(&pos, varNum, ineqNum);
    freeList(&neg, varNum, ineqNum);
    freeList(&zer, varNum, ineqNum);

    pos.head->fill = newPos.head->fill;
    pos.head->next = newPos.head->next;
    pos.number = newPos.number;

    neg.head->fill = newNeg.head->fill;
    neg.head->next = newNeg.head->next;
    neg.number = newNeg.number;

    zer.head->fill = newZer.head->fill;
    zer.head->next = newZer.head->next;
    zer.number = newZer.number;

    free(mat);

  }

  join(pos , neg , zer , &(outputll) , varNum , ineqNum);

  freeList(&pos, varNum, ineqNum);
  freeList(&neg, varNum, ineqNum);
  freeList(&zer, varNum, ineqNum);
  free(finalSize);
  free(size);

  return(outputll);
}
```

Listing 4.9: Minimal projective representation

Using unrolled linked lists we do not need to allocate more memory than we need and also we can put a new inequality in its proper list for the next variable elimination. Although adding an inequality to the list needs more time compared with adding it to an array, this approach enables us to consider large inequality systems.

4.5 Experimental results

In this section, we examine our experimental results. We test the running time of four implementations we have, against each other and also against the PolyhedralSets : −Projection command of MAPLE. Also, we show the effectiveness of different methods for eliminating redundancy by comparing the maximum number of inequalities we reach in the process of FME.

Table 4.1 provides the specification of our test cases. All test cases are consistent linear inequality systems and they are generated randomly. The specification of each test case is shown based on the number of initial inequalities, dimension of space (number of variables) and maximum absolute value of coefficients.

Tests	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13
#var	5	10	4	5	5	7	10	6	5	10	9	8	6
#ineq	10	12	8	10	8	10	12	8	11	20	19	19	18
max-val	99	10	20	20	98	99	90	97	100	1300	37345	999	99852

Table 4.1: Test cases specification

Table 4.2 shows the running time comparisons. The first column, (`Imp1`), shows the running time for the array representation and FME improved by Kohler's method for finding redundancy test cones. The second column, (`Imp2`), shows the running time for the array representation and the CDD library function for finding redundancy test cones. The third, (`Imp3`), and fourth (`Imp4`) columns are the respective results for the implementation of the same two algorithms in the list representation. The `Maple` column shows the running time for MAPLE. The last two columns show running time of Fourier elimination function in CDD library. The `CDD1` column is the running time using `dd_MatrixCanonicalize` function, to eliminate redundancies and `CDD2` uses Clarkson's algorithm, `dd_RedundantRowsViaShooting` function for redundancy elimination. Clarkson's algorithm [8] is a method to minimize number of LP solving needed for finding minimal representation. This method makes the algorithm more efficient. [1]

[1]Because the running time of the algorithm for eliminating all variables is too long, for some cases we only remove *some* of the variables. The numbers in level parts shows this number.

Case	Imp1	Imp2	Imp3	Imp4	Maple	CDD1 (level)	CDD2
t1	0.044692	0.020723	0.045387	0.012935	7.974	0.139568	0.050889
t2	0.077096	0.132471	0.047491	0.081380	3321.217	123.724047	8.503278
t3	0.007723	0.007387	0.007302	0.004062	0.736	0.004216	0.002026
t4	0.028956	0.018163	0.027712	0.010712	2.579	0.048190	0.018195
t5	0.009344	0.010665	0.009263	0.007122	3.081	0.032876	0.013812
t6	0.037363	0.046401	0.034927	0.029198	117.021	core dump	wrong result
t7	0.224652	0.274195	0.220846	0.158917	665.194 (2)	324.737169 (3)	59.388326
t8	0.009453	0.014710	0.009502	0.009344	4.950	0.125620	0.025555
t9	0.059641	0.026069	0.058178	0.014430	8.229	0.075116	0.039896
t10	core dump	core dump	201.182032	3.791912	240.182 (1)	1037.671452 (2)	810.589480 (3)
t11	core dump	core dump	587.921003	8.248047	192.898 (1)	2274.081326 (2)	66.623110(2)
t12	core dump	core dump	210.538391	2.648560	188.408 (1)	839.965153 (2)	25.842749 (2)
t13	core dump	core dump	14.644834	0.505207	911.354 (2)	78.954814	30.704574

Table 4.2: Running time comparison (second)

Table 4.3 shows the effectiveness of each redundancy elimination method by comparing the maximum number of inequalities generated by the FME algorithm. The first column shows this number when only using Kohler's method, the second column is this number when using Balas and Kohler, and the last one is this number when there is no check. In the original column of this table, N/A in the level column means that the program terminates normally, while the number in this column shows the level at which the program crashes because of the high number of redundancies.

Test case	Kohler	Kohler-Balas	original (level)
t1	36	18	4840 (3)
t2	88	66	9000 (3)
t3	20	11	56
t4	33	19	50736
t5	20	14	147679
t6	40	37	4773 (3)
t7	87	82	21384 (3)
t8	18	15	64386 (4)
t9	51	20	2048 (5)
t10	52	18	9039 (3)
t11	695	362	1920 (2)
t12	620	257	1932 (2)
t13	435	91	1292 (2)

Table 4.3: Number of inequalities after eliminating redundant inequalities

Chapter 5

Related work

During our study of the Fourier-Motzkin elimination, we found many related works. As discussed above, removing redundant inequalities during the execution of Fourier-Motzkin elimination is the central issue towards efficiency. Up to our knowledge, all available implementation of Fourier-Motzkin elimination relies on linear programming for remove redundant inequalities, an idea suggested in [24]. However, and as mentioned before, alternative approaches rely on linear algebra.

In [6], Chink proposed a redundancy test with little added work and which greatly improves the practical efficiency of Fourier-Motzkin elimination. Kohler proposed a method [25] which only use matrix arithmetic operations to test the redundancy of inequalities. As observed by Imbert in his work [18], the method he proposed in this paper as well as those of Chernikov and Kohler are essentially equivalent. Even though these works are very effective in practice, none of them can remove all redundant inequalities generated by Fourier-Motzkin elimination.

Besides Fourier-Motzkin elimination, block elimination is another algorithmic tool to project polyhedra on a lower dimensional subspace. This method relies on the extreme rays of the so-called projection cone. Kohler's work [25] is based on this latter concept. Although there exists efficient methods to enumerate the extreme rays of this projection cone, like the *double description method* [13] (also known as Chernikova's algorithm [7, 26]), this method can not remove all the redundant inequalities.

In [1], Balas shows that if certain invertibility conditions are satisfied, then the extreme rays of the projection cone exactly defines the minimal representation of the projected polyhedron. As Balas mentions in his paper, this method can be extended to any polytope. In our work, we observe that this method can be extended to all the pointed polyhedra. Via experimentation, we found the results and constructions in Balas' paper had some flaws:

1. In Balas' work, the projection cone is defined as $W^0 := \{(\mathbf{v}, \mathbf{w}, v_0) \in \mathbb{Q}^q \times \mathbb{Q}^{m-q} \times \mathbb{Q} \mid (\mathbf{v}, \mathbf{w})B_0^{-1}A_0 = 0, -(\mathbf{v}, \mathbf{w})B_0^{-1}\mathbf{c}_0 + v_0 = 0, (\mathbf{v}, \mathbf{w})B_0^{-1} \geq 0\}$ and the author claims that $\mathbf{vx} \leq v_0$ defines a facet of the projected cone $\mathsf{proj}_{\mathbf{x}}(Q)$ if and only if $(\mathbf{v}, v_0)$ is an extreme ray of the redundancy test cone $\mathsf{proj}_{\mathbf{v},v_0}(W^0)$. However, we have a counter example for this claim. Please refer to the page `http://www.jingrj.com/worksheet.html`. In this example, when we eliminate two variables, the cone $\mathsf{proj}_{\mathbf{v},v_0}(W^0)$ has 19 extreme rays while $\mathsf{proj}_{\mathbf{x}}(Q)$ has 18 facets. 18 of the 19 extreme rays of $\mathsf{proj}_{\mathbf{v},v_0}(W^0)$ give out the 18 facets of $\mathsf{proj}_{\mathbf{x}}(Q)$, while the remaining extreme ray gives out a redundant inequality *w.r.t.* the 18 facets. The main reason leading to this situation is due to the use of

Fakars' lemma in the proof of Balas' paper. We improved this situation by changing $-(\mathbf{v}, \mathbf{w})B_0^{-1}\mathbf{c}_0 + v_0 = 0$ to $-(\mathbf{v}, \mathbf{w})B_0^{-1}\mathbf{c}_0 + v_0 \geq 0$ and carefully showed the relations between the extreme rays of $\mathsf{proj}_{\mathbf{v},v_0}(W^0)$ and the facets of $\mathsf{proj}_{\mathbf{x}}(Q)$, for the details please refer to Theorem 3.1 and proposition 3.2 and Corollary 3.3.

2. In Balas' paper, the author suggests to enumerate the extreme rays of the redundancy test cone $\mathsf{proj}_{\mathbf{v},v_0}(W^0)$ to produce the minimal representation of $\mathsf{proj}_{\mathbf{x}}(Q)$, which is very consuming. Like Kohler's method, our technique tests the redundancy of the inequality $\mathbf{v}\mathbf{x} \leq v_0$ in $S_p^{(K)}$ by testing whether $(\mathbf{v}, v_0)$ is an extreme ray of the redundancy test cone $\mathsf{proj}_{\mathbf{v},v_0}(W^0)$.

Combining Kohler's method and our improved version of Balas' methods, we obtain an algorithm to remove all the redundant inequalities produced by Fourier-Motzkin elimination. Even though this algorithm still has exponential complexity, we found it is very effective in practice, as what we have shown in Chapter 4.

The projection of polyhedra is a useful tool to solve problem instances in parametric linear programming, which plays an important role in the analysis, transformation and scheduling of for-loops of computer programs, see for instance [5, 22, 23].

Chapter 6

Solving parametric linear programming problem with Fourier-Motzkin elimination

In this chapter, we show how to use Fourier-Motzkin elimination for solving parametric linear programming (PLP) problem instances.

Given a PLP problem instance:

$$\begin{aligned} z(\mathbf{\Theta}) &= \min \mathbf{c}\mathbf{x} \\ A\mathbf{x} &\le B\mathbf{\Theta} + \mathbf{b} \end{aligned} \tag{6.1}$$

where $A \in \mathbb{Z}^{m\times n}, B \in \mathbb{Z}^{m\times p}, \mathbf{b} \in \mathbb{Z}^m$, and $\mathbf{x} \in \mathbb{Q}^n$ are the variables, $\mathbf{\Theta} \in \mathbb{Q}^p$ are the parameters.

To solve this problem, first we need the following preprocessing step. Let $g > 0$ be the greatest common divisor of elements in $\mathbf{c}$. Via Gaussian elimination, we can obtain a unimodular matrix $U \in \mathbb{Q}^{n\times n}$ satisfying $[0, \ldots, 0, g] = \mathbf{c}U$. Let $\mathbf{t} = U^{-1}\mathbf{x}$, the above PLP problem can be transformed to the following equivalent form:

$$\begin{aligned} z(\mathbf{\Theta}) &= \min g t_n \\ AU\mathbf{t} &\le B\mathbf{\Theta} + \mathbf{b}. \end{aligned} \tag{6.2}$$

Applying Algorithm 10 to the constraints $AU\mathbf{t} \le B\mathbf{\Theta} + \mathbf{b}$ with the variable order $t_1 > \cdots > t_n > \mathbf{\Theta}$, we obtain $\mathsf{ProjRep}(Q; t_1 > \cdots > t_n > \mathbf{\Theta})$, where $Q \subseteq \mathbb{Q}^{n+p}$ is the polyhedron represented by $AU\mathbf{t} \le B\mathbf{\Theta} + \mathbf{b}$. We extract the representation of the projection $\mathsf{proj}_{t_n,\mathbf{\Theta}}(Q)$, denoted by $\Phi := \Phi_1 \cup \Phi_2$. Here we denote by Φ_1 the set of inequalities which have a non-zero coefficient in t_n and Φ_2 the set of inequalities which are free of t_n. Since $g > 0$, we only need to consider the lower bound of t_n, which is very easy to deduce from Φ_1.

Consider Example 3.3 in [5]:

$$\begin{aligned} &\min \quad -2x_1 - x_2 \\ &\begin{cases} x_1 + 3x_2 \le 9 - 2\theta_1 + \theta_2, 2x_1 + x_2 \le 8 + \theta_1 - 2\theta_2 \\ x_1 \le 4 + \theta_1 + \theta_2, \quad -x_1 \le 0, \quad -x_2 \le 0 \end{cases} \end{aligned}$$

We have $(-2,-1)U = (0,1)$, where $U = \begin{pmatrix} 1 & 0 \\ -2 & -1 \end{pmatrix}$. Let $(t_1, t_2)^T = U^{-1}(x_1, x_2)^T$, the above PLP problem is equivalent to

$$\min \quad t_2$$
$$\begin{cases} -5t_1 - 3t_2 \le 9 - 2\theta_1 + \theta_2, -t_2 \le 8 + \theta_1 - 2\theta_2 \\ t_1 \le 4 + \theta_1 + \theta_2, \quad -t_1 \le 0, \quad 2t_1 + t_2 \le 0 \end{cases}$$

Let P denote the polyhedron represented by the above constraints. Applying Algorithm 10 to P with variable order $t_1 > t_2 > \theta_1 > \theta_2$, we obtain the projected representation $\mathsf{ProjRep}(P; t_1 > t_2 > \theta_1 > \theta_2)$, from which we can easily extract the representation of the projected polyhedron $\mathsf{proj}_{t_2,\theta_1,\theta_2}(P)$:

$$\mathsf{proj}_{t_2,\theta_1,\theta_2}(P) := \begin{cases} -t_2 - \theta_1 + 2\theta_2 \le 8, & -3t_2 - 3\theta_1 - 6\theta_2 \le 29, \\ -t_2 + 4\theta_1 - 2\theta_2 \le 18, & t_2 \le 0, \\ -\theta_1 - \theta_2 \le 4, & -\theta_1 + 2\theta_2 \le 8, \\ -3\theta_2 \le 17, & 3\theta_2 \le 25. \end{cases}$$

t_2 has three lower bounds: $t_2 = -8 - \theta_1 + 2\theta_2, t_2 = -\theta_1 - 2\theta_2 - 29/3$ and $t_2 = 4\theta_1 - 2\theta_2 - 18$, under the constraints

$$\begin{cases} -\theta_2 \le 5/12, & -\theta_1 - \theta_2 \le 4, \\ \theta_1 + 2\theta_2 \le 8, & \theta_1 - 4/5\theta_2 \le 2. \end{cases}, \begin{cases} \theta_2 \le -5/12, \theta_1 \le 5/3, \\ -\theta_1 - \theta_2 \le 4. \end{cases}, \begin{cases} -\theta_1 \le -5/3, -\theta_1 + 4/5\theta_2 \le -2, \\ \theta_1 - \theta_2/2 \le 9/2 \end{cases}$$

Bibliography

[1] Egon Balas. Projection with a minimal system of inequalities. *Computational Optimization and Applications*, 10(2):189–193, 1998.

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[3] M. Benabderrahmane, L. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, June 2008.

[5] Francesco Borrelli, Alberto Bemporad, and Manfred Morari. Geometric algorithm for multiparametric linear programming. *Journal of optimization theory and applications*, 118(3):515–540, 2003.

[6] Sergei Nikolaevich Chernikov. Contraction of systems of linear inequalities. *Doklady Akademii Nauk SSSR*, 131(3):518–521, 1960.

[7] Natal'ja V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 5(2):334–337, 1965.

[8] Kenneth L Clarkson. More output-sensitive geometric algorithms. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 695–702. IEEE, 1994.

[9] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.

[10] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pages 79–103, London, UK, UK, 1996. Springer-Verlag. `http://dl.acm.org/citation.cfm?id=647429.723579`.

[11] K Fukuda. The CDD and CDDplus homepage. `https://www.inf.ethz.ch/personal/fukudak/cdd_home/`.

[12] Komei Fukuda et al. Frequently asked questions in polyhedral computation. *Swiss Federal Institute of Technology, Lausanne and Zurich, Switzerland. Available at: ftp://ftp. ifor. math. ethz. ch/pub/fukuda/reports/polyfaq040618. pdf*, 2004.

[13] Komei Fukuda and Alain Prodon. Double description method revisited. In *Combinatorics and computer science*, pages 91–111. Springer, 1996.

[14] Torbjrn Granlund and the GMP Development Team. *GNU MP 6.0 Multiple Precision Arithmetic Library*. Samurai Media Limited, United Kingdom, 2015.

[15] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L. Pouchet. Polly - polyhedral optimization in llvm. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.

[16] William Hart, Fredrik Johansson, and Sebastian Pancratz. Flint–fast library for number theory. 2011.

[17] William B. Hart. Fast library for number theory: An introduction. In *Proceedings of the Third International Congress Conference on Mathematical Software*, ICMS'10, pages 88–91, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] Jean-Louis Imbert. Fourier's elimination: Which to choose? In *PPCP*, pages 117–129, 1993.

[19] Rui-Juan Jing and Marc Moreno Maza. Computing the integer points of a polyhedron, I: algorithm. In *Computer Algebra in Scientific Computing - 19th International Workshop, CASC 2017, Beijing, China, September 18-22, 2017, Proceedings*, pages 225–241, 2017.

[20] Rui-Juan Jing and Marc Moreno Maza. Computing the integer points of a polyhedron, II: complexity estimates. In *Computer Algebra in Scientific Computing - 19th International Workshop, CASC 2017, Beijing, China, September 18-22, 2017, Proceedings*, pages 242–256, 2017.

[21] Rui-Juan Jing, Marc Moreno Maza, and Delaram Talaashrafi. Complexity estimates for fourier-motzkin elimination. *CoRR*, abs/1811.01510, 2018.

[22] Colin Jones. Polyhedral tools for control. Technical report, University of Cambridge, 2005.

[23] 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.

[24] Leonid Khachiyan. Fourier-motzkin elimination method. In Christodoulos A. Floudas and Panos M. Pardalos, editors, *Encyclopedia of Optimization, Second Edition*, pages 1074–1077. Springer, 2009.

[25] David A. Kohler. Projections of convex polyhedral sets. Technical report, California Univ. at Berkeley, Operations Research Center, 1967.

[26] Hervé Le Verge. *A note on Chernikova's algorithm*. PhD thesis, INRIA, 1992.

[27] Peter McMullen. The maximum numbers of faces of a convex polytope. *Mathematika*, 17(2):179–184, 1970.

[28] David Monniaux. Quantifier elimination by lazy model enumeration. In *International Conference on Computer Aided Verification*, pages 585–599. Springer, 2010.

[29] Arnold Schönhage and Volker Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3-4):281–292, 1971.

[30] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[31] Arne Storjohann. *Algorithms for matrix canonical forms*. PhD thesis, Swiss Federal Institute of Technology Zurich, 2000.

[32] Marco Terzer. *Large scale methods to enumerate extreme rays and elementary modes*. PhD thesis, ETH Zurich, 2009.

[33] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *TACO*, 9(4):54, 2013.

Appendix A

Array Representation Detailed Program

Kohler check function:

```c
#include "inequality.h"
#include "linAlg.h"

int kohlerCheck(inequality a, inequality * d, int  elim, int v , int varNum , int
    ineqNum)
{

  int k = 0;
  int el = 0;
  int col = 0;
  int row = 0;
  int rank;
  mpq_t temp;
  mpq_init(temp);

  col = v + 1  ;
  mpq_t * mat = (mpq_t *) malloc(ineqNum * varNum * sizeof(mpq_t));

  for (int i = 0; i < ineqNum * varNum; i++)
    mpq_init(mat[i]);

  for (int i = 0; i < ineqNum; i++)
    if (a.history[i] == 1)
    {
      for (j = 0; j <= v; j++)
        mpq_set(mat[k++], d[i].coef[j]);
      row++;
    }

  rank = rankOfMatrix(mat, row, col);

  free(mat);
  if (rank == row - 1)
    return 1;
  else
    return 0;
}
```

Listing A.1: Kohler check function in array representation

Balas check function and its related functions:

```
int balasCheck(inequality a, mpq_t * pw0, int varNum, int ineqNum , int sizepw0 ,
    int elimNum)
{
  int r;
  int vcheck = 0;
  vcheck = varNum - elimNum + 1;
  mpq_t * v = (mpq_t *) malloc((varNum - elimNum + 1) * sizeof(mpq_t));
  for(int i = 0 ; i < varNum - elimNum + 1 ; i++)
    mpq_init(v[i]);

  for(int i = elimNum ; i < varNum ; i++)
    mpq_set(v[i - elimNum] , a.coef[i]);
  mpq_set(v[varNum - elimNum] , a.constant);
  r = checkExtreme(pw0, v, varNum - elimNum + 1, sizepw0 , vcheck);
  return (r);
}
```

Listing A.2: Balas check function in array representation

```
int checkExtreme(mpq_t * A, mpq_t * v, int varNum, int ineqNum, int vcheck)
{

  mpq_t * d = (mpq_t *) malloc(ineqNum * sizeof(mpq_t));
  matrixMatrixMult(A, v, d, ineqNum, varNum, 1);

  int e = 0;
  for (i = 0; i < ineqNum; i++)
    if (mpq_sgn(d[i]) == 0)
      e++;

  mpq_t * w = (mpq_t *) malloc(e * varNum * sizeof(mpq_t));
  for (i = 0; i < e * varNum; i++)
    mpq_init(w[i]);

  int c = 0;
  for (i = 0; i < ineqNum; i++)
    if (mpq_sgn(d[i]) == 0)
      for (j = 0; j < varNum; j++)
        mpq_set(w[c++], A[i * varNum + j]);
  int r = rankOfMatrix(w, e, varNum);
  if (r == vcheck - 1)
    return (1);
  else
    return (0);
}
```

Listing A.3: Check extreme ray function in array representation

```
inequality * balasW0(inequality * data, int elimNumber, int varNum, int ineqNum)
{

  //find matrix A
  mpq_t * A1 = (mpq_t *) malloc(elimNumber * ineqNum * sizeof(mpq_t));
  for (int k = 0; k < elimNumber * ineqNum; k++)
    mpq_init(A1[k]);

  mpq_t * A = (mpq_t *) malloc(elimNumber * ineqNum * sizeof(mpq_t));
  for (int k = 0; k < elimNumber * ineqNum; k++)
```

```
    mpq_init(A[k]);

  whatIsA(data, A, elimNumber, ineqNum);

  //find matrix B
  mpq_t * B1 = (mpq_t *) malloc(
      (varNum - elimNumber) * ineqNum * sizeof(mpq_t));
  for (int k = 0; k < (varNum - elimNumber) * ineqNum; k++)
    mpq_init(B1[k]);

  mpq_t * B = (mpq_t *) malloc(
      (varNum - elimNumber) * ineqNum * sizeof(mpq_t));
  for (int k = 0; k < (varNum - elimNumber) * ineqNum; k++)
    mpq_init(B[k]);

  whatIsB(data, B, elimNumber, ineqNum, varNum);

  //find vector d
  mpq_t * d1 = (mpq_t *) malloc(ineqNum * sizeof(mpq_t));
  for (int k = 0; k < ineqNum; k++)
    mpq_init(d1[k]);

  mpq_t * d = (mpq_t *) malloc(ineqNum * sizeof(mpq_t));
  for (int k = 0; k < ineqNum; k++)
    mpq_init(d[k]);

  whatIsd(data, d, ineqNum);

  //put independent rows at first
  orderMatrix(A, B, d, A1, B1, d1, ineqNum, (varNum - elimNumber),
      elimNumber);

  //fide B0 matrix
  mpq_t * B0 = (mpq_t *) malloc(ineqNum * ineqNum * sizeof(mpq_t));
  for (int k = 0; k < ineqNum * ineqNum; k++)
    mpq_init(B0[k]);

  whatIsB0(B1, B0, ineqNum, varNum - elimNumber, elimNumber);

  //find W0 cone
  inequality *W0; //<----
  W0 = whatIsW0(A1, B0, d1, ineqNum, varNum - elimNumber, elimNumber,
      varNum - elimNumber);
  return (W0);
}
```

Listing A.4: Make up W^0 function in array representation

```
mpq_t * blockElimination(inequality * w0, int p, int q, int m , int * size)
{
  mpq_t * coefMat = (mpq_t *) malloc((m * m) * sizeof(mpq_t));

  for(int i = 0 ; i < (m * m) ; i++)
    mpq_init(coefMat[i]);

  coefMatrix(w0, coefMat , m, m);

  int pprime = pPrime(coefMat, p, m);

  q = q + 1;
  int qprime = q + pprime;
  mpq_t * out1 = (mpq_t *) malloc(((m - qprime) * m) * sizeof(mpq_t));
  mpq_t * out2 = (mpq_t *) malloc(((q) * m) * sizeof(mpq_t));

  for (int i = 0; i < ((m - qprime) * m); i++)
    mpq_init(out1[i]);

  for (int i = 0; i < ((q) * m); i++)
    mpq_init(out2[i]);

  projectionCone(coefMat , out1 , out2, pprime , q , m);

  mpq_t * extrIn = (mpq_t *) malloc((m - qprime) * qprime * sizeof(mpq_t));
  for(int i = 0 ; i < (m - qprime )* qprime ; i++)
    mpq_init(extrIn[i]);
  for(int i = 0 ; i < (m-qprime) * qprime ; i++)
    mpq_set_d(extrIn[i] , 0);

  makeExtrInput(out1 , extrIn , m , qprime);

  mpq_t * extrs;
  int grow, gcol;
  extrs = extr(extrIn, m, qprime, &grow , &gcol);

  *size = grow;
  mpq_t * mat = (mpq_t *) malloc(grow * q * sizeof(mpq_t));
  for(int i = 0 ; i < grow * q ; i++)
    mpq_init(mat[i]);

  mpq_t * extrVec = (mpq_t *) malloc(m * sizeof(mpq_t));
  for(int i = 0 ; i < m ; i++)
    mpq_init(extrVec[i]);

  mpq_t * rowResult = (mpq_t *) malloc(q * sizeof(mpq_t));
  for(int i = 0 ; i < q ; i++)
    mpq_init(rowResult[i]);

  int v = 0;
  int c = 0;
  int cmat = 0;

  for(int i = 0 ; i < grow ; i++)
  {
    for(int j = 0 ; j < m ; j++)
      mpq_set(extrVec[j] , extrs[c++]);

    matrixMatrixMult(extrVec, out2 , rowResult, 1 , m , q);

    for(int j = 0 ; j < q ; j++)
      mpq_set(mat[cmat++] , rowResult[j]);
  }
  return(mat);
}
```

Listing A.5: Block eliminate function in array representation

Appendix B

Unrolled Linked List Representation Detailed Program

Kohler check function:

```
#include "inequality.h"
#include "linAlg.h"
#include "balas.h"
#include "unrolledll.h"

int kohlerCheck(inequality a, linkedList * d, int v , int varNum , int ineqNum)
{
  int k = 0;
  int el = 0;
  int col = 0;
  int row = 0;

  mpq_t temp;
  mpq_init(temp);

  col = v + 1  ;
  mpq_t * mat = (mpq_t *) malloc(ineqNum * varNum * sizeof(mpq_t));

  for (i = 0; i < ineqNum * varNum; i++)
    mpq_init(mat[i]);

  node * current = d->head;

  for (int i = 0; i < ineqNum; i++)
    if (a.history[i] == 1)
    {
      int no = (i / LL_SIZE) + 1;
      int da = i % LL_SIZE;

      current = d->head;

      for(int i1 = 0 ; i1 < no ; i1++)
        current = current->next;

      for (int j = 0; j <= v; j++)
        mpq_set(mat[k++], current->data[da].coef[j]);

      row++;
    }

  int rank = rankOfMatrix(mat, row, col);
  for(i = 0 ; i < ineqNum * varNum ; i++)
    mpq_clear(mat[i]);
  free(mat);

  if (rank == row - 1)
    return 1;
  else
    return 0;
}
```

Listing B.1: Kohler check function in unrolled linked list representation

Balas check function and its related functions:

```
int balasCheck(inequality a, mpq_t * pw0, int varNum, int ineqNum , int sizepw0 ,
    int elimNum)
{
  int vcheck = 0;
  int vcheck = varNum - elimNum /* + 1*/;

  mpq_t * v = (mpq_t *) malloc((varNum - elimNum) * sizeof(mpq_t));
  for(int i = 0 ; i < varNum - elimNum; i++)
    mpq_init(v[i]);

  int c = 0;
  for(int i = elimNum + 1 ; i < varNum  ; i++)
    mpq_set(v[c++] , a.coef[i]);

  mpq_set(v[varNum - elimNum - 1] , a.constant);
  int r = checkExtreme(pw0, v, varNum - elimNum , sizepw0 , vcheck);
  return(r);
}
```

Listing B.2: Balas check function in unrolled linked list representation

```
int checkExtreme(mpq_t * A, mpq_t * v, int varNum, int ineqNum, int vcheck)
{

  mpq_t * d = (mpq_t *) malloc(ineqNum * sizeof(mpq_t));
  for (int i = 0; i < ineqNum; i++)
  {
    mpq_init(d[i]);
    mpq_set_d(d[i], 21);
  }
  matrixMatrixMult(A, v, d, ineqNum, varNum, 1);

  int e = 0;
  for (int i = 0; i < ineqNum; i++)
    if (mpq_sgn(d[i]) == 0)
      e++;

  mpq_t * w = (mpq_t *) malloc(e * varNum * sizeof(mpq_t));
  for (int i = 0; i < e * varNum; i++)
    mpq_init(w[i]);

  int c = 0;
  for (int i = 0; i < ineqNum; i++)
    if (mpq_sgn(d[i]) == 0)
      for (int j = 0; j < varNum; j++)
        mpq_set(w[c++], A[i * varNum + j]);

  c = 0;
  int r = rankOfMatrix(w, e, varNum);

  if (r == vcheck - 1)
    return (1);
  else
    return (0);
}
```

Listing B.3: Check extreme ray function in unrolled linked list representation

Curriculum Vitae

Name:	Delaram Talaashrafi
Post-Secondary Education and Degrees:	Isfahan University of Technology Isfahan, Iran 2012 - 2017 BSc
	University of Western Ontario London, ON 2017 - 2018 MSc
Related Work Experience:	Teaching Assistant The University of Western Ontario 2017 - 2018

Publications:

Amir Hashemi and Delaram Talaashrafi. A note on dynamic gröbner bases computation. In *International Workshop on Computer Algebra in Scientific Computing*, pages 276-288. Springer, 2016