Universität Passau
Fakultät für Mathematik und Informatik

Volume Calculation and Estimation of Parameterized Integer Polytopes

Diploma Thesis

by
Tilmann Rabl

30. Januar 2006

Supervisor:
Priv. Doz. Dr. Martin Griebl
Co-Supervisor:
Prof. Christian Lengauer, Ph.D.

Abstract

Mathematical models have proved to be useful abstractions in many divisions of computer science. In automatic parallelization the conversion from concrete loop code to an abstract model such as a polytope allows the use of techniques such as linear programming, with the benefits of guaranteed correctness and optimal results. These techniques are, however, often restricted to fixed sized or at most linearly parameterized polytopes. Recent research has overcome this limitation.
We will explain Chernikova's algorithm (a method used to compute the extremal vertices and rays of a polytope defined by a set of inequalities and equations) and introduce an extension to process non-linearly parameterized polytopes. Based on the resulting dual descriptions we will show methods to compute the number of integral points in linearly parameterized polytopes, an algorithm for computing the volume of non-linearly parameterized polytopes and its application as an estimate of the number of integral points in polytopes.

Acknowledgments

I would like to thank the people who made this work possible, first of all my tutor Priv. Doz. Martin Griebl who aroused my interest in loop parallelization, and made many valuable suggestions. He also gave up valuable time in order to proofread this thesis. I would like to thank Armin Größlinger for explaining his fascinating method of quantifier elimination and quick implementations of auxiliary functions. Thanks to all members of the LooPo Team for interesting and helpful discussions. Joachim Hofer also proofread this thesis and I am very grateful for that. Furthermore I have to thank Neil Highnam for straightening out my abuse of the English language. Finally I would like to thank my family, my parents for supporting my career in many different ways and especially for never letting me down and last but not least Maria and Maximilian for giving me love, strength and understanding.

Contents

List of Figures

Chapter 1

Introduction

1.1 Polyhedra and Polytopes in Automatic Parallelization

Single processor solutions reach the limits of possible performance and hardware designers use ever more multi-processor and multi-core systems to overcome this problem. As the number of processing units rises, the demand for exploiting parallelism in software grows. A key concept to producing good parallel software is automatic parallelization. There are various techniques for transforming a sequential to a parallel program. A very effective technique is the parallelization on the polyhedron model. In this model a loop nest is seen as multi-dimensional space and statements within the loops as polyhedra (cf. figure 1.1). Each loop corresponds to one dimension in the model space and the executions of a statement form a polyhedron: all transformations are performed on the polyhedra. The target loop code is generated from the polyhedra. To distribute the program loops effectively over several processors of a parallel computer, cluster or grid efficiently, it is important to know how many executions each statement will have. In the polyhedron model this means counting the number of integral points within a polytope. If the number of iterations is parameterized, the size and shape of the polytope may vary. There are several techniques that take parameters into account (in the constants of the equations and inequalities which describe polyhedra) (e.g. [Cla96], [VSB+04]). For many applications this is sufficient.
However, the parallelism found by the polytope model is very finely grained. Potentially, every operation can be distributed to its own processor. Of course this would be very inefficient, since dependencies between calculations cause communication. In general, this is much more expensive than the operations themselves. A good compromise between distribution and locality is tiling. By tiling we mean the process of grouping adjacent operations in the form of parallelepipeds [Gri00]. For a universal solution we need variable tile sizes, in order not to restrict the code to a certain architecture. We will see in an example in the next section that this introduces parameters in the coefficients of the variables of the equations and inequalities.

```
DO i=0,4
  DO j=0,4
    A[i,j]=...
  END DO
END DO
```

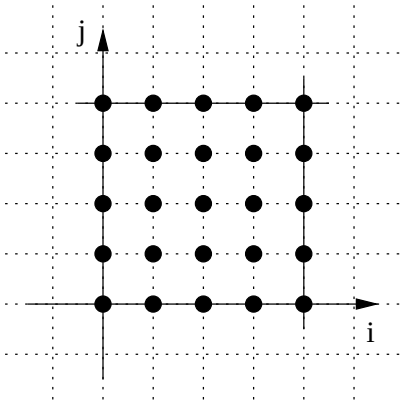


Figure 1.1: Program Code and Corresponding Polytope

Motivated by this setting, we will present an algorithm to compute the vertices, rays and lines of polyhedra with non linear parameters in chapter 3, in chapter 4 we will show existing techniques to compute the number of integral points in linearly parameterized polytopes and finally introduce an estimation for this number based on the volume of polytopes in chapter 5.

1.2 Motivating Example

In the following example we will demonstrate that we have to deal with non-linear expressions even in the simplest form of parameterized tiling. This example can also be found in greater detail in Armin Größlinger's work about non-linear parameterization [Grö03]. Consider the following loop:

```
for i:=0 to n do
    A[i]:=f(i);
od
```

We define that $n \geq 0$ and f has no side effects. For each $i \in \{0, \ldots, n\}$ the loop evaluates a function call $f(i)$ and then assigns its value to an array. If we now want to distribute the program to a parallel computer with $p \geq 1$ processors, we partition the array in tiles of size p and allocate one operation per processor per part. This is a common round robin or cyclic distribution, it is shown in figure 1.2, where t denotes the tile number and o the processor number. We can describe this tiling by the following inequality system:

$$\begin{aligned} 0 \leq \;& i \;\leq n \\ 0 \leq \;& o \;\leq p-1 \\ & i \;= p \cdot t + o \end{aligned} \tag{1.1}$$

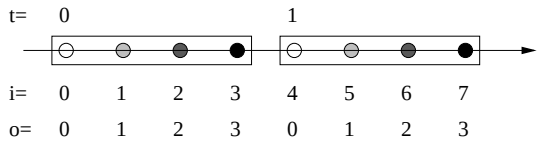


Figure 1.2: Tiling of a One-Dimensional Loop.

We are now interested in the number of tiles T to estimate the work each processor has to do. We can do this by eliminating the variables o and i from the system. Both have only constant coefficients, so we won't come across any case distinctions. First we substitute o by $i - p \cdot t$:

$$\begin{array}{rcl} 0 \leq & i & \leq n \\ 0 \leq & i - p \cdot t & \leq p - 1 \end{array} \tag{1.2}$$

Then we eliminate i by solving the inequalities for i

$$\begin{array}{rcl} 0 \leq & i & \\ p \cdot t \leq & i & \\ & i & \leq n \\ & i & \leq p \cdot t + p - 1 \end{array} \tag{1.3}$$

and then comparing i's lower and upper bounds

$$\begin{array}{rcl} 0 & \leq & n \\ 0 & \leq & p \cdot t + p - 1 \\ p \cdot t & \leq & n \\ p \cdot t & \leq & p \cdot t + p - 1 \end{array} \tag{1.4}$$

We predefined $n \geq 0$ and $p \geq 1$ and hence the inequalities $0 \leq n$ and $p \cdot t \leq p \cdot t + p - 1$ are redundant and can be omitted. We can now solve the remaining inequalities for t, since we assumed $p \geq 1$ we get one distinct system:

$$\frac{1}{p} - 1 \leq t \leq \frac{n}{p} \tag{1.5}$$

The example is very simple and the solution is obvious,

$$T = \left| \left\{ t \left| \frac{1}{p} - 1 \leq t \leq \frac{n}{p} \right. \right\} \right| = \left\lceil \frac{n}{p} \right\rceil . \tag{1.6}$$

However, the inequality system is not linear, since it contains fractions. It shows that non-linear inequalities actually appear even in simple tasks.

Chapter 2

Prerequisites

2.1 Polyhedra and Polytopes

Definition 2.1 (Linear equation) A linear equation is an equation of the form

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = b, \tag{2.1}$$

where $x_1, \ldots, x_n$ are the variables, $a_1, \ldots, a_n$ the coefficients and b the additive constant. In the following, three types of linear equations are differentiated:
Equations with

- real coefficients (e.g. $3x_1 + 3.14x_2 = 5$).
- linearly parameterized coefficients (e.g. $2x_1 - 27x_2 = 3.5b$),
  where b is a symbolic constant.
- non-linearly parameterized coefficients (e.g. $13a_1x_1 - a_2x_2 = 3.3b$), where a_1, a_2 and b are symbolic constants.

An equation is in *homogeneous form*, if its right side is equal to zero,

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n - b = 0. \tag{2.2}$$

Definition 2.2 (Linear inequality) Similar to linear equations, linear inequalities are inequalities of the form

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n \geq b \tag{2.3}$$

with the same properties and differentiations as in definition 2.1.

Notation 2.1 An equation can be represented by two inequalities:

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = b \iff \begin{array}{l} a_1x_1 + a_2x_2 + \ldots + a_nx_n \geq b \\ a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq b \end{array} \tag{2.4}$$

Notation 2.2 A system of m inequalities in homogeneous form can be written as an $m \times (n+1)$ matrix A,

$$A * v \geq 0, \tag{2.5}$$

where A may contain symbolic constants. Vector v holds n variables and the additional constant,

$$v = \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{pmatrix}. \tag{2.6}$$

Definition 2.3 (Hyperplane) A hyperplane is an affine subspace of codimension 1, i.e. a $d-1$-dimensional subspace in a d-dimensional space. It can be described by one affine equation.

Definition 2.4 (Half-space) A $n-1$-dimensional hyperplane divides a n-dimensional space into two half-spaces, therefore a half-space can be represented by one affine inequality.

Definition 2.5 (Polyhedron) A polyhedron is a finite intersection of half-spaces. Thus it can be represented by a finite set of inequalities, i.e. a system of inequalities. This is called *implicit representation*. We will see an other representation below (def. 2.10).

Definition 2.6 (Polytope) A polytope is a bounded polyhedron.
The hyperplanes that bound the polytope are called *supporting hyperplanes*.

Definition 2.7 (Line) A line is a one-dimensional space. It is infinitively long and can be represented by one affine equation. Lines are sometimes also referred to as bidirectional rays.

Definition 2.8 (Ray) A ray is a half-line, it has an origin and is infinitively long in one direction. It can be represented by one affine inequality.

Definition 2.9 (Polyhedral cone) A cone is a set C with the following property

$$x \in C, a \in \mathbb{R}^+ \Rightarrow ax \in C. \tag{2.7}$$

A cone C is *convex* if

$$x, y \in C, a, b \in \mathbb{R}^+ \Rightarrow ax + by \in C. \tag{2.8}$$

A cone C is *polyhedral* if there exits a matrix A with

$$C = \{x | Ax \geq 0\}. \tag{2.9}$$

A polyhedral cone is a special form of a polyhedron with only one vertex, the origin. The representation in 2.9 is called *implicit representation*. Below polyhedral cones will simply be referred as cones.
There is another representation, which is dual to the above (cf. [Sch86]), the *parametric representation*. A cone can be represented by its lines and rays.

$$C = \{x | \sum_{i=1}^{n} x_i r_i + \sum_{j=1}^{m} x_j l_j \wedge (\forall i : x_i \geq 0)\} \tag{2.10}$$

where $\{r_1 \dots r_n\}$ is the set of extremal rays and $\{l_1 \dots r_m\}$ the set of extremal lines of C. It is possible to convert one representation to the other. This principle is called *cone duality*
A cone C is *pointed* if its set of lines is empty.

Definition 2.10 (Polyhedron - parametric representation) Motzkin showed that a polyhedron P can be uniquely decomposed in a polytope V and a characteristic cone C [MRTT53],

$$P = C + V. \tag{2.11}$$

As described in equation 2.10 a cone can be decomposed in its extremal rays and its lines. Combined with equation 2.11 this results in another representation for polyhedra

$$P = L + R + V \tag{2.12}$$

where L is P's *lineality-space* generated by the set of lines, R is a pointed cone generated by the extremal rays and V the polytope, which is the convex hull of P's vertices. This representation is again dual to definition 2.5 (cf. [MRTT53]), analogue to cones it is called *parametric representation*.
A polyhedron of dimension n is called *degenerate* if it contains an extremal vertex that is the intersection of more than n half-spaces.

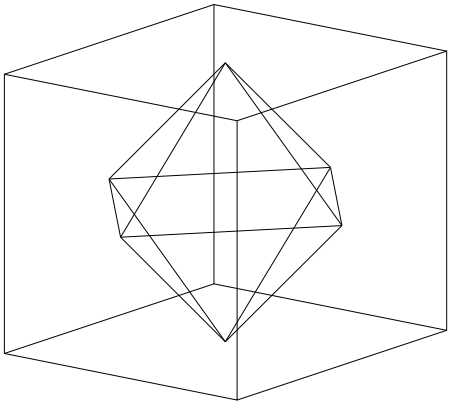


Figure 2.1: Cube and its Dual Octahedron

Definition 2.11 (Dual polyhedron) Every convex polyhedron has a dual. It is a convex polyhedron whose vertices and facets have complementary positions to the original polytope's, i.e. vertices of the dual polyhedron correspond to the facets of the original and vice versa. The dual of the dual is the original polyhedron. Accordingly, the number of vertices of the dual polyhedron is equal to the number of faces of the original plus the number of faces the number of vertices.
An example is the duality between cube and octahedron, which is shown in figure 2.1[1].

[1]The octahedron is scaled down in the figure to illustrate the relative position of the vertices

Chapter 3

Chernikova's Algorithm

As explained in definition 2.5, polyhedra, and polytopes respectively, have two, dual representations. A polytope can be described as the intersection of a finite set of half-spaces or as combination of its vertices, rays and lines. To transform one form into the other Chernikova's algorithm is often used. Although there are other algorithms for the same problem (see [MR80]), this one is the common choice, for it can also process polytopes with vertices with negative coordinates without encoding them in higher dimensional space.
The first part of this chapter explains how to compute the dual polytope representation, section 3.2 the principle of Chernikova's algorithm [Che64, Che65, Che68]. Section 3.3 shows the improvements made by H. Le Verge [Ver94] and the last section 3.4 discusses an extension which allows the computation of polytopes with non-linear parameters.

3.1 How to Compute the Dual Representation

The basic technique used to compute the vertex representation from the half-space representation is to find the extremal solutions of the system of inequalities containing the half-space descriptions. So, for a polytope, the following procedure could be used. The vertices of a polytope always lie on the hyperplanes that border the defining half-spaces. If the considered polytope is n-dimensional, then at least n hyperplanes meet at a vertex. So a straightforward approach would be to try out all n-combinations of the supporting hyperplanes, to calculate the points of intersection and to discard the outlying, redundant and non-extremal points.
To get the half-space representation from the vertices the same procedure can be used, because of the duality of both representations (see definition 2.11). The resulting algorithm would of course be much too expensive, so more sophisticated solutions have to be used.

3.2 Function of Chernikova's Algorithm

Chernikova's Algorithm was originally designed to compute the non-negative solutions of a system of linear inequalities without parameters [Che64, Che65, Che68]. But it can easily be enhanced for parameterized systems as shown in section 3.4.
The technique used by this algorithm was reinvented several times. It is originally known as the double description method and was introduced by Motzkin et al. [MRTT53]. Nevertheless we will refer to it as Chernikova's algorithm, for the enhancements shown later (section 3.3) base on Chernikova's reinvention. In the following we will give an informal explanation of the algorithm and later the formal procedure as explained in the original paper [Che68].

3.2.1 Informal Description

Because the algorithm is not very intuitive, we will present a more easily understandable explanation first. We will only show the computation of the vertex representation from the half-space representation, the reverse calculation will be explained later in section 3.2.2. Summarized briefly, the algorithm consists of the following steps:

Initialization the set of vectors, that represent the vertices, is initialized

Iteration we iterate over the given set of constraints, at each iterative step we

- divide the set of vertices into three sets which hold vectors that saturate, verify and violate the current constraint,
- combine violating and verifying vectors into new saturating vectors,
- discard redundant vectors from the combinations,
- union saturating, verifying and new found vectors as new vertex set

Finalization the set of vertices is transfered to the desired representation

We will explain the single steps in detail in the following.

Initialization The input of our algorithm is an inequality system with n variables, for example

$$\begin{aligned} x_1 &\geq 1 \\ x_2 &\geq 1 \\ x_1 &\leq 5 \\ x_2 &\leq 4 \\ x_1 - x_2 &\geq -1 \end{aligned} \tag{3.1}$$

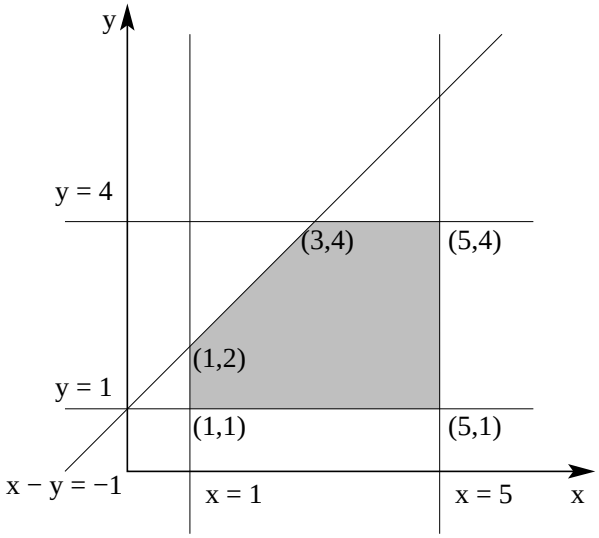


Figure 3.1: 2-Dimensional Polytope Defined by 5 Hyperplanes

The polytope defined by this system is shown in figure 3.1. For calculation and simplification purposes we will use a matrix A, where each row vector is one inequality of the system (cf. notation 2.2). To do so we have to transform the inequalities in a homogeneous form (cf. definition 2.1). We will refer to these inequalities as constraints.

$$A = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ -1 & 0 & 5 \\ 0 & -1 & 4 \\ 1 & -1 & 1 \end{pmatrix} \tag{3.2}$$

The algorithm works on a set of vectors R. Its members are incrementally transformed and combined, until they represent the vertices and rays of the polytope. We need a set of vectors that spans the whole space. A possible initial set contains the unit vectors. Again we will write it as matrix,

$$R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{3.3}$$

The vectors can be interpreted in the following way: the first two unit vectors are rays, where the first points in x direction and the second in y direction. The third vector is a vertex at the origin. So our initial set is a cone that spans the first quadrant.

Iteration Now the iterative part begins. For each constraint in A we do the following:

Vertex Set Splitting We calculate the scalar product of the constraint and each vector and split the set according to the result in three groups:

- The set of verifying vectors R^+ which holds the vectors that have a positive result,
- the set of saturating vectors R^0, where the result of the vectors is 0,
- the set of violating vectors R^-, with negative results.

For the first constraint $(1\ 0\ \text{-}1)$ this would be the as follows

$$R^+ = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix},\ R^0 = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix},\ R^- = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \tag{3.4}$$

These three sets can be interpreted as follows:

- R^+ contains the vectors that point into the half-space defined by the constraint,
- R^0 contains the vectors that lie parallel to the border of the half-space,
- R^- contains the vectors that point out of the half-space.

Linear Combination Since the vectors in R^+ point into the half-space and the vectors in R^- point outwards, we can form linear combinations of the elements that lie on the border to the half-space. To build all pairs we simply form the Cartesian product of the two sets. For each pair $(v_1, v_2) \in R^+ \times R^-$ we build a linear combination $(av_1 + bv_2)$ with positive coefficients a, b, that saturates the current constraint c

$$(av_1 + bv_2) \bullet c = 0 \tag{3.5}$$

Where $x \bullet y$ denotes the inner product of the vectors x and y. The linear combination can easily be found by this formula

$$(v_1 \bullet c) * v_2 - (v_2 \bullet c) * v_1 \tag{3.6}$$

Since the coefficients may grow significantly we normalize the vectors, that means, we divide the coefficients by their greatest common divisor. In our example the Cartesian product gives us just one pair

$$R^+ \times R^- = \left\{ \left(\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \right) \right\} \tag{3.7}$$

The linear combination of this pair is

$$\begin{aligned} &\left(\begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \bullet \begin{pmatrix} 1 & 0 & -1 \end{pmatrix} \right) * \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \\ -&\left(\begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \bullet \begin{pmatrix} 1 & 0 & -1 \end{pmatrix} \right) * \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 1 \end{pmatrix} \end{aligned} \tag{3.8}$$

The normalization does not change the vector.

Redundancy Check Some of the newly generated vectors may now be redundant. So we have to test the vectors and keep only the valid ones. This can be done by deleting the members of the linear combination from the set of vectors and checking if there is still a vector which saturates the same, and maybe some more, already processed constraints. If there is, the vector is redundant, or else it is new. The new vectors are stored in the set $\overline{R}$
In the first iteration there is, of course, no processed constraint and hence the generated vectors are never redundant. To see a more expressive example, we jump to the last, i.e. the fifth, iteration:

- The current constraint is

$$c = \begin{pmatrix} 1 & -1 & 1 \end{pmatrix}, \tag{3.9}$$

- the set of processed constraints A^{old} is

$$A^{old} = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ -1 & 0 & 5 \\ 0 & -1 & 4 \end{pmatrix}, \tag{3.10}$$

- the sets of verifying, saturating and violating vectors are

$$R^+ = \begin{pmatrix} 1 & 1 & 1 \\ 5 & 1 & 1 \\ 5 & 4 & 1 \end{pmatrix},\ R^0 = \emptyset,\ R^- = \begin{pmatrix} 1 & 4 & 1 \end{pmatrix} \tag{3.11}$$

- The set of pairs and their linear combinations is

$$\left(\begin{array}{ccc|ccc|ccc} \multicolumn{3}{c|}{R^+} & \multicolumn{3}{c|}{R^-} & \multicolumn{3}{c}{R^{new}} \\ 1 & 1 & 1 & 1 & 4 & 1 & 1 & 2 & 1 \\ 5 & 1 & 1 & 1 & 4 & 1 & 15 & 22 & 7 \\ 5 & 4 & 1 & 1 & 4 & 1 & 3 & 4 & 1 \end{array}\right) \tag{3.12}$$

At this point we have to test if the new vectors are redundant. For each new vector $(av_1 + bv_2) \in R^{new}$ we calculate the set of saturated, processed constraints and verify that no other vector $v \in R \setminus \{v_1, v_2\}$ saturates a set which contains this set, where $R = R^+ \cup R^0 \cup R^-$. We do this here only for the vectors $(15\ 22\ 7)$ and $(1\ 2\ 1)$

- the set of saturated, processed constraints for $(15\ 22\ 7)$ is $\emptyset$, this means the vector is not extremal and hence is not a valid new vector. This happens, when the sets of saturated constraints of the combined vectors are disjoint.

- $(1\ 2\ 1)$ saturates only one old constraint

$$\begin{pmatrix} 1 & 0 & -1 \end{pmatrix} \tag{3.13}$$

since neither $(5\ 1\ 1)$ nor $(5\ 4\ 1)$ saturate the same constraint, this vector is new.

Union As a last step we have to combine the sets of vectors that we want to keep. The vectors we need are

- the saturating vectors R^0,
- the verifying vectors R^+,
- the new, irredundant vectors $\overline{R}$.

So our new set of vectors is $R^0 \cup R^+ \cup \overline{R}$.

Finalization After we have processed all constraints the set of vectors should be exactly the set of vertices of the polytope. Each vector can be read as Cartesian coordinates

$$\begin{pmatrix} a_1 & \dots & a_n & b \end{pmatrix} \Longrightarrow \left(\frac{a_1}{b}, \dots, \frac{a_n}{b}\right) \tag{3.14}$$

In our example we have

$$\begin{pmatrix} 1 & 1 & 1 \\ 5 & 1 & 1 \\ 5 & 4 & 1 \\ 1 & 2 & 1 \\ 3 & 4 & 1 \end{pmatrix} \Longrightarrow \begin{matrix} (1,1) \\ (5,1) \\ (5,4) \\ (1,2) \\ (3,4) \end{matrix} \tag{3.15}$$

which can be verified in figure 3.1.

3.2.2 Formal Description

Now we want to describe the algorithm in the form given in Chernikova's original paper [Che68]. For an example see section 3.3.2. Again the algorithm consist of three parts, initialization, iteration and finalization.

Initialization The input of the algorithm is a system of inequalities of the form

$$\begin{array}{l} a_{11}x_1 + \ldots + a_{1n}x_n \geq b_1 \\ \ldots \\ a_{m1}x_1 + \ldots + a_{mn}x_n \geq b_m \\ x_i \geq 0, i \in \{1, \ldots, n\} \end{array} . \tag{3.16}$$

If in one of the inequalities all the coefficients of the x_i are non-positive and the free term is positive then we are finished, because the system is inconsistent and hence unsolvable. If there is no such inequality a matrix of the following form is constructed

$$\left(\begin{array}{ccccc|ccc} 1 & 0 & \ldots & 0 & 0 & a_{11} & \ldots & a_{m1} \\ & & \vdots & & & & \vdots & \\ 0 & 0 & \ldots & 1 & 0 & a_{1n} & \ldots & a_{mn} \\ 0 & 0 & \ldots & 0 & 1 & -b_1 & \ldots & -b_m \end{array}\right) . \tag{3.17}$$

On the right side of the matrix the constraints are inserted column wise and the left side is an $n+1$-dimensional unit matrix, which represents the ray-space.

Iteration From this matrix the algorithm works incrementally on the constraint columns. At each step we select one of the columns with at least one negative element as reference and transform the matrix according to the following steps:

- we build all pairs of rows which have opposite signs at the reference column.
- for each pair we check if there is a column of the left side or an already processed column of the right side, where both rows have zeros,
- if there is, we form a linear combination with positive coefficients, such that it has a zero at the reference column, else we omit the pair,
- to check if the linear combination is redundant, we check if there is no other verifying or saturating vector, that saturates the same constraints, if there is none we have a new vector if not, we omit the combination.

The new matrix contains all new linear combinations and all rows that were non-negative at the reference column.

Finalization After at most $m+1$ steps the matrix has either:

1. one negative column on the right side;
2. only non-negative columns at the right side and the last column at the left side is zero;

3. only non-negative columns at the right side and the last column at the left side is non-zero.

The first two cases indicate that the inequality system is insolvable, the third gives the desired result. In this case the left side of the matrix gives us the vertices of the polytope row-wise. A row can be interpreted as follows:

$$(d_{i1}, \ldots, d_{in}, d_{i,n+1}, a'_{i1}, \ldots, a'_{im}) \Rightarrow \left(\frac{d_{i1}}{d_{i,n+1}}, \ldots, \frac{d_{in}}{d_{i,n+1}} \right). \tag{3.18}$$

If the entry at position $n+1$ on the left side is equal to zero, the polytope is unbounded and thus no polytope, but a polyhedron. Since we are only interested in polytopes we exclude this possibility.

Computing the Inequalities

To do the reverse computation we simply build the inequality system of the dual polytope. This polytope's supporting hyperplanes have the same equations as the vertices of the original polytope and vice versa. So a vertex of a polytope would be transformed to an inequality as follows:

$$\left(\frac{a_1}{1}, \ldots, \frac{a_n}{1} \right) \Rightarrow (a_1 x_1 + \ldots + a_n x_n \geq 1). \tag{3.19}$$

3.3 H. Le Verge's Improvements

The algorithm as proposed in 3.2.2 has some flaws which produce many redundant and unnecessary computations. Herve Le Verge presented some effective enhancements [Ver94]. He gathered some new criteria that filter out redundant linear combinations earlier and he introduced the computation of negative solutions. His algorithm was integrated in the PolyLib library where it is improved continuously [Wil93].
In this section we will first explain ways to reduce costs of computation. In section 3.3.2 we will show how to additionally find negative solutions.

3.3.1 New Criteria

A bad influence on the algorithms efficiency is the effect of redundant vectors. The way they are dealt with reveals much of the algorithm's performance. The most primitive form to handle redundant vectors is to remove them at the end of the algorithm, which results in an explosion of vector combinations in the iteration. It is already a huge advantage to remove them at each iteration. A straightforward way to do so is described in section 3.2.1. At each step we have to check all rays of V for each element

of $V^+ \times V^-$, where V grows in general exponentially.
An obvious idea to reduce computational costs further is to prevent the occurrence of redundant vectors. After each iterative step of the algorithm there are no redundant vectors, so they can only be generated by a linear combination. The challenge is to identify properties of the combined rays that indicate that the new vector will be redundant.

Chernikova's Criterion The original algorithm already uses a rule, to ensure that a new ray will be an extremal ray of the polytope [Che65]. If the sets of saturated constraints $S(y_1), S(y_2)$ of the two combined rays y_1, y_2 are disjoint the new ray will not belong to the boundary of the convex cone and hence not create an extremal vertex. If $|S(y_1) \cap S(y_2)| \geq 1$ holds the linear combination will be generated and we have to prove that there is no other ray that saturates the same constraints, that means there is no ray $y_i \in V/\{y_1, y_2\}$ such that $S(y_1) \cap S(y_2) \subseteq S(y_i)$.

New Criterion Chernikova's criterion was extended, so that far more combinations can be rejected, without calculation. Le Verge showed that in an n-dimensional space only combinations that saturate $n-2$ constraints are extremal [Ver94]. By this criterion in general far more rays are omitted than by the first criterion.

3.3.2 Finding General Solutions

Unlike algorithms basing on the simplex method, Chernikova's algorithm does not need to encode polytopes in higher dimensional space to find non-negative and negative solutions. We only have to differentiate between unidirectional and bidirectional rays, i.e. lines. We will first explain the changes and then illustrate them with an example, which can be seen in figure 3.2. We use a set of lines L to span the initial space, which we restrict more and more by iteratively processing the constraints. But instead of one distinct iterative step we have two cases, depending on the current set of lines. The initial set L_0 is the set of basis vectors and the initial set of rays $V = \emptyset$. At each step we choose one way of proceeding:

- If there is a line l_k which does not saturate the current constraint c, that means $c * l_k \neq 0$,
  - we project all l_i on the defining hyperplane of c along the positive part l'_k

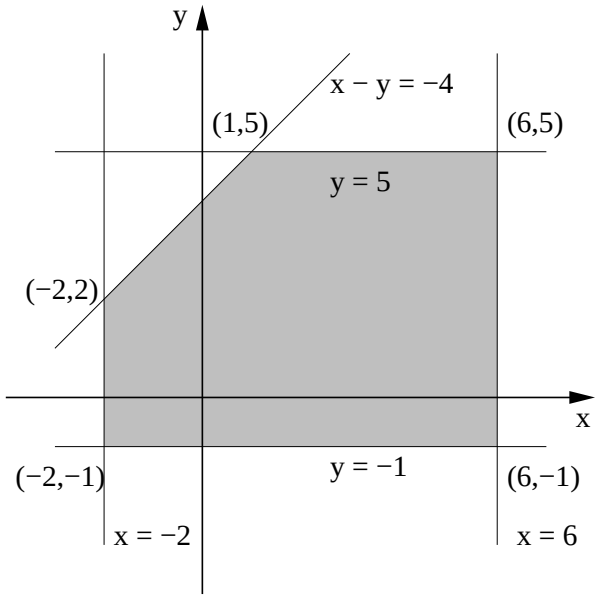


Figure 3.2: 2-Dimensional Polytope with Negative Vertices

of line l_k

$$\begin{aligned} l'_k &= \pm l_k \\ c * l'_k &> 0 \\ l'_i &= \lambda l_i + \mu l'_k, \forall i \neq k \\ c * l'_i &= 0, \forall i \neq k \\ L' &= \{z'_1, \ldots, z'_{k-1}, z'_{k+1}, \ldots, z'_m\} \end{aligned} \tag{3.20}$$

– we do the same with all rays v_j and add the positive part l'_k to the set of rays:

$$\begin{aligned} v'_i &= \lambda v_i + \mu l'_k, \lambda > 0 \\ c * v'_i &= 0 \\ V' &= \{v'_1, \ldots, v'_r\} \cup \{l'_j\} \end{aligned} \tag{3.21}$$

- if all lines are such that $c * l_k = 0$, E is not changed and we do the same transformations of V as in the original algorithm, but with the new criterion. A linear combination can be avoided if the set of common saturated constraints contains at most $n - m - 3$ constraints, where n is the dimension of the space and m is the number of lines.

Now we will consider the example in figure 3.2. The polytope shown has the implicit representation:

$$A = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 1 \\ -1 & 0 & 6 \\ 0 & 1 & 5 \\ 1 & -1 & 4 \end{pmatrix}. \tag{3.22}$$

In the following we will refer to the lines and rays by their row numbers, i.e. line l_1 is the line in row 1 in the lineality-space matrix. As a first step, we initialize our matrix for the lineality-space and the ray-space:

$$L = \left(\begin{array}{ccc|ccccc} 1 & 0 & 0 & 1 & 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & -1 & -1 \\ 0 & 0 & 1 & 2 & 1 & 6 & 5 & 4 \end{array}\right) \qquad V = () \tag{3.23}$$

The matrix for the lineality-space also contains all of the constraints as columns. We will now process the constraints one by one in the order above. We will not use the original constraints below, since we extended the vectors in L with the according dimension of the constraints. If we want to test if a vector satisfies the ith constraint, we only have to check if the $3 + i$th column of the vector is equal to zero. This is valid because we use all transformations on the complete vector and so the relationship between the vector and all constraints is always correct. As we can see the first constraint $(1\ 0\ 2)$ is not saturated by l_1 and l_3, so we have the first case. We pick l_1 and build a linear combination with the other lines so that all saturate the constraint and add its positive part to the rays. Since l_2 saturates the constraint we only have to combine l_1 and l_3, as coefficients we choose the entries in the constraint column, so the new line will result from $l_3 - 2l_1$. We don't have to combine the rays since there are none yet. As new lineality- and ray-spaces we get:

$$L = \left(\begin{array}{ccc|ccccc} 0 & 1 & 0 & 0 & 1 & 0 & -1 & -1 \\ -2 & 0 & 1 & 0 & 1 & 8 & 5 & 2 \end{array}\right) \quad V = \left(\begin{array}{ccc|ccccc} 1 & 0 & 0 & 1 & 0 & -1 & 0 & 1 \end{array}\right) \tag{3.24}$$

The next constraint is not saturated by l_1 and l_2, we pick l_1, combine it with l_2 and add it to the rays. r_1 saturates this constraint, so we don't have to build a linear combination here.

$$L = \left(\begin{array}{ccc|ccccc} -2 & -1 & 1 & 0 & 0 & 8 & 6 & 3 \end{array}\right) \quad V = \left(\begin{array}{ccc|ccccc} 1 & 0 & 0 & 1 & 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & -1 & -1 \end{array}\right) \tag{3.25}$$

The third constraint is again not saturated by l_1, but since no other line is left, we only have to build combinations of rays. We combine $8r_1 + l_1$. r_2 saturates the constraint

and, therefore, doesn't need to be changed.

$$L = () \qquad V = \left(\begin{array}{rrr|rrrrr} 0 & 1 & 0 & 0 & 1 & 0 & -1 & -1 \\ 6 & -1 & 1 & 8 & 0 & 0 & 6 & 11 \\ -2 & -1 & 1 & 0 & 0 & 8 & 6 & 3 \end{array}\right) \tag{3.26}$$

Now we have no more lines left and so are always in the second case, in which only the rays are processed. The forth constraint is verified by r_2 and r_3, while r_1 violates the constraint. So we build the following linear combinations, $r_2 + 6r_1$ and $r_3 + 6r_1$.

$$L = () \qquad V = \left(\begin{array}{rrr|rrrrr} 6 & -1 & 1 & 8 & 0 & 0 & 6 & 11 \\ -2 & -1 & 1 & 0 & 0 & 8 & 6 & 3 \\ -2 & 5 & 1 & 0 & 6 & 8 & 0 & -3 \\ 6 & 5 & 1 & 8 & 6 & 0 & 0 & 5 \end{array}\right) \tag{3.27}$$

The last constraint is saturated by r_1, r_2 and r_4 and violated by r_3. This time we can omit one combination, r_1 and r_3 have no common saturated constraint and, therefore, their linear combination will not produce an extremal ray. We build the combinations, $r_2 + r_3$ and $3r_4 + 5r_3$.

$$L = () \qquad V = \left(\begin{array}{rrr|rrrrr} 6 & -1 & 1 & 8 & 0 & 0 & 6 & 11 \\ -2 & -1 & 1 & 0 & 0 & 8 & 6 & 3 \\ 6 & 5 & 1 & 8 & 6 & 0 & 0 & 5 \\ -4 & 4 & 2 & 0 & 6 & 16 & 6 & 0 \\ 8 & 40 & 8 & 24 & 48 & 40 & 0 & 0 \end{array}\right) \tag{3.28}$$

After normalizing we get the reduced ray-space

$$V^* = \begin{pmatrix} 6 & -1 & 1 \\ -2 & -1 & 1 \\ 6 & 5 & 1 \\ -2 & 2 & 1 \\ 1 & 5 & 1 \end{pmatrix} \Longleftrightarrow \begin{matrix} (6,-1) \\ (-2,-1) \\ (6,5) \\ (-2,2) \\ (1,5) \end{matrix}, \tag{3.29}$$

which can be verified in figure 3.2.

3.4 Parameterized Polytopes

The algorithms in the preceding sections only covered polytopes of fixed sizes, but since mathematical and computational problems are often parameterized we need to extend the algorithm so that it can cope with these.
We will show how to process two levels of parameterization, first linear and later non-linear parameters.

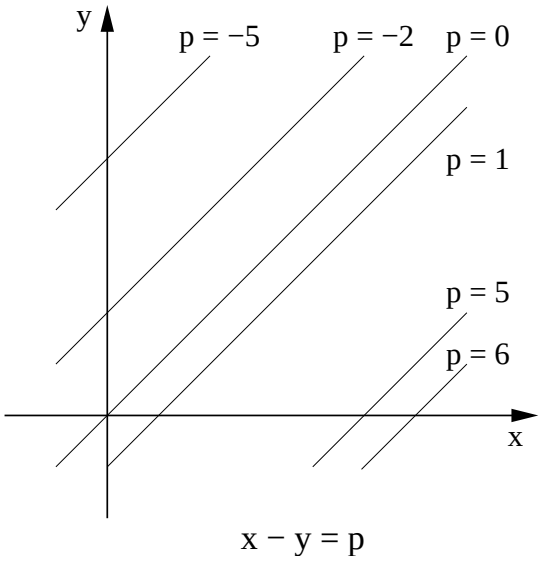


Figure 3.3: Parameterized Hyperplane

3.4.1 Linearly Parameterized Polytopes

A linearly parameterized polytope can be described by an inequality system as follows

$$Ax + Bn \geq b \tag{3.30}$$

where x is the vector of variables, n the vector of parameters and b the constant vector. As we don't know about the values of the parameters, we can simply also treat them as variables. This leads to a simpler representation

$$Ax \geq b, \tag{3.31}$$

which is the same as for non-parameterized polytopes. And, once again, we can build a homogeneous form by encoding the constant vector in matrix A.
The following example explains this, we will use a similar polytope to that in section 3.2:

$$\begin{aligned} x &\geq 1 \\ y &\geq 1 \\ x &\leq 5 \\ y &\leq 4 \\ x - y &\geq p \end{aligned} \tag{3.32}$$

This polytope has one parameterized inequality, $x - y \leq p$. The border of the corresponding half-space depends on the value of p. Figure 3.3 shows the bounding hyperplane for some values. Depending on the value of the parameter, the polytope may

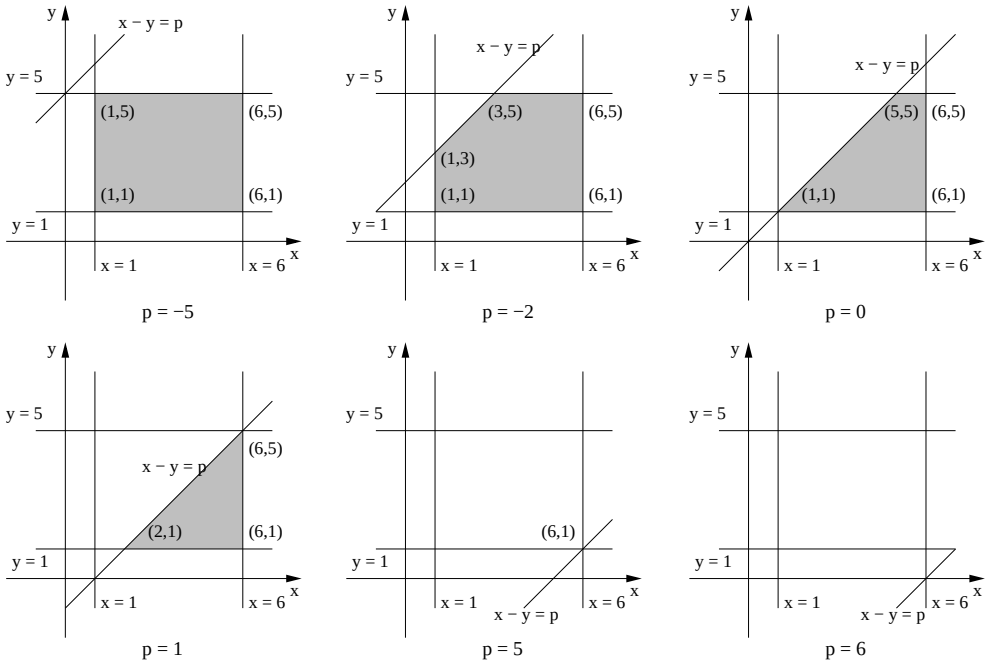


Figure 3.4: Different Shapes due to Different Parameter Values

have different shapes (cf. figure 3.4). This is no problem for the algorithm; as we treat the parameters like variables, the algorithm itself does not change. The input matrix for the inequality system above would look as follows:

$$A = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 0 & 6 \\ 0 & -1 & 0 & 5 \\ 1 & -1 & -1 & 0 \end{pmatrix} \tag{3.33}$$

The corresponding output is:

$$R = \begin{pmatrix} & 0 & 0 & -1 & 0 \\ & 6 & 1 & 5 & 1 \\ more & 1 & 1 & 0 & 1 \\ & 1 & 5 & -4 & 1 \\ & 6 & 5 & 1 & 1 \end{pmatrix} \tag{3.34}$$

The computed lineality- and ray-spaces are independent of the parameters, since a linear parameter may shift a ray or a line, but it will never change its direction (cf. [LW97]). As a result, they need no further processing. Vertices, on the other hand, may only be defined over certain domains. Consider the vertex $(1, 5)$ of the polytope in equation 3.32, as shown in figure 3.4. It is only existent for $p \geq 4$.

Computing the Vertices

As mentioned before, vertices that are dependent on parameters $v_i(p)$ are only defined for fixed domains, thus we need a representation that takes this fact into account. We will use a tuple of a vector and a domain $[v_i, dom_i]$. The definition in homogeneous space is

$$v(p) = \begin{cases} v & \text{if } p \in dom \\ 0 & \text{else} \end{cases} \tag{3.35}$$

To compute the set of parameter dependent vertices $V(p)$ we need the implicit and the parametric representation. Then we do the following steps:

- We compute the m-faces of the homogeneous polytope, where m is the number of parameters. A m-face is a m-dimensional facet of a polytope. Each m-face F_i^m is represented by a set of rays R_i and lines L and can be written as a matrix $M_i = [L|R_i]$. The m-face algorithm is shown below.
- For each m-face we build a projection to the parameters $Proj_p(M_i)$, which is simply deleting the rows of M_i which correspond to variable coefficients. We try to compute its right inverse $Proj_p(M_i)^{-R}$. If this is not possible the m-face does not describe a single vertex. If it is possible we also build the projection to the variables $Proj_d(M_i)$ and compute the coordinate describing matrix $T_i = Proj_d(M_i)Proj_p(M_i)^{-R}$. T_i is the homogeneous matrix representation of v_i, it has $m + 1$ columns and $n + 1$ rows, where n is the dimensionality of the polytope.
- We compute the validity domain for each m-face by projecting it to the parameter space $Proj_p(F_i^m)$. Again $Proj_p(F_i^m)$ is the homogeneous matrix representation of dom_i.
- We combine the coordinate vector and domain $[v_i, dom_i]$.

m-Face Algorithm To find m-dimensional faces of a polytope of dimension d we process the following steps (cf. [LW97]):

1. We build all combinations of $d - m$ constraints. If there are equations among the constraints, they are always included.
2. If we don't already have, we build a saturation matrix (see below) and intersect the rows that correspond to the constraints to be combined.
3. We count the number of "ones" in the intersections, which is the number of saturated vertices, rays and lines. If it is greater than m we have a non-empty m-face.

4. To eliminate redundant faces, we check if any other face saturates the same set of vertices, rays and lines. If so, the face is redundant.

Saturation Matrix In order to compute only the set of saturated constraints of a vector once, we can store the result. To keep things simple we use a boolean matrix, which has a row for every constraint and a column for every vector. We call this matrix saturation or incidence matrix. The elements $x_{i,j}$ of the matrix are defined as follows:

$$x_{i,j} = \begin{cases} True, & \text{if constraint } i \text{ is saturated by vector } j \\ False, & \text{constraint } i \text{ is verified by vector } j \end{cases} \tag{3.36}$$

The saturation matrix does not have to be computed in a separate step, it can also be a part of the result of Chernikova's algorithm, since the matrix it works on contains essentially this data (see equation 3.16). For efficiency reasons we can additionally store the transpose of this matrix, as done in the Parma Polyhedra Library [BHRZ03]. Consider the polytope P in figure 3.4, its implicit and parametric representation can be seen in equation 3.33 and 3.34. In order to compute the vertices, we first have to find the 1-faces of P, since the number of parameters is 1. To do so we need the incidence matrix I,

$$I = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{3.37}$$

To find the 1-faces, we build all possible combinations of $d - m$ constraints, where d is the dimension of the polytope in homogeneous form, i.e. 3. So we build all combinations of 2 constraints and intersect their rows in the incidence matrix. When the constraints have more than m rays, lines or vertices in common, the m-face describes a single vertex. We will do this for (c_1, c_2) and (c_1, c_3), where c_i is the constraint in row i. The first row and the second row of the incidence matrix have the first and the third vertex/ray in common. So we have a valid face with the matrix representation

$$M_1 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ -1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{3.38}$$

(c_1, c_3) have only the first ray/vertex in common and, therefore, do not generate a single vertex. When we do this for all combinations, we get 8 1-faces. For each face we compute the projection to the parameters $Proj_p(M_i)$ and the right inverse of that $Proj_p(M_i)^{-R}$ and the projection to the domain $Proj_d(M_i)$. For M_1 we get

$$Proj_p(M_1) = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} = Proj_p(M_1)^{-R}, \quad Proj_d(M_1) = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}. \tag{3.39}$$

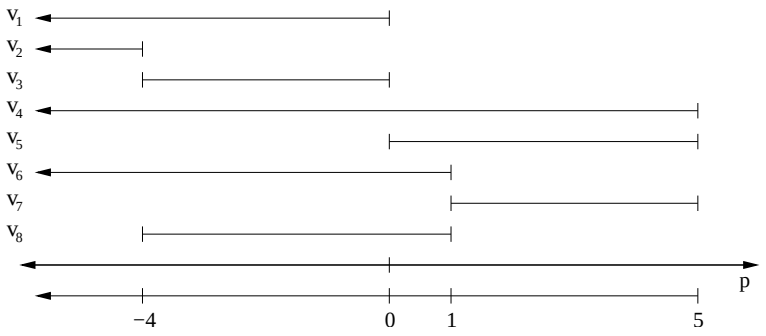


Figure 3.5: Domain Decomposition

Now we can compute $T_1 = Proj_d(M_1) \times Proj_p(M_1)^{-R}$

$$T_1 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \tag{3.40}$$

with T_1 we can compute the vertex' coordinates by multiplying it with $\binom{p}{1}$, the result is the vertex in homogeneous form. In coordinates it is $v_1 = (1, 1)$. The domain of v_1 can be calculated from $Proj_p(M_1)^{-R}$, for v_i we have the domain $p \leq 0$ and $1 \geq 0$. The other vertices are computed accordingly.

Domain Decomposition

The algorithm above gives a set of parameterized vertices and the parameter space where they are valid. For further use we can partition the parameter space so that we get distinct vertex sets, i.e. all vertices in one set are valid for all parameter values in that partition. Consider the example above; when we calculate all vertices, we get

$$\begin{array}{ll} v_1 = (1, 1) & p \leq 0 \\ v_2 = (1, 5) & p \leq -4 \\ v_3 = (1, 1 - p) & p \geq -4,\ p \leq 0 \\ v_4 = (6, 1) & p \leq 5 \\ v_5 = (p + 1, 1) & p \geq 0,\ p \leq 5 \\ v_6 = (6, 5) & p \leq 1 \\ v_7 = (6, 6 - p) & p \geq 1,\ p \leq 5 \\ v_8 = (p + 5, 5) & p \leq 1,\ p \geq -4 \end{array} \tag{3.41}$$

We can now get a partition by comparing the parameter domains and divide the space, into the overlapping and disjoint parts. For example v_1 and v_2 are both valid for $p \leq -4$ and only v_1 is valid for $-4 \leq p \leq 0$, so we get 2 partitions. If we continue with the other domains we get the desired partition, which is shown in figure 3.5. The domain partitions are also referred to as chambers.

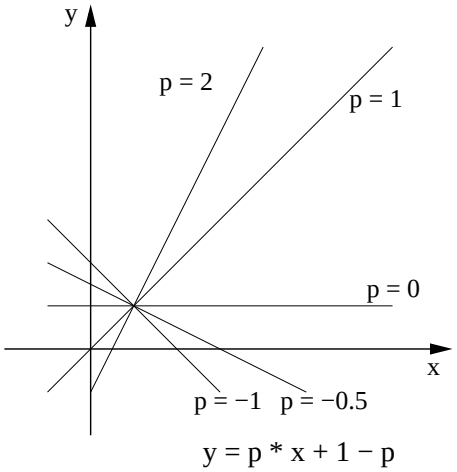


Figure 3.6: A Line with Non-Linear Parameterization

3.4.2 Non-Linearly Parameterized Polytopes

As shown before, polytopes with linear parameters can mostly be handled like unparameterized polytopes. Non-linear polytopes, however, are not so easy to process, since the orientation of rays and lines changes with the parameters (compare figure 3.3 and figure 3.6) and therefore we have to deal with many complex case distinctions. We transformed Chernikova's algorithm with a method introduced by Armin Größlinger, which uses quantifier elimination to simplify the arising case distinctions[Grö03]. We will very briefly explain the transformation and then show where and how the algorithm has to be changed and how we implemented it.

Processing Non-linear Parameters using Quantifier Elimination

In the preceding sections we represented polytopes by inequality systems with constant coefficients. This was also possible for linearly parameterized inequalities, for we treated the parameters like variables. To process non-linear parameters, we use fractions of polynomials as coefficients instead. So an inequality in our new representation has the following form:

$$c_1 x_1 + \ldots + c_n x_n + d \geq 0 \tag{3.42}$$

The domain of $c_1, \ldots, c_n$ is denoted by $\mathbb{Q}(p_1, \ldots, p_m)$, with

$$\mathbb{Q}(p_1, \ldots, p_m) := \{\frac{a}{b} | a, b \in \mathbb{Q}[p_1, \ldots, p_m], b \neq 0\} \tag{3.43}$$

where $\mathbb{Q}[p_1, \ldots, p_m]$ denotes the ring of polynomials with indeterminates $p_1, \ldots, p_m$ over $\mathbb{Q}$. A simple example, taken from [GGL04], for such an inequality is $p*x-1 \geq 0$.

As mentioned before, when we are using non-linear parameters we have to deal with case distinctions. The solution of $p * x - 1 \geq 0$ for x gives us three cases:

- $x \geq \frac{1}{p}$ if $p > 0$
- false if $p = 0$
- $x \leq \frac{1}{p}$ if $p < 0$

Armin Größlinger proposed a decision tree to represent such inequalities. It is defined by the following datatype:

$$\begin{array}{lll} \mathbf{data}\ \ Tree\ \alpha & = Leaf\ \alpha & \\ \quad | \quad SCond & Polynomial\ (Tree\ \alpha)\ (Tree\ \alpha)\ (Tree\ \alpha) & \\ \quad | \quad EqCond & Polynomial\ (Tree\ \alpha)\ (Tree\ \alpha) & (3.44) \\ \quad | \quad GeCond & Polynomial\ (Tree\ \alpha)\ (Tree\ \alpha) & \\ \quad | \quad FCond & QfFormula\ (Tree\ \alpha)\ (Tree\ \alpha) & \end{array}$$

The interpretation is:

- $Leaf\ x$ represents a result with value x
- $SCond\ fp\ b^-\ b^0\ b^+$ a case distinction over the sign of the fraction of polynomials $fp \in \mathbb{Q}(p_1, \ldots, p_m)$. If $fp < 0$ then branch b^- is chosen, if $fp = 0$ branch b^0 and finally if $fp > 0$ then branch b^-.
- $EqCond$ and $GeCond$ are used just like $SCond$, they are simply binary case distinctions over $fp = 0$ against $fp \neq 0$ ($EqCond$), and $fp < 0$ against $fp \geq 0$
- $FCond\ qf\ b^{true}\ b^{false}$ is a case distinction over the quantifier free logical formula qf, where b^{true} is applied if $qf = true$ and b^{false} otherwise.

For the previous example $p * x - 1 \geq 0$ the solution would be represented as:

$$SCond\ p\ (Leaf\ x \leq \frac{1}{p})\ (Leaf\ false)\ (Leaf\ x \geq \frac{1}{p}) \tag{3.45}$$

With this datatype, we can transform algorithms that rely on solving equations so that they can handle non-linear parameters. These changes have to be made:

- Whenever a decision over a sign is made, we use the $SCond$ constructor to build a new branching in the resulting data structure. In the same way we use the $EqCond$, $GeCond$ and $FCond$ constructors.
- Every expression e which leads to a final result is replaced by $Leaf\ e$.

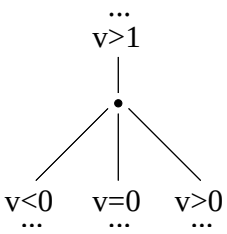


Figure 3.7: Decision Tree Node

- If a function $f :: \alpha \to \beta$ is applied to an expression e, that has, due to the transformation, changed its type from α to $Tree\ \alpha$, we have to apply the function to every leaf of the expression. Armin Größlinger provided a function $fmap$, which does this[1].

With these transformations it is surprisingly easy to enhance existing algorithms, so that they can process non-linear parameters.
Fortunately, many (probably most) case distinctions are either redundant or superfluous and can be eliminated. One way to simplify the decision tree is quantifier elimination. At every decision the domain of the current branch is narrowed and maybe some cases even become impossible because of the context. If we have a branch with context $v > 1$ and a case distinction, as shown in figure 3.7, we can set up the formulas $\forall v(v > 1 \to v < 0)$, $\forall v(v > 1 \to v = 0)$ and $\forall v(v > 1 \to v > 0)$. The quantifier elimination determines that only the last is true and hence $p > 0$ is the only possible case.
For further information about this method we refer to [Grö03] and [GGL04].

Changes on Chernikova's Algorithm for Non-linear Parameterized Polytopes

To process non-linearly parameterized polytopes several changes have to be made to the Chernikova algorithm. Let us recapitulate the phases of the algorithm:

Initialization the necessary data structures are initialized.

Iteration the constraints are processed one after the other, and for each we check if there is a line that does not saturate the current constraint. If there is we

- take the positive half of the line
- combine all other lines with that half-line, so that they lie on the hyperplane defined by the inequality

[1] In the later implementation the function was renamed to $cmapDTree$, in our text we will, however, stick to the naming of the paper [Grö03]

- combine all rays in the same manner
- and add the new ray to the other rays

else we

- split the set of rays into the sets of saturating, verifying and violating rays
- build linear combinations of verifying and violating rays
- discard redundant combinations
- union saturating, verifying and new rays

Finalization we transform the set of lines and rays to our desired representation

We will now see where and what kind of case distinctions will be introduced.

Initialization Only slight changes have to be made to the initialization. Since we don't need to compute anything, there is no need to incorporate any case distinctions. What we need is an enhanced datatype to represent our inequality systems and rays. As explained above, rational coefficients are insufficient and we use fractions of polynomials instead. Again, we initialize our lineality-space with unit vectors and our ray-space as empty.

Iteration We iterate over the set of constraints. At each step we have two options to proceed with. If there is a line which does not saturate the current constraint we project lines and rays (*case A*), or else we split and combine rays (*case B*). To make this decision for a constraint c and a lineality-space L, we solve $c * l = 0$ for every $l \in L$. This gives us the following branching:

$$\begin{array}{llll} EqCond & (c * l_n) & & \\ & (EqCond & (c * l_{n-1}) & \\ & & (\ldots\ (EqCond & (c * l_1) \\ & & & (Leaf\ \texttt{case A}) \\ & & & (Leaf\ \texttt{case B}))\ldots) \\ & & (\texttt{case B})) & \\ & (\texttt{case B}) & & \end{array} \tag{3.46}$$

This should not be mistaken as being an ordinary *case*-statement because it is a part of the resulting data structure. The *case A* and *case B* parts are the intermediate results that have to be further processed in the following steps.

case A We have a line l that does not saturate the current constraint c:

- Firstly, we have to identify the positive directed half-line which produces a case distinction over the sign of the result of $l * c$. We can represent this with an $GeCond$ node.
- The projection of the lines and rays along the current constraint does not necessarily produce any case distinctions, but as we want to keep our calculations simple and normalize the intermediate results we get additional branches here.
- Adding the positive half of l to the ray space does not require any case distinctions.

case B All lines saturate the current constraint c, so we only process the ray space:

- We split the set of rays R into sets of verifying, saturating and violating rays, this is done by the sign of the calculation $c * r,\ r \in R$. Thus we get an $SCond$ node.
- We build linear combinations of violating and verifying rays. Again only the normalization produces new branchings.
- To identify redundant vectors, we can use the $EqCond$ node type.
- The union does not produce new branches.

Finalization The result type of our algorithm is now a decision tree, with the single results at its leaves. Without processing they are represented as inequalities in homogeneous form. If we want any other representation we have to apply a transformation with the $fmap$ function. The decision tree may also be transformed into other data structures such as a list of domains.

We will now illustrate the procedure in an example. In order to keep the intermediate steps simple, we will omit some calculations that have no effect on the result. Consider a non-linearly parameterized polytope $P = \{(x, y) | 0 \leq x \leq 4, 0 \leq y \leq 4, px - y \geq 2p - 2\}$, the according matrix representation is

$$L = \left(\begin{array}{ccc|ccccc} 1 & 0 & 0 & 1 & 0 & -1 & 0 & p \\ 0 & 1 & 0 & 0 & 1 & 0 & -1 & -1 \\ 0 & 0 & 1 & 0 & 0 & 4 & 4 & 2-2p \end{array}\right) \qquad V = (). \tag{3.47}$$

The first 4 constraints are not parameterized at all, so we can process them as in examples before (cf. equation 3.23). After processing them we have the following matrices:

$$L = () \qquad V = \left(\begin{array}{ccc|cccc} 0 & 0 & 1 & 0 & 0 & 4 & 4 & 2-2p \\ 4 & 0 & 1 & 4 & 0 & 0 & 4 & 2+2p \\ 0 & 4 & 1 & 0 & 1 & 4 & 0 & -2-2p \\ 4 & 4 & 1 & 4 & 4 & 0 & 0 & -2+2p \end{array}\right) \tag{3.48}$$

When processing the last constraint, we have to make several case distinctions. First

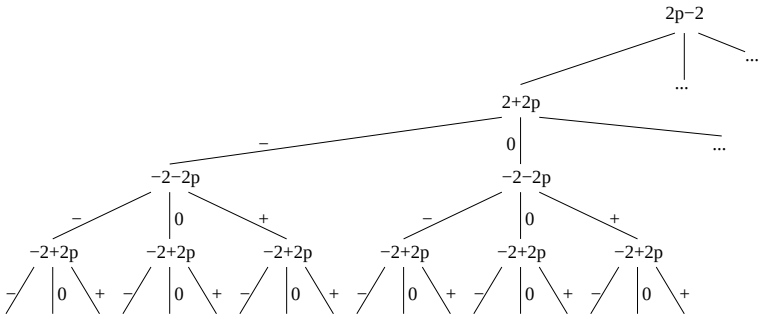


Figure 3.8: Part of a Decision Tree

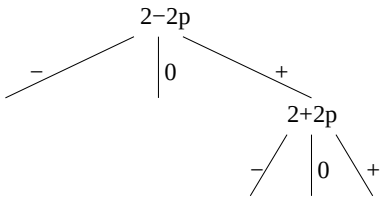


Figure 3.9: Reduced Decision Tree

we group the vectors in saturating, verifying and violating rays. Part of the resulting decision tree can be seen in figure 3.8. For each of the 4 vectors we get one $SCond$ node and hence we have $3^4 = 81$ possible solutions, of course many are redundant and invalid. If we reduce the tree, we get a much simpler result:

$$\begin{aligned}
p > 1 &: V^+ = \{v_2, v_4\}, V^0 = \{\}, V^- = \{v_1, v_3\} \\
p = 1 &: V^+ = \{v_2\}, V^0 = \{v_1, v_4\}, V^- = \{v_3\} \\
-1 < p < 1 &: V^+ = \{v_1, v_2\}, V^0 = \{\}, V^- = \{v_3, v_4\} \\
p = -1 &: V^+ = \{v_1\}, V^0 = \{v_2, v_3\}, V^- = \{v_4\} \\
p < -1 &: V^+ = \{v_1, v_2\}, V^0 = \{\}, V^- = \{v_3, v_4\}
\end{aligned} \tag{3.49}$$

The tree representation of the case distinctions can be seen in figure 3.9. We will do the following computations only for the parameter values $p > 1$, the other domains are treated accordingly. As the next step, we build combinations of violating and verifying rays and normalize the results. Although we get new case distinctions through the normalization, they are invalid. For $p > 1$ we have $V^- = \{v_1, v_3\}$ and $V^+ = \{v_2, v_4\}$ and, therefore, we could build combinations of the tuples $(v1, v2)$,$(v1, v4)$,$(v3, v2)$ and $(v3, v4)$, but since only $(v1, v2)$ and $(v3, v4)$ saturate enough common constraints we can omit the other two:

$$V^{new} = \left(\begin{array}{ccc|ccc} 1 & 0 & \frac{p}{2p-2} & 1 & 0 & \frac{p+1}{p-1} & \frac{2p}{p-1} & 0 \\ 1 & \frac{2p}{p+1} & \frac{p}{2p+2} & 1 & \frac{5p+3}{4p+4} & \frac{p-1}{p+1} & 0 & 0 \end{array} \right) \tag{3.50}$$

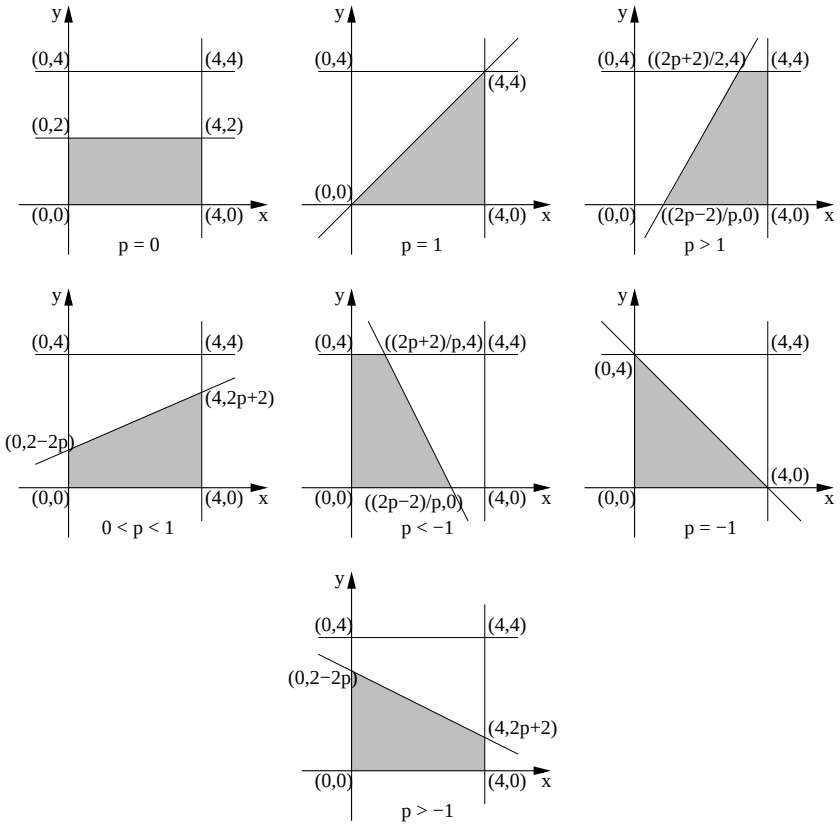


Figure 3.10: Polytope with the Non-Linear Inequality $px - y \geq 2p - 2$.

Together with the vectors in v^+ we for $p > 1$ have the vertices $(4, 0), (4, 4), ((2p - 2)/p, 0)$ and $((2p + 2)/p, 4)$. For all shapes the polytope takes see figure 3.10

3.5 Implementation Details and Conclusion

We implemented Chernikova's algorithm in three stages:

1. similar to the informal description (cf. section 3.2.1)
2. with Le Verge's enhancements (cf. section 3.3)
3. for non-linear parameterization (cf. section 3.4.2)

We will shortly present each version and point out the differences. For all of our implementations we used the functional programming language Haskell, plus data structures and functions provided by HsLooPo, a part of the LooPo project [Leh].

3.5.1 Simple Version

Our first implementation is a straight forward approach to the double decision method which is similar to our first informal description (cf. section 3.2.1). It is a conversion of an SML-implementation of Le Verge. We only show the code for the main function. The auxiliary functions are explained below.
The input of the function are three lists of vectors, where `old` is the list of processed constraints, `new` the list of unprocessed constraints and `ray` the list of rays. As initial input `old` should be empty, `new` should contain all constraints and `ray` an initial ray space (i.e. the unit vectors).
The function works recursively. At each step one constraint is chosen and the ray space is altered accordingly (cf. section 3.2.1). If all constraints are processed the resulting list of rays is output.

```
chernikova old new ray =
  if (length new) == 0
    then ray
    else
      let
        -- current is an arbitrary unprocessed constraint
        current = (head new);

        -- the rayspace is divided into verifying, saturating
        -- and violating rays
        verify    = filter (\x -> (mulVV current x) > 0) ray;
        saturates = filter (\x -> (mulVV current x) == 0) ray;
        violates  = filter (\x -> (mulVV current x) < 0) ray;

        -- newray is the set of linear combinations of verifying and
        -- violating rays
        newray = map (\(x, y) -> ((addVV (mulSV (mulVV current x) y)
                                   (negV (mulSV (mulVV current y) x))),
                                  x,
                                  y))
                     (cartesianProdukt verify violates)

        -- s filters all processed constraints that are saturated
        -- by x
        s x = filter (\c -> (mulVV c x) == 0) old

        -- irredundant filters all irredundant rays
        irredundant =
          map (\(x, y, z) -> normalize x)
                (filter (\(x,y,z) ->
                  (not (any (isSubList (s x))
                       (map s (delete y (delete z ray))))))
                 newray)
      in
        chernikova (old++[current])
                   (tail new)
                   (union (union verify saturates) irredundant)
```

Auxiliary functions:

`mulVV v1 v2` multiplies vector `v1` by vector `v2`.

`mulSV s v` multiplies vector `v` by scalar `s`.

`cartesianProdukt l1 l2` computes the Cartesian product of two list `l1` and `l2`.

`normalize v` normalizes vector `v`, by computing its entries greatest common divisor `gcd` and dividing `v` by `gcd`.

`isSubList l1 l2` checks if `l1` is a sublist of `l2`, that means `l2` contains all elements of `l1`

This implementation is very comprehensible, but it can only compute non-negative solutions and is quite inefficient since many superfluous calculations are done.

3.5.2 Version with Le Verge's Enhancements

This implementation can already compute negative solutions and is much faster than the simple one. Its based on Chernikova's approach and utilizes Le Verge's improvements.
Once again, we only present the main function and explain the auxiliary functions below. The detailed description of the functionality can be found in section 3.3.
This function needs much more information for each iterative step and hence has more arguments, which contain the following:

line is the lineality-space, represented by a list of vectors. The vectors are extended with the constraints as described in section 3.2.2. The initial set of vectors contains the unit vectors, where each vector is extended with the entries of the constraints in the according dimension.

ray is the ray-space, with the same representation as the lineality-space. It is initialized as empty.

n is the number of the current constraint, where $0 \leq$ `n` $<$ `nbConstraint`.

dim is the dimensionality of the constraints and the lineality- and ray-space.

nbConstraint is the number of constraints.

The result is a tuple of two lists, where the first contains the lineality-space and the second the ray-space. Both lines and rays are still extended with the constraint entries and therefore need further processing.

```
chernikova line ray n dim nbConstraint =
 if n < dim
 then (line,ray)
 else let
  -- find the list of lines that do not saturate the current constraint
  notSaturatingLines = filter (\x -> (vectorEntry x n) /= 0) line

  -- this section is only applied if notSaturatingLines is not empty
   -- we choose one line that is not saturating as current line
   currentLine = head notSaturatingLines
   saturatingLines = filter (\x -> (vectorEntry x n) == 0) line

   -- the not saturating lines are combined with the current line
   newLine = (map (\x -> combine x currentLine n)
              (delete currentLine notSaturatingLines)) ++ saturatingLines
   -- the positive half of the line is added to the rays
   absCurLine = if ((vectorEntry currentLine n) < 0)
                then mulVS currentLine (-1) else currentLine
   newRay = (map (\x -> combine x absCurLine n) ray)++[absCurLine]

  -- this section is only applied if notSaturatingLines is empty
   -- the rayspace is divided into verifying, saturating
   -- and violating rays
   verify    = filter (\x -> (vectorEntry x n) > 0) ray
   saturates = filter (\x -> (vectorEntry x n) == 0) ray
   violates  = filter (\x -> (vectorEntry x n) < 0) ray

   -- counts the number of constraints that two vectors saturate
   common vec1 vec2 lowerb ind nbCom sats =
    if lowerb == ind
     then (vec1,vec2,nbCom,sats)
     else if ((vectorEntry vec1 ind) == 0) &&
             ((vectorEntry vec2 ind) == 0)
          then common vec1 vec2 lowerb (ind-1) (nbCom+1) (sats++[ind])
          else common vec1 vec2 lowerb (ind-1) nbCom sats

   -- pairs of rays that saturate enough constraints
   commonConstraints =
    filter (\(_,_,z,_) -> ((z + (length line)) >= (dim -2)))
           (map (\(x,y) -> common x y n (nbConstraint + dim -1) 0 [])
           (cartesianProdukt verify violates))

   -- new rays from linear combinations, may contain rays
   -- from prior iterations
   newRedRays = (map (\(x,y,_,_) -> normalize (combine y x n))
                (filter (\(x,y,_,z) -> isNewRay x y z ray)
                        commonConstraints))
   newrays = union (union verify saturates) newRedRays
 in
  if notSaturatingLines == []
   then
    -- notSaturatingLines is empty
    chernikova line newrays (n-1) dim nbConstraint
   else
    -- notSaturatingLines is not empty
    chernikova newline newRay (n-1) dim nbConstraint
```

Auxiliary functions:

`vectorEntry v n` returns the entry of vector `v` in dimension `n`.

combine v1 v2 n computes a linear combination of the vectors `v1` and `v2`, that has the value 0 at dimension `n`.

mulVS v s multiplies the vector `v` by a scalar `s`.

common v1 v2 lb i nbSat satList checks how many and which constraints are saturated by a linear combination of vectors `v1` and `v2` that has the value 0 at dimension `i`. The function works recursively, the initial input is:

v1, v2 are two vectors

lb is the dimensionality of the constraint and the lineality- and ray-space (in our representation the constraint entries start at this entry of the vectors).

i is the number of the constraint, that the linear combination of `v1` and `v2` saturates.

nbSat is 0.

satList is an empty list.

The result of this function is a quadruple of the two vectors, the number of constraints that their linear combination saturates and a list of these constraints numbers.

notcommonzero v cList checks if a vector `v` saturates the constraints, whose numbers are given in the list `cList`. If it does the result is `False` else `True`.

isNewRay v1 v2 satList rayList checks if the linear combination of vector `v1` and `v2` that saturates the constraints with the numbers in `satList` saturates a different set of rays than any other ray in `rayList`.

normalize v normalizes the vector `v`, by computing its entries greatest common divisor `gcd` and dividing `v` by `gcd`.

Chernikova's algorithm is much more efficient than the first implementation, especially with Le Verge's improvements. It can also compute negative solutions and is able to process linear parameterized constraints (cf. section 3.4.1). This implementation is in large parts a conversion of Le Verge's C-implementation to the Haskell language (see [Ver94]).

3.5.3 Version for Non-linear Parameterization

Our last implementation makes use of the work of Armin Größlinger [Grö03] to deal with non-linear parameters in the coefficients of the constraints. It is based on the preceding version, but has some big differences. To handle the non-linear parameterization we changed the used data structures and built in case distinctions.

The input is the same as in 3.5.2. Only the data structures for the ray-space R and lineality-space L are different. L and R are again represented as lists of vectors, but now their entries may be fractions of polynomials. The initial values are the same as before.
Because of the case distinctions a single tuple of lineality and ray space is not sufficient anymore. Therefore a decision tree is used, which holds the different result spaces at its leaves.

```
chernikova line ray n dim nbConstraint =
  if n < dim
  then dLeaf (line,ray)
  else let
    -- filters the lines that do not saturate the current constraint
    lineEqUneqTree = filterVecEq line n

    -- if all lines saturate the current constraint apply caseB else caseA
    caseAOrB = compDTreeG (\(x,y) ->
                if (length y) > 0 then caseA x y else caseB x) lineEqUneqTree

    -- caseA is only applied if at least one line does not saturate
    -- the current constraint
    caseA satLine (curr:notSatLine) =
    -- find the positive directed half of the current line -> geCond
      geCond (vectorEntry curr n)
        (prod2DTreeG (prodDTreeG
          -- the lines are projected along the current constraint
                        ((map (\x -> combineFP x curr n) notSatLine)
                         ++(map dLeaf satLine)))
                     (prodDTreeG
          -- the positive directed half of the current line is added to
          -- the rayspace and all rays are projected along the constraint
                        ((map (\x -> combineFP curr x n) ray)
                         ++[(dLeaf curr)])))
        (prod2DTreeG (prodDTreeG
                        ((map (\x -> combineFP x curr n) notSatLine)
                         ++(map dLeaf satLine)))
                     (prodDTreeG
                        ((map (\x -> combineFP x (mulVS curr (-1)) n) ray)
                         ++[dLeaf (mulVS curr (-1))])))

    -- caseB is only applied if all lines saturate the current constraint
    -- the rayspace is partitioned in verifying, saturating and violating rays
    rayVerSatVio = filterVecS ray n
    caseB satLine = prod2DTreeG (dLeaf satLine) newRaysNorm

    -- pairs of rays that saturate enough common constraints with the number
    -- of the saturated constraints and a list of the constraints numbers
    commonConstraints = mapDTreeG (\(x,y,z) -> (x,y,
           (filter (\(_,_,z,_) -> ((z + (length line)) >= (dim -2)))
            (map (\(x,y) -> common x y n (nbConstraint + dim -1) 0 [])
             (cartesianProdukt x z))))) rayVerSatVio

    -- list of new rays from linear combinations
    newRays =
      mapDTreeG (\(ver,sat,coms) ->
       (ver++sat++ (map (\(x,y,_,_) -> combineNoNormFP y x n)
        (filter (\(x,y,_,z) -> isNewRay x y z ray) coms)))) commonConstraints
```

```
    newRaysNorm = compDTreeG normalizeListFP newRays
  in
    compDTreeG (\(x,y) -> chernikova x y (n-1) dim nbConstraint) caseAOrB
```

Auxiliary functions (functions not mentioned are explained in section 3.5.1 or 3.5.2):

`filterVecEq ls n` splits a list of vectors `ls` in two lists, one with vectors that have 0 at position `n` and one that does not. The result is a decision tree with all possible splittings at its leafs.

`compDTreeG f dt` applies a function `f`, that produces a decision tree to every leaf of the decision tree `dt`

`prod2DTreeG dt1 dt2` combines two decision trees `dt1` and `dt2`, so that there are tuples of `dt1` and `dt2` leaf values at the new tree's leaves.

`prodDTreeG dts` combines a list of decision trees `dts` so that on the combined tree's leafs lists of the leaf values of the old trees are.

`filterVecS ls n` splits a list of vectors `ls` in three lists, one with vectors that have a value greater than 0 at position `n` one with a value less than 0 and one with the value 0. The result is a decision tree with all possible splittings at its leaves.

`mapDTreeG f dt` applies a normal function `f` to every leaf of the decision tree `dt`

`combineFP v1 v2 n` computes a linear combination of two vectors `v1` and `v2` so, that the resulting vector has the value 0 at position `n`. The result is a decision tree with all possible combinations as its leaves.

This version of the algorithm is very different in its way of processing from the other two versions, since it passes much of the computational work to the decision tree and the underlying logic. Hence, the performance is very dependent on the implementation of the used data structure.

Chapter 4

Enumerator Computation

The number of integral points in a polytope is an important measurement for many applications. There are several implementations that can calculate this parameter for linearly parameterized polytopes. This chapter describes two existing methods to calculate the exact value and an estimation. The first technique is invented by Philippe Clauss [Cla96] and uses Ehrhart polynomials for its *symbolical* evaluation. The second approach [VSB$^+$04] also uses Ehrhart polynomials, but works with an *analytical* method based on Barvinok's decomposition [Bar93]. Finally we will present a method by Thomas Fahringer, which uses *symbolical analysis*.

4.1 Clauss' Method

Philippe Clauss presented a method to count the exact number of integer points in a linearly parameterized polytope [Cla96]. For the calculation he uses Ehrhart polynomials or more precisely Ehrhart pseudo-polynomials.

4.1.1 Ehrhart Polynomials

Eugene Ehrhart showed that the number of integer points in a polytope can be expressed by so called pseudo-polynomials or Ehrhart pseudo-polynomials [Ehr77]. A pseudo-polynomial is similar to a normal polynomial with the exception that it has rational periodic numbers as coefficients. Rational periodic numbers are defined as follows:

Definition 4.1 A rational periodic number $c(p)$ is a function $f : \mathbb{Z} \rightarrow \mathbb{Q}$ with a period q, so that $c(p) = c(p')$ whenever $p = p' \ mod \ q$.

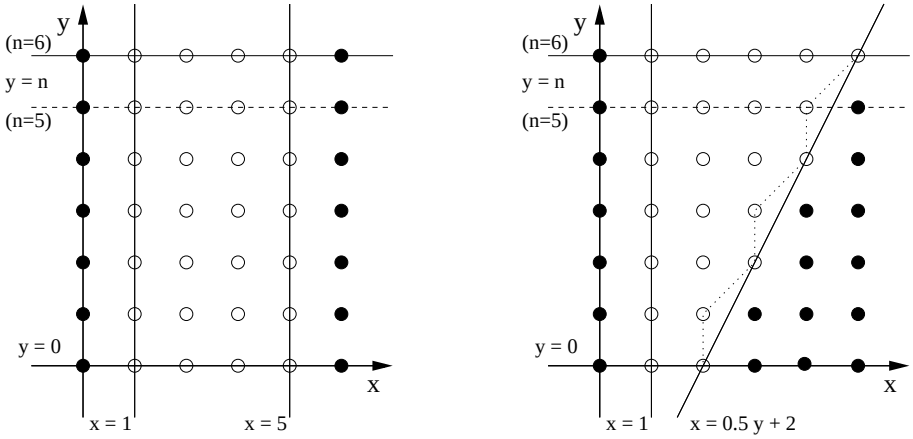


Figure 4.1: Linear Parameterized Polytopes

Ehrhart proposed a notation with q rational numbers enclosed in square brackets. A rational periodic number $c(p)$ with a period $q = 2$, where $c(p) = 5$ if $p \; mod \; q = 0$ and $c(p) = \frac{3}{4}$ if $p \; mod \; q = 1$ would for example be written as

$$c(p) = \left[5, \frac{3}{4}\right]_p \tag{4.1}$$

Now we can formally define pseudo-polynomials:

Definition 4.2 A pseudo-polynomial of degree d is a function f

$$f(x) = c_d(x)x^d + \ldots + c_1(x)x + c_0 \tag{4.2}$$

where $c_i(x)$ are rational periodic numbers. The period of f is the least common multiple of all $c_i(x)$. A pseudo-polynomial of degree 1 is a polynomial.

The connection between the number of integer points in a polytope and pseudo-polynomials is formulated in Ehrhart's theorem:

Theorem 4.1 Let $P \subset \mathbb{Q}^d$ be a rational polytope, the number of integer points in the dilations sP with $s \in \mathbb{N}$ is given by a pseudo-polynomial of degree d. The period of the pseudo-polynomial is a divisor of the least common multiple of the denominators of the vertices of P.

To provide a deeper insight to the theorem above we want to explain, where we need periodical functions to describe the number of integer points in a parameterized polytope. Let us explain this on figure 4.1. Both polytopes have three fixed and one parameterized supporting hyperplane. The left polytope's vertices however are always

integral for all (integral) values of the parameter n, thus we can, as mentioned above, represent the number of integer points in the polytope by a normal polynomial f, it is easy to see, that in this case it would be $f(n) = 4n$. The right polytope on the other hand has a parameterized vertex, that is not integral for certain values of the parameter n. The upper right vertex has the parameterized coordinates $\left(n, \frac{1}{2}n + 2\right)$ and for odd values of n the vertex is therefore not integral. The dotted line indicates that the number of integer points does not grow continuously, but periodical. So we can't set up a polynomial as before, instead we can either define a number of polynomials, as many as there are periods, and make a case distinction on n, or we set up an Ehrhart pseudo-polynomial with according period. We will explain below how this can be done.

4.1.2 Clauss' Method

Philippe Clauss developed an algorithm to compute the the number of integer points in a linearly parameterized polytope. In this section we will explain this method informally, the outline of the algorithm will be given in the next section. For proofs and more mathematical explanations please refer to the original publication [Cla96].
As explained before a linearly parameterized polytope may take different shapes for different parameter values (see section 3.4.1). The parameter space is divided into disjoint domains, where the shape is not changing, i.e. the vertices are well-defined. For each of those domains the enumerator is calculated. The enumerator of a domain is the polynomial or pseudo-polynomial, that describes the number of integer points in the polytope as a function of its parameters.
From the definitions above we have some information about the structure of the enumerator function of a polytope within a certain domain. The degree d of the (pseudo-) polynomial f is equal to the dimensionality of the domain. If we know the number of parameters we can set up a prototype of the pseudo-polynomial. For one parameter n, this would look as follows:

$$f(n) = c_d(n)n^d + \ldots + c_1(n)n^1 + c_0(n) \tag{4.3}$$

where $c_0, \ldots, c_d$ are rational periodic numbers. The period of c_i is a divisor of the denominator of the domain, so we assume it is the denominator and simplify later if possible. For the example in figure 4.1 (right polytope) the prototype, with expanded periodical rational numbers, would be $f(n) = [c_{12}, c_{22}]_n n^2 + [c_{11}, c_{21}]_n n + [c_{10}, c_{20}]_n$. The coefficients have period 2, since the denominator of the polytope is 2. We obtain the denominator by calculating the LCM of the denominators of the vertices of the polytope. The vertices are provided by Chernikova's algorithm, which we discussed in detail in chapter 3.
To compute more than one parameter Clauss uses symbolic variables instead of calculating the equation with all parameters at once. So for a two-dimensional polytope with two parameters a, b, he first sets up a system with the first parameter a, like

$f(a,b) = d_2a^2 + d_1a + d_0$ and recursively generate equations for additional parameters:

$$\begin{array}{rcl} d_2 & = & c_{22}b^2 + c_{21}b^2 + c_{20} \\ d_1 & = & c_{12}b^2 + c_{11}b^2 + c_{10} \\ d_0 & = & c_{02}b^2 + c_{01}b^2 + c_{00} \end{array} \tag{4.4}$$

Now the task is to compute the coefficients of the pseudo-polynomial. If the pseudo-polynomial has degree d, period q and p parameters we have $(d+1)^p q$ unknown coefficients and $(d+1)^p$ unknowns for each period. Clauss uses a form of interpolation to compute these.
To compute $(d+1)^p q$ variables we need in general $(d+1)^p q$ equations. To get these we actually count the number of integer points for $(d+1)^p q$ parameter values. For each period $q_i \in \{0, \ldots, q-1\}$ we choose the first d values v_j in every parameter's domain, where $v_i \ mod \ q = q_j$, and calculate the number of integer points of the polytope. In our example is $q = 2$ and $d = 3$, so we count the numbers of integer points for q_0 where $n \in \{0, 2, 4\}$ and for q_1 where $n \in \{1, 3, 5\}$. The solutions are:

$$q_0 \begin{cases} f(0) = 2 \\ f(2) = 7 \\ f(4) = 14 \end{cases} \quad \text{and} \quad q_1 \begin{cases} f(1) = 4 \\ f(3) = 10 \\ f(5) = 18 \end{cases} \tag{4.5}$$

With the calculated results we can set up a system of equations for each period, where we simply set in the values of the parameters and the calculated result. By solving these systems we get the coefficients for each period. In our example this would be:

$$q_0 \begin{cases} 0c_{12} + 0c_{11} + c_{10} = f(0) = 2 \\ 4c_{12} + 2c_{11} + c_{10} = f(2) = 7 \\ 16c_{12} + 4c_{11} + c_{10} = f(4) = 14 \end{cases} \implies \begin{array}{l} c_{10} = 2 \\ c_{11} = 2 \\ c_{12} = \frac{1}{4} \end{array} \tag{4.6}$$

$$q_1 \begin{cases} c_{22} + c_{21} + c_{20} = f(1) = 4 \\ 9c_{22} + 3c_{21} + c_{20} = f(3) = 10 \\ 25c_{22} + 5c_{21} + c_{20} = f(5) = 18 \end{cases} \implies \begin{array}{l} c_{20} = \frac{7}{4} \\ c_{21} = 2 \\ c_{22} = \frac{1}{4} \end{array} \tag{4.7}$$

After solving these systems we have the coefficients for each period, and we can combine them to the complete pseudo-polynomial,

$$f(n) = \left[\frac{1}{4}, \frac{1}{4}\right]_n n^2 + [2,2]_n \, n + \left[2, \frac{7}{4}\right]_n . \tag{4.8}$$

Obviously we can simplify this result. The first two coefficients are constant, since they have the same values for all periods, so the simplified pseudo-polynomial is:

$$f(n) = \frac{1}{4}n^2 + 2n + \left[2, \frac{7}{4}\right]_n \tag{4.9}$$

4.1.3 Algorithm Outline

The input of the algorithm is an inequality system, which describes a linearly-parameterized polytope. These steps will be applied: First the polytopes domains and according vertices are computed, for each domain the following steps are applied:

Basic factors: The number parameters and the dimensionality of the domain are calculated

Denominator: The domain denominator is calculated.

Prototype: A prototype of the domain enumerator is set up.

Interpolation: The enumerator is calculated:

- The integral points are counted for a number of parameter values
- The enumerators coefficients are calculated, by solving a system of equations

Simplification: The pseudo-polynomial is simplified

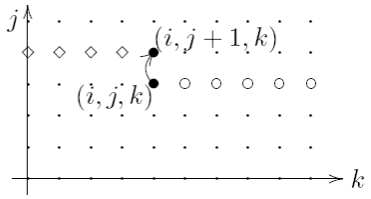
4.1.4 Review

The use of Ehrhart pseudo-polynomials to describe the enumerator of polytopes is very convenient and gives easily understandable formulas since it does not use complex functions such as min, max and mod. Clauss' method to compute the Ehrhart polynomials is also very comprehensible. It is implemented in the PolyLib library [Wil93]. However, Beyls lists three serious limitations, which will be discussed below [Bey04]. Methods to reduce these limitations will be shown in the next section.

Counting non-parameterized polytopes To set up initial equation systems in the interpolation method, the number of integral points in fixed sized polytopes have to be counted. In PolyLib this is done, by creating a set of loops, which enumerate the integer points of an hyperrectangle, which encloses the original polytope. At each iteration, that is for each point, there is a test, if it lies within the original polytope.
Consider a polytope with constraints $0 \leq i \leq N, 0 \leq j \leq N$ where $1000000 \leq N \leq 2000000$. To calculate the enumerator the integral points of 3 polytopes have to be calculated for $N \in \{1000000, 1000001, 1000002\}$. For $N = 1'000'000\,100'000'000'000$ points have to be checked whether they lie in the polytope or not, which of course results in a noticeable execution time.
A more efficient way to count the number of integer points in a fixed sized polytope was given by Barvinok [Bar94]. In fact the interpolation method is not depending on the inefficient counting method. It could easily be replaced by Barvinok's algorithm and therefore this limitation does not reveal a substantial problem. We will discuss this algorithm later in section 4.2.

```
do i = 0, 199
    do j = 0, 199
        s = 0
        do k = 0, 199
            s = s + A[i][k] * B[k][j]
        enddo
        C[i][k] = s
    enddo
enddo
```

(a) Source code



(b) Intermediate accesses

Figure 4.2: Matrix Multiplication: Code and Intermediate Accesses

Degenerate domains A more serious limitation is the problem of degenerate domains. For the initial equation systems the number of integer points in $(d+1)^p q$ fixed sized polytopes is counted. In order to get linearly independent equations Clauss takes the values from a hyperrectangle in the parameter domain. This method is also known as Vandermonde interpolation. There are however cases where no large enough hyperrectangle can be found in the parameter domain, then Clauss method fails to produce an answer, although with different parameter values the answer might be found. The parameter domains where this problem occurs are known as degenerate domains. Solutions could be found with different interpolation methods.

Large solution size For some polytopes the periods of the Ehrhart polynomials get very large and hence a representation in another form is preferable. This is illustrated best in a practical example from Beyls [Bey04].
Consider the program code for matrix multiplication in figure 4.2a, suppose we want to count the number of distinct Translation Lookaside Buffer (TLB) pages accessed between two consecutive accesses to the same TLB page. This indicates the number of TLB page misses that can be expected and is called the reuse distance.
For simplicity, we will assume that `A[i][k]` and `B[k][j]` access different TLB pages and we will concentrate on `A[i][k]`. We assume that `A` is a 200×200 matrix, which is laid out in column major order, and starts at address zero. Furthermore, an element size of 4 bytes is assumed. As such, `A[i][k]` is located at address $4 \times (200k + i)$.
Iterations (i, j, k) and $(i, j + 1, k)$ access the same array element `A[i][k]`. Figure 4.2b shows the iterations that are executed between these two iterations, iterations $(i, j, k+1 \cdots 199)$ ($\circ$ in the figure) and iterations $(i, j+1, 0 \cdots k-1)$ ($\diamond$ in the figure). The set of TLB pages accessed by the $\circ$-iterations can be described as

$$S_1 = \left\{ t \,\middle|\, \exists k' : t = \left\lfloor \frac{800k' + 4i}{L} \right\rfloor \wedge 0 \leq i, j, k \leq 199 \wedge k + 1 \leq k' \leq 199 \right\}, \quad (4.10)$$

where i, j and k are parameters. Assuming page size $L = 4096$, this can be written as a set of linear constraints:

$$S_1 = \{t | \exists k' : 1024t \leq 200k' + i \leq 1024t + 1023 \wedge 0 \leq i, j, k \leq 199 \wedge k + 1 \leq k' \leq 199\} \tag{4.11}$$

and further simplified to (for example by using the Omega library)

$$\begin{aligned} S_1 = \{t | 0 \leq i \wedge 1024t - 39800 \leq i \leq 199 \wedge 0 \leq k \leq 198 \\ \wedge 0 \leq j \leq 199 \wedge i + 200k \leq 823 + 1024t\}. \end{aligned} \tag{4.12}$$

For the $\diamond$-iterations a similar equation is obtained. The total count of TLB pages is $\#(S1 \cup S2) = \#S1 + \#(S2 \setminus S1)$. Concentrating on S1, we see that it is a one-dimensional polytope and using PolyLib we can find out that its vertices are

$$\frac{i}{1024} + 25\frac{k}{128} - \frac{823}{1024} \text{ and } \frac{i}{1024} + \frac{4975}{128}. \tag{4.13}$$

Since the dimension of this polytope is $d = 1$ and the periods are $q_i = 1024$, $q_j = 1$ and $q_k = 128$, the resulting Ehrhart polynomial has 4 terms and each has a periodic coefficient with period 1024. In total it is represented by $4 \cdot 1024 = 4096$ rational numbers.
Using Barvinok's decomposition (see section 4.2), a much simpler form could be obtained:

$$\left\lfloor \frac{i + 888}{1024} \right\rfloor - \left\lfloor \frac{i + 200k + 199}{1024} \right\rfloor + 39 \tag{4.14}$$

4.2 Using Barvinok's Decomposition

An alternative approach to the interpolation method by Clauss was presented by Verdoolaege et al. [VWBC05], it reduces the limitations of Clauss' method by using Barvinok's decomposition [Bar93]. To compute the enumerator of a polytope this algorithm considers its *generating function*. We will first explain generating functions, then how to find them for polytopes and later how to evaluate them.
The algorithm consists of the following transformations and calculations:

Supporting cones The polytope is decomposed into its supporting cones.

Shifted cones The cones are shifted to the origin.

Simplicial cones The cones are triangulated into simplicial cones.

Unimodular cones The simplicial cones are decomposed into unimodular cones.

Figure 4.3: One-dimensional polytope with 5 lattice points

Generating functions For each unimodular cone the generating function is determined.

Number of lattice points The generating functions of the unimodular cones are evaluated, shifted back and summed up

4.2.1 Generating Functions

Even if generating functions are often used as a more general construct, we will rely on the following definition:

Definition 4.3 (Generating function) The generating function of a set $A \subset \mathbb{Q}^d$ is the generating function of the integer points in A. It is a formal power series with a term for each integer point in A and can be written as:

$$f(A, x) = \sum_{a \in A \cap \mathbb{Z}^d} x^a \tag{4.15}$$

Evaluating this function at 1 returns the number of terms of f which equals the enumerator of A.

With the above definition we can calculate the enumerator of a polytope via it's generating function. But by now we still have to enumerate all integer points, which is one of the deficits of the interpolation method (see section 4.1). Consider the following simple example:
Let $P = \{p | 0 \leq p \wedge 2p \leq 9\}$ be a one-dimensional polytope (shown in figure 4.3), in the form of equation 4.15 the generating function is:

$$f(P, x) = x^0 + x^1 + x^2 + x^3 + x^4 \tag{4.16}$$

If we evaluate this function at $x = 1$, we get the correct enumerator 5:

$$f(P, 1) = 1^0 + 1^1 + 1^2 + 1^3 + 1^4 = 5 \tag{4.17}$$

Let's take a look at the cone $C = \{c | c \geq 0\}$, we see that it's generating function is the geometric series:

$$f(C, x) = x^0 + x^1 + x^2 + \ldots = \sum_{a=0}^{\infty} x^a = \frac{1}{1 - x} \tag{4.18}$$

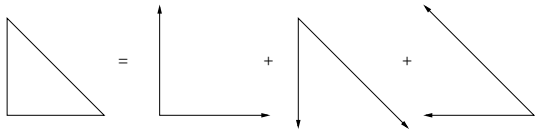


Figure 4.4: Polytope and its Supporting Cones.

Since this function contains more terms than we want for P, we subtract the part that lies outside of the polytope:

$$f(P,x) = f(C,x) - f(C+7,x) = \sum_{a=0}^{\infty} x^a - \sum_{a=5}^{\infty} x^a = \frac{x^0}{1-x} - \frac{x^5}{1-x} \quad (4.19)$$

The problem with this function is, that it always leads to a division by zero, when we evaluate it at point $x = 1$. To circumvent this we have to compute $\lim_{x\to 1} f(P,x)$. We will show how this can be done in section 4.2.6. But first we will explain how to set up the generating function for arbitrary polytopes. For this we have to decompose the polytopes into unimodular cones.

4.2.2 Supporting Cones

The first step from a polyhedron towards its unimodular cones is to consider the supporting cones of the polyhedron.

Definition 4.4 (Supporting cone) Let $P = \{x \in \mathbb{Q}^d | Ax \geq 0\}$ be a (parametric) polyhedron with a (possibly parametric) vertex v, then

$$cone(P,v) = \{x \in \mathbb{Q}^d | Bx \geq 0\} \quad (4.20)$$

is the supporting cone of P at v, where B is the submatrix of A containing the constraint that are active on v, i.e. the constraints that define the supporting hyperplanes that intersect at v.

An example of the supporting cones of a polytope is shown in figure 4.4.
Brion showed in 1988 that the generating function of a polyhedron is the same as the sum of the generating functions of its cones [Bri88].

Theorem 4.2 (Brion's theorem) The generating function $f(P,x)$ of a polyhedron P is equal to the sum of generating functions of its supporting cones,

$$f(P,x) = \sum_{v(p)\text{vertex of}P} f(cone(P,v),x) \quad (4.21)$$

We already used this theorem intuitively in equation 4.19. These supporting cones are in general not (shifted) unimodular, so we have to further decompose them.

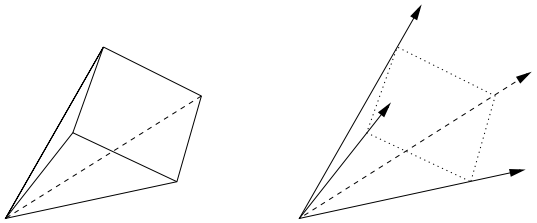


Figure 4.5: Degenerate Polytope and its Non-Simplicial Supporting Cone.

4.2.3 Triangulating Non-Simplicial Cones

If the polytope is degenerated, the supporting cones are not only not unimodular, but also not simplicial. A simplicial cone can be defined as follows.

Definition 4.5 (Simplicial cone) A cone of dimension d is simplicial, if has only d rays and lines, i.e. it's number of generators is d.

An example of a non-simplicial cone can be seen in figure 4.5. If we have a non-simplicial supporting cone, we have to triangulate it. There are several techniques to do so. Verdoolaege uses the Delaunay triangulation [VWBC05]. We will only give a short overview, how this can be done.
The basic idea of the Delaunay triangulation is to lift the cone, i.e. to add a new dimension. We will refer to this dimension as height of the cone's generators. If our cone is $C = \{x \in \mathbb{Q}^d | Ax \geq 0\}$ with generators u_i, we extend each generator with a coordinate λ_i, so the new generators are $\{(u_i, \lambda_i)\}$ and

$$C^\lambda = \{x \in \mathbb{Q}^{d+1} | A^\lambda x \geq 0\} \tag{4.22}$$

Now we can compute this lifted cones lower faces, which are defined as

$$\{x \in C^\lambda | A^\lambda_+ x = 0\} \tag{4.23}$$

where A^λ_+ contains only those row vectors of A^λ, where the $d+1$st entry is positive. The polyhedral complex built by the lower faces is called lower envelope. If we project it to the first d dimensions, we have a subdivision of C, which will be a triangulation for generic choices of λ.
The Delaunay triangulation uses $\lambda_i = \sum_j u^2_{i,j}$. This however may not always succeed and Verdoolaege then uses random heights, which in practice always lead to a triangulation.
To get the lower envelope we compute the implicit representation of the polyhedron generated by the rays $\{(u_i, \lambda_i)\}$ and the origin. This can be done by Chernikova's algorithm, which we discussed earlier (see chapter 3). The result will also contain the upper envelope and to avoid this additional computation we add the ray $(0, \ldots, 0, 1)$

to the cone, so only the lower and some vertical facets are computed.
Let us illustrate this with an example. Consider the non-simplicial cone in figure 4.5, its generators may be

$$R^C = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}. \tag{4.24}$$

The Delaunay triangulation uses the heights $\lambda = (1, 2, 2, 3)$, as we will see this does not lead to a valid triangulation, so we will also use random heights $\lambda' = (3, 2, 5, 6)$. With the optimization the lifted cones $C^{\lambda}_{\uparrow}$ and $C^{\lambda'}_{\uparrow}$ have the generators

$$R^{C^{\lambda}_{\uparrow}} = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \\ 3 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right\} \tag{4.25}$$

and

$$R^{C^{\lambda}_{\uparrow}} = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ 3 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 1 \\ 5 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \\ 6 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right\}. \tag{4.26}$$

The implicit representation is

$$A^{\lambda} = \begin{pmatrix} -1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 1 & -1 & 0 & 0 \end{pmatrix} \tag{4.27}$$

and

$$A^{\lambda'} = \begin{pmatrix} -3 & 1 & -2 & 1 \\ -1 & -1 & -4 & 1 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{4.28}$$

It is obvious that the Delaunay triangulation will fail, since A^{λ}_{+} has only one row $(-1, -1, -1, 1)$ and hence we have only one lower facet, which is the original cone. $A^{\lambda'}_{+}$ however has two rows, $a_1 = (-3, 1, -2, 1)$ and $a_2 = (-1, -1, -4, 1)$. Now we

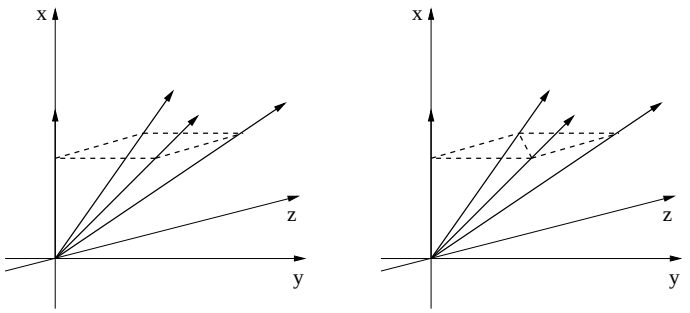


Figure 4.6: Triangulation of a Non-Simplicial Cone.

have to find the sets of lifted rays (u_i, λ_i) which saturate the equation $a_j(u_i, \lambda_i) = 0$. We get the sets

$$R_{a_1} = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ 3 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 1 \\ 5 \end{bmatrix} \right\} \tag{4.29}$$

and

$$R_{a_2} = \left\{ \begin{bmatrix} 1 \\ 1 \\ 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 1 \\ 5 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \\ 6 \end{bmatrix} \right\}. \tag{4.30}$$

Projected to the first d dimensions we have two simplicial cones, which are a triangulation of the original cone (see figure 4.6).

After the triangulation we have cones, which have common borders. Since we calculate the number of integral points in each cone, points that lie on the borders might be counted twice and hence we should subtract the affected faces. We will now see how to overcome this problem.

To circumvent handling lower dimensional faces we use Brion's polarization trick [Bri88]. The polar cone C^* of a cone C has rays and constraints interchanged, i.e.

$$C^* = \{y | \forall x \in C : x \bullet y \geq 0\}. \tag{4.31}$$

We now use the fact that any linear equality between a set of cones also holds for the set of their polar cones [BP99]. Hence we can polarize the cone, decompose it and ignore the lower dimensional faces, since they contain lines after the repolarization.

To compute the polar we only interchange rays and constraints. An example can be seen in figure 4.7, the right cone $C = \{(x, y) | x + 2y \geq 0, x - 2y \geq 0\}$ has the generators $u_1 = (2, 1)$ and $u_2 = (-2, 1)$, while it's polar $C^* = \{(x, y) | 2x + y \geq 0, -2x + y \geq 0\}$ on the left has the generators $u_1^* = (1, 2)$ and $u_2^* = (1, -2)$.

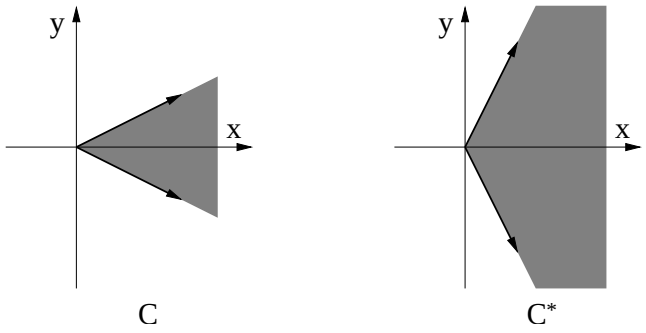


Figure 4.7: Cone and its Polar.

In the following we will always assume that we work on the polar of a cone, when we perform any kind of decomposition. Therefore we ignore lower dimensional faces.

4.2.4 Barvinok's Decomposition

Definition 4.6 (Unimodular cone) A rational polyhedral cone

$$C = \{x \in \mathbb{Q}^d | Ax \geq 0\} \tag{4.32}$$

is unimodular, if $A \in \mathbb{Z}^d \times \mathbb{Z}^d$ is a unimodular matrix, i.e. the determinant of A is ± 1. We call the cone shifted if the right-hand side is not zero, i.e. $C = \{x \in \mathbb{Q}^d | Ax \geq \gamma\}$. It may also be parametric, then $\gamma = Bp + b$, where $B \in \mathbb{Q}^{d \times n}$ and $b \in \mathbb{Q}^d$.

After the triangulation we have only simple cones, which however are in general still not unimodular. To achieve unimodularity, we have to further decompose them. Barvinok described a polynomial algorithm to do so [Bar94].
The main idea is to decompose a cone C, whose generator matrix K has determinant $det(K)$, into several cones C_i with $det(K_i) < det(K)$. This is done repeatedly until all resulting cones are unimodular, i.e. their generator matrices have determinant 1.
We decompose the cone C by finding an integral vector w, which can be written as a linear combination of the generators of C with small coefficients, to be more precisely

$$w = \sum_{i=1}^{d} \alpha_i u_i \qquad |\alpha_i| \leq |det(K)|^{-\frac{1}{d}} \tag{4.33}$$

With a suitable w we can decompose a cone C into d cones C_i with smaller determinants. For each cone C_i we simply replace a different generator by w^T. It can be shown that we can always find a proper w, which will lead to smaller determinants [VWBC05].
We will only describe the method to find w briefly, for deeper insights in the mathematical background we again refer to [VWBC05]. Let $B = K^T$ be the matrix with generators of a cone C as rows and $\Delta = det(B)$. We are searching for an integral $w^T = \alpha^T B$

such that the infinity norm $||\alpha||_\infty = max(|\alpha_1|, \ldots, |\alpha_d|)$ is minimal. To find the w, we are looking for a small α, which is equal to searching a $\lambda^T = \Delta\alpha^T = \Delta w^T B^{-1}$. ΔB^{-1} is a basis to an integer lattice, hence we can use Lenstra, Lenstra and Lovasz' basis reduction algorithm (LLL) to find an $B' = U(\Delta B^{-1})$, where B' is a reduced basis for the lattice ΔB^{-1} with nearly orthogonal vectors and U is an unimodular matrix [LJL82]. These vectors are short in sense of the Euclidean but not the infinity norm, so search for a linear combination with small coefficients μ. This gives us

$$\lambda^T = \mu^T B' \tag{4.34}$$

and with $\lambda^T = \Delta w^T B^{-1}$ and $B' = U(\Delta B^{-1})$ we get

$$w^T = \mu^T U. \tag{4.35}$$

The search for μ may be very expensive and so only unit vectors are tried, which means that possible values for w are only rows of U. If no element of λ is strictly positive, we replace w by $-w$. In practice this always leads to a valid reduction. The implementation of the LLL algorithm Verdoolaege uses is part of Victor Shoup's Number Theory Library (NTL) [Sho04].
Let's look at a very simple example, recall the cone $C = \{(x, y)|x + 2y \geq 0, x - 2y \geq 0\}$ in figure 4.7. It is not unimodular, since the determinant of generator matrix

$$B = \begin{pmatrix} 2 & 1 \\ -2 & 1 \end{pmatrix} \tag{4.36}$$

is 4. Now we have to calculate the reduced basis of $4B^{-1}$

$$4B^{-1} = \begin{pmatrix} 1 & -1 \\ 2 & 2 \end{pmatrix}. \tag{4.37}$$

The reduced basis B' is

$$B' = \begin{pmatrix} 1 & -1 \\ 2 & 2 \end{pmatrix} \qquad U = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{4.38}$$

We used the Maple$^{\mathrm{TM}}$ implementation of the LLL algorithm for this step. The first row of B' is smaller than the second in the infinity norm and hence we choose $w^T = (1, 0)$. This gives us two new cones C_1 and C_2, which have generators matrices

$$K_1 = \begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix} \qquad K_2 = \begin{pmatrix} 1 & -2 \\ 0 & 1 \end{pmatrix}, \tag{4.39}$$

which are both unimodular.

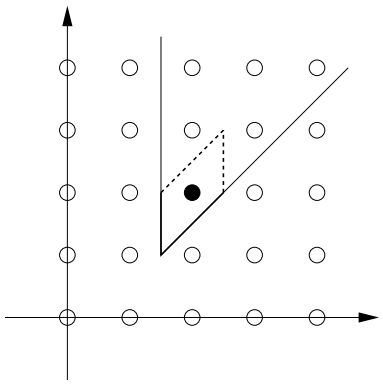


Figure 4.8: Fundamental Parallelepiped of a Shifted Unimodular Cone.

4.2.5 Generating Functions for Unimodular Cones

At first we only consider a unimodular cone $C = \{x \in \mathbb{Q}^d | Ax \geq 0\}$, it may be defined by it's rays $u_1, \ldots, u_d$, which are the columns of $U = C^{-1}$,

$$C = \{\lambda_1 u_1 + \lambda_2 u_2 + \ldots + \lambda_d u_d | \lambda \geq 0\}. \tag{4.40}$$

It can be shown, that the generating function of such a unimodular cone can be expressed by the following formula [BP99]:

$$f(C, x) = \frac{1}{(1 - x^{u_1})(1 - x^{u_2}) \cdots (1 - x^{u_d})} \tag{4.41}$$

If the cone is shifted by an integer vector, that means its vertex is not the origin, but an integer vertex $v = U\gamma$, we can simply multiply the $f(C, x)$ by the factor x^v. But since vertices may be non-integral, we have to calculate the correct integral multiplication factor. We need the integral point, that lies within the fundamental parallelepiped Π of cone C.

Definition 4.7 (Fundamental parallelepiped) The fundamental parallelepiped Π of a shifted cone C with generators u_i and vertex v is

$$\Pi = v + \left\{ \sum_{i=1}^{d} \alpha_i u_i \Big| 0 \leq \alpha_i < 1 \right\} \tag{4.42}$$

It contains one single integer point w with

$$w = \sum_{i=1}^{d} \lceil \gamma_i \rceil u_i = \sum_{i=1}^{d} \alpha_i u_i \tag{4.43}$$

An example of a fundamental parallelepiped can be seen in figure 4.8, it is not closed, since the dashed borders lie outside. The cone's vertex is $v = (1.5, 1)$ and its generators are $u_1 = (0, 1)$ and $u_2 = (1, 1)$. We have

$$C = \{(x_1, x_2) | 2x_1 \geq 3, 2x_1 - 2x_2 \leq 1\} \tag{4.44}$$

and therefore

$$A = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}, \quad \gamma = \begin{pmatrix} -0.5 \\ 1.5 \end{pmatrix}. \tag{4.45}$$

We get $w = \lceil \gamma_1 \rceil u_1 + \lceil \gamma_2 \rceil u_2$, which is

$$w = \lceil -0.5 \rceil \begin{pmatrix} 0 \\ 1 \end{pmatrix} + \lceil 1.5 \rceil \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \end{pmatrix} \tag{4.46}$$

As expected the fundamental parallelepiped contains one integer point $w = (2, 2)$. With this integral point w we get the following general definition of $f(C, x)$,

$$f(C, x) = \frac{x^w}{(1 - x^{u_1})(1 - x^{u_2}) \cdots (1 - x^{u_d})}, \tag{4.47}$$

with

$$w = \sum_{i=1}^{d} \lceil \gamma_i \rceil u_i = -\sum_{i=1}^{d} \lfloor -\gamma_i \rfloor u_i. \tag{4.48}$$

For our example this is

$$f(C, x) = \frac{x_1^2 x_2^2}{(1 - x_1)(1 - x_1 x_2)}, \tag{4.49}$$

With this formula it is easy to calculate the generating function of a (shifted) unimodular cone.

4.2.6 Evaluating Generating Functions

Now we can set up generating functions for any non-parameterized polytope. What remains is to evaluate them at 1. We will only show the calculations that have to be done, since the mathematical background is quite challenging (see [VWBC05]). We will illustrate this on our introducing example, which is shown in figure 4.3. It was a one dimensional polytope containing 5 lattice points. We found the following generating function

$$f(P, x) = \frac{x^0}{1 - x} - \frac{x^5}{1 - x}. \tag{4.50}$$

We can treat each summand separately and build the sum later:

$$\begin{aligned} f(P,x) &= f(C_1,x) + f(C_2,x) \\ f(C_1,x) &= \frac{x^0}{1-x},\ f(C_2,x) = \frac{x^5}{1-x} \end{aligned} \tag{4.51}$$

It should be noticed, that this property allows us to evaluate the generating functions for all unimodular cones separately. At first we substitute x by $s+1$ and expand about $s=0$ by polynomial division.

$$\begin{aligned} f(C_1,s) &= \frac{1}{1-(s+1)} = \frac{1}{s} \\ f(C_2,s) &= \frac{(s+1)^5}{1-(s+1)} = \frac{1+5s+10s^2+10s^3+5s^4+s^5}{s} \end{aligned} \tag{4.52}$$

In our example this is no problem but in general the x might be multidimensional, then we either chose a vector λ, such that the inner product of λ and the generators is different from zero, or we recursively apply the evaluation for each dimension (an example for the first can be found in [Bey04] and for the second in [VWBC05]). The general form of this summands is

$$f(C_i,s) = E[i]\frac{(1+s)^{num_i}}{\prod_{j=1}^{r}((1+s)^{den_{ij}}-1)} \tag{4.53}$$

where $E[i]$ is the sign and r is the number of rays of the corresponding cone. The function has still a pole at $s=0$ of an order equal to the number of factors in the denominator, which is r. We can write this as

$$f(C_i,s) = E[i]\frac{1}{s^r}\frac{P_i(s)}{Q_i(s)} \tag{4.54}$$

where $P_i(s) = (1+s)^{num_i}$ and $Q_i(s) = \prod_{j=1}^{r}\frac{((1+s)^{den_{ij}}-1)}{s}$. The $\frac{P_i(s)}{Q_i(s)}$ polynomial's s^r coefficient is $\lim_{s\to 0}\frac{1}{s^r}\frac{P_i(s)}{Q_i(s)}$. To compute the coefficients

$$\frac{P_i(s)}{Q_i(s)} = c_0 + c_1 s + c_2 s^2 + \dots \tag{4.55}$$

we expand $P_i(s) = a_0 + a_1 s + a_2 s^2 + \dots$ and $Q_i(s) = b_0 + b_1 s + b_2 s^2 + \dots$ and use the following recurrence relation:

$$\begin{aligned} c_0 &= \frac{a_0}{b_0} \\ c_l &= \frac{1}{b_0}(a_l - \sum_{i=1}^{l} b_i c_{l-i}) \end{aligned} \tag{4.56}$$

For our example this means for cone C_1

$$\begin{aligned} P(s) &= 1 \\ Q(s) &= \frac{1-(s+1)}{s} = 1 \\ c_0 &= \frac{a_0}{b_0} = \frac{1}{1} = 1 \\ c_1 &= \frac{1}{b_0}(a_1 - (b_1 * c_0)) = \frac{1}{1}(0 - (0 * 1)) = \underline{0} \end{aligned} \tag{4.57}$$

and for cone C_2

$$\begin{aligned} P(s) &= (s+1)^5 = 1 + 5s + 10s^2 + 10s^3 + 5s^4 + s^5 \\ Q(s) &= \frac{1-(s+1)}{s} = 1 \\ c0 &= \frac{a_0}{b_0} = \frac{1}{1} = 1 \\ c1 &= \frac{1}{b_0}(a_1 - (b_1 * c_0)) = \frac{1}{1}(5 - (0 * 1)) = \underline{5} \end{aligned} \tag{4.58}$$

The sum of the coefficients of the cones is $0 + 5 = 5$ which is the same result as in our first calculation in equation 4.17.

4.2.7 Extension to Linear Parameters and Review

We have described the single steps, how to calculate the enumerator of a non-parameterized polytope using the Barvinok decomposition. Processing linearly parameterized polytopes works essentially the same way, but additionally we keep track of the domain of each vertex and hence each cone. In the final step we sum up cones that lie in the same chamber, for an good example see [VWBC05]. Here is an overview of the algorithm for linear parameterized polytopes:

1. For each vertex v_i of polytope P:
   (a) Determine the supporting cone C_i of P with vertex v_i
   (b) Let C_i^0 be C_i shifted to the origin
   (c) Decompose C_i^0 in unimodular cones C_j^i
      i. Polarize C_i^0 to C_i^*
      ii. Triangulate C_i^* in simplicial cones
      iii. Decompose each simplicial cone into unimodular cones
      iv. Polarize the unimodular cones back to C_j^i

   (d) For each unimodular cone C_j^i determine $f(C_j^i, x)$

   (e) Determine $f(C_i, x)$

2. For each chamber D of polytope P:

   (a) Determine $f(P, x)$

   (b) Evaluate $f(P, 1)$

Further information can be found in Verdoolaege's technical report [VWBC05]. This algorithm overcomes the limitations of Clauss' method (cf. section 4.1):

- No counting of the integer points of non-parameterized polytopes has to be done. Calculating the enumerator of non-parameterized polytopes is proportional to the vertices not the integer points.
- There is no problem with degenerate domains.
- Long periodic numbers can be avoided by using modulo or ceiling expressions.

4.3 Fahringer's Method

During our work we came across another method which was presented by Thomas Fahringer. It estimates the number of integral points in non-linearly parameterized polytopes using symbolical analysis [Fah98, FS03] and is based on an approach by William Pugh [Pug94]. This method has been implemented, but unfortunately, the implementation was lost and, therefore, we could not compute any tests and compare them to our results.
We will provide a simple example to illustrate the techniques used. Instead of computing the vertices of the polytope with Chernikova's algorithm, Fahringer uses a set of rewrite rules and simplifications to compute the lower and upper bounds of the symbolic expressions in the equations and inequalities. An inequality $y \leq x + 1$ would for example define an upper bound $U_y = \{x + 1\}$ for y and a lower bound $L_x = \{y - 1\}$ for x.
To estimate the number of integral points within the polytope a symbolic sum algorithm is applied to the resulting tuples of expressions and bounds. The sum is computed by eliminating the variables one after the other. For each variable, subsets of the original constraints are built where their maximum lower and minimum upper bounds can be determined uniquely and then the variable is replaced in the remaining symbolic expressions by an algebraic sum. For each resulting disjoint constraint subset,

the sums of the according algebraic sums are built. Consider the following inequality system I:

$$\begin{array}{rll} 1 \leq & x & \leq N_1 \\ & x & \leq N_2 \\ 1 \leq & y & \leq x \end{array} \tag{4.59}$$

we have two variables x, y with the bounds $L_x = \{1\}$, $U_x = \{N_1, N_2\}$, $L_y = \{1\}$ and $U_y = \{x\}$. We start with eliminating y, we have only one upper and lower bound and hence only one intermediate result:

$$\sum_{y=1}^{x} 1 = x - 1 \tag{4.60}$$

For x we have one lower and two upper bounds which gives us two subsets with additional constraints, for $N_1 \leq N_2$ we get

$$\sum_{x=1}^{N_1} x - 1 = \frac{N_1(N_1 - 1)}{2} \tag{4.61}$$

and for $N_2 < N_1$

$$\sum_{x=1}^{N_2} x - 1 = \frac{N_2(N_2 - 1)}{2}. \tag{4.62}$$

This simple example gives an impression of the function of Fahringer's technique.

Chapter 5

Volume Computation

We have described several algorithms to calculate the exact number of integral points in linearly parameterized polytopes. Yet many problems concern the volume of non-linearly parameterized polytopes. We have introduced a technique to transform algorithms for non-parameterized polytopes to algorithms for non-linearly parameterized polytopes in section 3.4.2. In this section we will show how to compute the volume of a non-linearly parameterized polytope as a rough approximation to its enumerator. Of course the volume may extremely differ from the enumerator, but in general cases it is sufficiently accurate for many applications.
In the following we will explain a method to compute the volume of non-parameterized polytopes and then show our implementation of this algorithm for non-linearly parameterized polytopes.

5.1 Algorithm for the Volume of Polytopes

There are several techniques for computing the exact volume of a polytope. A good survey can be found in [GK94]. We implemented a very simple technique which is based on triangulation and is taken from [Fah96].
The algorithm triangulates a polytope into a set of simplices and then then sums up the volumes of the simplices. First, we will give a formula for calculating the volume of a simplex.

5.1.1 Volume of a Simplex

Definition 5.1 (Simplex) A simplex is an n-dimensional convex polytope with $n + 1$ vertices.

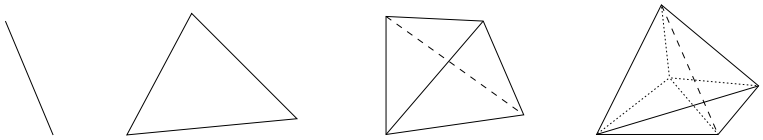


Figure 5.1: Simplices of Dimension 1-4.

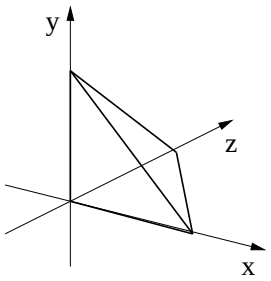


Figure 5.2: Tetrahedron

Therefore a simplex of dimension 1 is a line segment, one of dimension 2 is a triangle and one of dimension 3 is a tetrahedron, which can be seen in figures 5.1 and 5.2. The volume of a simplex is given by a closed form formula.

Definition 5.2 (Volume of a simplex) Let P be an n-dimensional simplex and $V = \{v_1, \ldots, v_{n+1}\}$ the set of vertices of P then its volume is defined by

$$vol(P) = \frac{1}{n!}|det(v_2 - v_1, v_3 - v_1, \ldots, v_{n+1} - v_1)| \tag{5.1}$$

where $v_i - v_j$ the distance vector of two vertices is.

This formula can even be applied to parameterized simplices without change, since we can compute the determinant also with a parametrical matrix. However, it may lead to one case distinction because of the absolute. Consider the simplex shown in figure 5.2. It may be defined by the set $P = \{(x, y, z) \in \mathbb{R}^3 | x \geq 0, y \geq 0, z \geq 0, x + y + z \leq p, p \geq 0\}$ where p is a rational parameter. The set of vertices is $V = \{(0, 0, 0), (p, 0, 0), (0, p, 0), (0, 0, p)\}$ and the formula gives the volume:

$$vol(P) = \frac{1}{3!} \left| det \begin{pmatrix} p & 0 & 0 \\ 0 & p & 0 \\ 0 & 0 & p \end{pmatrix} \right| = \frac{1}{6}p^3 \tag{5.2}$$

5.1.2 Triangulation

We have already seen a form of triangulation in section 4.2.3. Here we will see another technique which makes use of the center of mass vertex. The center of mass vertex of a polytope is defined as follows:

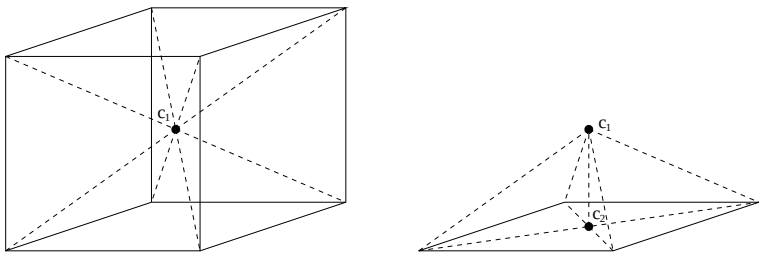


Figure 5.3: Triangulation of a Cube

Definition 5.3 (Center of mass vertex) Let P be a polytope of dimension d, with a set of vertices V, then

$$v_c = \frac{1}{|V|} \sum_{v \in V} v \tag{5.3}$$

is the center of mass vertex of P.

To triangulate a polytope P of dimension n with the center of mass vertex, we use an recursive algorithm. Here in pseudo code:

```
triangulate (Polyhedron p, Dimension d, Vertices v)
  simplices = EmptSet;
  IF d>1
    c = centerOfMass(p);
    v = v UNION c;
    FORALL facets f of p do
      simplices = simplices UNION triangulate(f,d-1,v)
  ELSE
    simplices = {p UNION v};
  FI
  RETURN simplices;
END.
```

The initial input is a set of vertices p of the polyhedron to be computed, the dimension of the polyhedron and an empty set. The output is a set of sets of vertices, where each defines a simplex. Informally we do the following: We calculate the set of facets F of P and decompose P in $|F|$ new polytopes, where each is the convex hull of the vertices the facet $f \in F$ and the polytopes center of mass v_m. Then we do this for each of the facets recursively, until we have only 1-dimensional facets, (i.e. lines) left. At this point, we have $n-1$ center of mass vertices to each line. With the line vertices we have $n+1$ vertices that define a simplex. The set of all simplices is a triangulation of P.
Consider a 3-dimensional cube; its triangulation can be seen in figure 5.3. On the left we see the first recursive step, the center of mass c_1 of the whole cube is calculated and the cube is decomposed in 6 square pyramids. On the right we see the next for one of

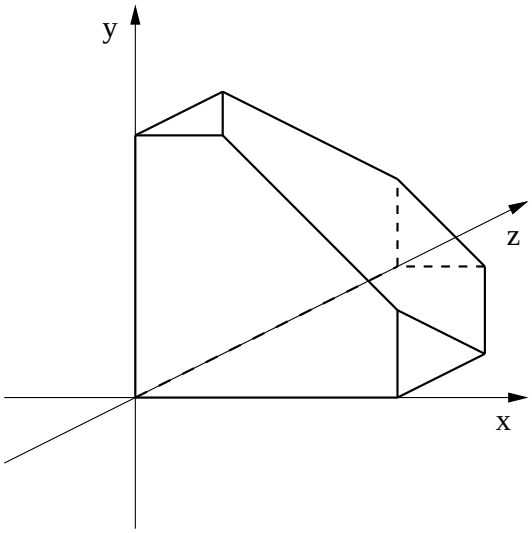


Figure 5.4: 3-dimensional Polytope

these pyramids. The center of mass c_2 of the according face is calculated and is split up into 4 triangles. As lines do not have to be triangulated, we can stop here and have a valid triangulation of the cube.
This triangulation is by far not minimal in the number of resulting simplices, for it will also decompose simplices. Nevertheless, it is easy to adopt to parameterization, which we will see later (section 5.3).

5.1.3 Volume

With the explicit formula for simplices and the triangulation the volume computation is straightforward. We simply triangulate any polytope and then sum up the volumes of the resulting simplices. In pseudo code:

```
volume(Polyhedron p, dimension k)
  simplices = triangulate(p,k,{});
  sum = 0;
  forall s in simplices do
    sum = sum + simplexVolume(s);
  od
  return sum;
end.
```

We will now have a look at an example concerning the polytope

$$P = \{(x, y, z) \in \mathbb{R}^3 | 0 \leq x \leq 3, 0 \leq y \leq 3, 0 \leq z \leq 3, x + y + z \leq 4\}, \quad (5.4)$$

which can be seen in figure 5.4. With Chernikova's algorithm (section 3.2) we can get its set of vertices

$$V = \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 3 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 3 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 3 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 3 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 3 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 3 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 3 \end{pmatrix} \right\}. \tag{5.5}$$

Now we calculate the center of mass c_1 of the whole polyhedron

$$c_1 = \frac{1}{|V|} \sum_{v \in V} v = \begin{pmatrix} 11/10 \\ 11/10 \\ 11/10 \end{pmatrix}. \tag{5.6}$$

In order to decompose the polyhedron, we have to identify the facets and the according vertices. Each facet is defined by the set of the vertices that lie on one supporting hyperplane. We know the supporting hyperplanes from the constraints. A vertex lies on a hyperplane if the scalar product of the vertex and the hyperplane is zero (both in homogeneous form).
We can demonstrate this for the constraint $a_7 := x + y + z \leq 4$. The supporting hyperplane is defined by the equality $x + y + z = 4$ and the set of vertices that lies on that hyperplane is

$$F_7 = \left\{ \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 3 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 3 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 3 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 3 \end{pmatrix} \right\}. \tag{5.7}$$

We do this for all constraints and get 7 facets, this can be seen in figure 5.5 on the right side. For each of those facets we again compute the center of mass. We do this for the facet we computed above,

$$c_2 = \frac{1}{|F|} \sum_{v \in F} v = \begin{pmatrix} 4/3 \\ 4/3 \\ 4/3 \end{pmatrix}. \tag{5.8}$$

Now we have to find all facets of F. This is done as before by finding all sets of vertices $F^* \subset F$, that lie on a supporting hyperplane of F. The result can be seen in figure 5.5 on the left. We find 6 supporting hyperplanes and therefore get 6 lines. With the centers of mass c_1 and c_2 each line defines a simplex, for which we can compute the volume with the formula in equation 5.1. We will do this for example for the simplex

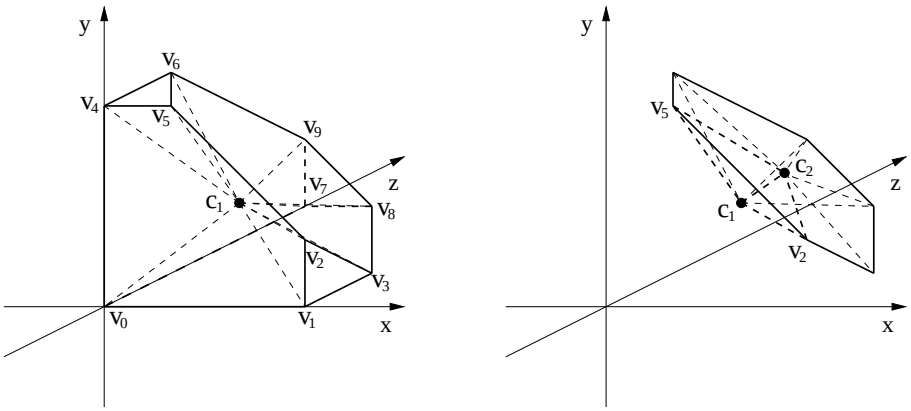


Figure 5.5: Triangulation of a Polytope.

with vertices $F^* = \{c_1, c_2, v_2, v_5\}$. The volume of this simplex is

$$volume(F^*) = \frac{1}{n!} \left| det(c_2 - c_1, v_2 - c_1, v_5 - c_1) \right| = \frac{1}{6} \left| det \begin{pmatrix} 7/30 & 19/10 & -1/10 \\ 7/30 & -1/10 & 19/10 \\ 7/30 & -11/10 & -11/10 \end{pmatrix} \right| = \frac{14}{45} \tag{5.9}$$

If we do this for all simplices, we get the volume of P which is $25\frac{2}{3}$.

5.2 Comparison of Volume and Number of Integral Points

We already mentioned that the real volume of a polytope can only be a very rough estimate of the number of its integral points. But, nevertheless, for many applications it is, in most cases, sufficient.
We have to be aware of two extremal cases:

- the volume of a non-full-dimensional polytope is always zero,
- a polytope with a width of less than one in one dimension can contain zero points.

The first case is very common, but we excluded it intentionally before. If we have to deal with polytopes of lower dimensions, we can project them to a space with proper dimensionality. The second case is harder to predict, especially in parameterized polytopes.
The difference between the volume and the enumerator of a polytope becomes

less significant with increasing size. As the main application of our technique is parallelization, we must expect large problem sizes. We will illustrate this in a sample calculation: Consider a hypercube of dimension d with integral edge length x and integral vertices, its volume is x^d and its enumerator is $(x+1)^d$. The deviation of the volume from the enumerator in percent is shown as $d \in \{1,2,3,4\}$ and $x \in \{1, 10, 1000, 1000000\}$ in the following tabular:

d	$x=1$	$x=10$	$x=1000$	$x=1000000$
1	50	9.1	0.001	0.000001
2	75	17	0.002	0.000002
3	88	25	0.003	0.000003
4	93	32	0.004	0.000004

As we can see is the deviation very small for large polytopes. For the problem sizes that we expect it is negligible (see also [Fah96]).

5.3 Non-Linear Parameterized Volume Computation

5.3.1 Changes on the Algorithm

For computing the volume of a non-linearly parameterized polytope we will once again use the technique of Armin Größlinger, which we introduced in section 3.4.2. It will give us the result in form of a decision tree, where each node indicates a case distinction. At first we have to identify whereabouts in the algorithm we need to introduce case distinctions.
The algorithm above needs both the implicit and parametric representation of a polytope. As we only (in general) have one, we use our implementation of Chernikova's algorithm to get the other. This step will produce several case distinctions (see section 3.4.2).
For computing a polytope's volume, we have to triangulate it first. The triangulation consists of 3 recursive steps (except the last recursion):

- Compute center of mass vertex.
- Compute facets.
- Triangulate facets.

The computation of the center of mass vertex results in no new case distinctions since it is a closed formula, which we can also use with parameters. The computation of the

facets gives us new case distinctions, since we have to check if a vertex lies on a hyperplane. For every combination of any hyperplane and vertex we get a $EqCond$ node, but if we use Chernikova's algorithm these case distinctions should be redundant and lead to no new nodes. The triangulation of the facets is the recursive step and nothing is changed here. In the last recursion the simplex is returned and here also nothing has to be done.
When we triangulated the polytope, we have to compute and then sum up the simplices' volumes. For the calculation of the volumes we use the formula in equation 5.1. As we indicated before, this results in one case distinction for each simplex, since we have to calculate an absolute value, so we introduce a $GeCond$ node here. The summation can be done without change.

5.3.2 Implementation Details

We implemented the volume computation algorithm for non-linearly parameterized polytopes. The implementation is very similar to the pseudo code in section 5.1.3, as before we used the Haskell programming language and data structures and functions provided by HsLooPo, a part of the LooPo project [Leh]. We will first explain some auxiliary functions and then show the algorithm itself.
The first function computes the volume of a simplex:

```
simplexVolume vertices =
  let
    -- an arbitrary vertex is chosen as root vertex
    -- and negated for substraction
    negRoot = negV (head vertices)

    -- the root vertex is subtracted from all the other vectors,
    -- and a matrix is built from the resulting distance vectors
    vectors = map (addVV negRoot) (tail vertices)
    vMatrix = rowVectorsToMatrix1 vectors

    -- the determinant of the matrix is computed and divided
    -- by the faculty of the number of vertices of the simplex
    det     = determinantFP (rows' vMatrix)
    divBy   = fac (length vertices)
    result  = det / (int2FracPoly divBy)

    -- to compute the absolute value of the result a geCond
    -- node is introduced
    absResult = geCond result (dLeaf result) (dLeaf (-result))
  in absResult
```

It is a straightforward implementation of the formula for simplex volumes, input is a list of vectors, containing the simplexes vertices. Auxiliary functions are:

`negV v` computes the negative of a vector v.

`addVV v1 v2` adds two vectors `v1` and `v2`.

`rowVectorsToMatrix1 vs` builds a matrix with row vectors from a list of vectors `vs` (it should be the column matrix, but it makes no difference, because the determinant is equal for a matrix and its transpose).

`determinantFP m` computes the determinant of a matrix `m` (we use the Laplace formula to simplify matters).

`fac n` computes the faculty of `n`.

`int2FracPoly x` converts the type of a variable from `Int` value to `FracPoly`.

The result of the function is not a polynomial, but a decision tree, with parameterized polynomials at its leafs.
The next function computes the center of mass vertex of a polytope, input is a list of its vertices:

```
centerOfMass vertices =
  let
    divBy = 1 / (fromIntegral (length vertices))
    vertexSum = foldl addVV (head vertices) (tail vertices)
    result    = mulVS vertexSum divBy
  in result
```

This function does nothing special, but computing the center of mass vertex with the formula presented in equation 5.3.
In our implementation, the triangulation and volume summation are not separated, so we don't have to additionally iterate over the list of simplex volumes:

```
volume hyperplanes vertices n centersOfMass =
  -- if dimension 1 is reached volume of the simplex is computed
  if n == 1 then
    let
      allVertices = vertices++centersOfMass
      result = simplexVolume allVertices
    in result

  -- the polytope has to be further decomposed
  else
    let
      -- the center of mass of the polytope defined by vertices is computed
      -- and is added to the list of center of mass vertices
      centerVertex = centerOfMass vertices
      newCentersOfMass = centersOfMass++[centerVertex]

      -- the facets of the polytope defined by vertices are computed
      -- and the volume function is called for all facets
      facets = getFacets hyperplanes vertices
      volumes = map (\x -> volume hyperplanes x (n-1) newCentersOfMass) facets

      -- the sum of all simplices volumes is computed and returned
      sumOfVolumes = mapDTreeG (foldl (+) 0) (prodDTreeG volumes)
    in sumOfVolumes
```

Initial input of this function is the set of supporting hyperplanes, a list of the vertices, the dimension of the polytope and an empty list which will hold the center of mass vertices in later recursions. We will discuss the representation of the hyperplanes later. As in the pseudo code version we have two cases, either dimension 1 is reached and the vertices in `vertices` combined with the center of mass vertices in `centersOfMass` define a simplex, or we have to further triangulate the polytope.
If `n` is 1, we simply unite `vertices` and `centersOfMass` and compute the volume of their convex hull. If we have to further triangulate the polytope, we calculate the center of mass vertex and the facets of the polytope. For each facet we call the volume function recursively and sum up the results. We use following auxiliary functions:

`mapDTreeG f dt` applies a function `f` to every leaf of the decision tree `dt`

`prodDTreeG dts` combines a list of decision trees `dts` so, that on the combined trees leafs lists of the leaf values of the old trees are.

Additionally, we use the function `getFacets hs vs`, which returns the facets of the polytope with vertices `vs`, hyperplanes `hs` in a decision tree. The facets are specified by a list of their vertices. This function needs some extra consideration. To find a facet we first have to find the supporting hyperplane and then the vertices on that hyperplane. If we do this for each recursion, we have many redundant computations, so it is more advisable to initially compute the set of vertices for each hyperplane and implement the `getFacet` function simply by intersection. Let us first consider the functions to build the set of vertices to each hyperplane:

```
-- Filters a list of vertices, that lie on a hyperplane
-- Returns a DTreeGen with the corresponding lists at the leaves
verticesOnHyperplane _ [] = dLeaf []
verticesOnHyperplane hyp (v:vs) =
  let
    -- the vertex is transfered to vector representation
    -- and multiplied with the hyperplanes vector
    vector = insertEntries v (vectorLength v) [(int2FracPoly 1)]
    prod = mulVV hyp vector

  -- if the scalar product is null the vertex lies on the hyperplane
  -- else it does not
  in EqCond prod
      (mapDTreeG (\x -> x++[v]) (verticesOnHyperplane hyp vs))
      (verticesOnHyperplane hyp vs)

-- Applies the verticeOnHyperplane function on a list of hyperplanes
-- Returns a list of DTreeGens
_verticesOnHyperplanes [] _ = []
_verticesOnHyperplanes (hyp:hs) vs =
  [verticesOnHyperplane hyp vs] ++(verticesOnHyperplanes hs vs)

-- Joins the list of DTreeGens to a DTreeGen with lists at it's leaves.
verticesOnHyperplanes hs vs =
  prodDTreeG _verticesOnHyperplanes hs vs
```

With the function `verticesOnHyperplanes hs vs` we get a decision tree containing a subset of the vertices in `vs` for each hyperplane in `hs` at each leaf. We can now apply the volume calculation to each leaf of the decision tree. This representation of the hyperplanes gives us a very easy implementation of the `getFaces` function:

```
-- Computes a list of lower dimensional facets of a polytope
-- by intersection
getFacets hyperplanes vertices dim =
  let
    faces = map (\x -> intersect vertices x ) hyperplanes
    facets = filter (\x -> (length x) < (length vertices)
        && (length x) >= (dim -1)) faces
  in facets
```

We only have to intersect the hyperplanes' vertices with the vertices of the current polytope. We then filter the result, because if the polytope lies within a hyperplane, this hyperplane will give us the complete polytope again.

Chapter 6

Conclusions

In parallel computing it is essential that programs have a good work distribution. Automatic parallelization techniques therefore need information on the overall amount of work in an program. In the polyhedron model this can be expressed as the number of integral points in a polytope.
We have presented a method to estimate this number for non-linearly parameterized polytopes. Our approximation of the enumerator of a polytope is its volume. This is a good approximation in most cases (cf. [Fah96]). We also implemented our method as part of the HsLooPo project.
Our implementation is based on Armin Größlinger's decision tree data structure [Grö03]. It does much of the computational work concerning the case distinctions. Since this is very complex, the overall runtime is dominated by the decision making. Therefore, our technique addresses exclusively non-linear problems. For linear problems to date the technique by Sven Verdoolaege et al. is best in performance and applicability [VSB$^+$04].
Thomas Fahringer and Bernhard Scholz presented an equally expressive technique [Fah98]. Unfortunately, we could not compare our results, since their implementation was lost and they no reference was made to its complexity.
A large part of our implementation was the adaption of Chernikova's algorithm for non-linearly parameterized polyhedra. We discussed it extensively because the double description is not only used for volume and enumerator computation, but is also the basis for several operations performed on polyhedra, such as intersection, difference and union. As it is one goal of the HsLooPo project to completely extend the polyhedron model to non-linear parameterized polyhedra, this algorithm may well be needed in different applications in the future. Like the volume computation, it also is highly dependent on the decision tree data structure and, therefore, only suitable for non-linear problems.
An overview of the running time of some of our experiments can be seen in table 6.1. The first test (eq. 3.32) was a linearly parameterized polytope with a few more case

Polytope	#Vars	#Params	#Tree Nodes (C)	Chernikova	Volume
Eq. 3.32, p. 21	2	1	15	1.09 sec.	2.08 sec
Eq. 3.47, p. 30	2	1	12	1.02 sec.	1.31 sec.
Eq. 4.59, p. 58	2	2	38	10.7 sec.	18.9 sec.

Table 6.1: Runtime Experiments

distinctions than the polytope in the second test (eq. 3.47). The longer, but almost equal, runtime of first test shows that linear parameterization leads to the same amount of computation as non-linear parameterization. The third test shows how fast the complexity of non-linear parameterization grows.

Further improvements could address the inefficiency of linear and unparameterized problems by providing alternative implementations for these. Another issue is the way equations are handled in the volume computation. In our current implementation the exact volume is computed, and so an equation will always result in the volume being zero. This is correct, but in this case not desirable. One solution would be to project the polytope to a lower dimensional space, so that only the equation is eliminated.

Bibliography

[Bar93] Alexander I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. In *34th Annual Symposium on Foundations of Computer Science*, pages 566–572. IEEE, Nov. 1993.

[Bar94] Alexander I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Math. Oper. Res.*, 19(4):769–779, 1994.

[Bey04] Kristof Beyls. *Software Methods to Improve Data Locality and Cache Behavior*. PhD thesis, 6 2004.

[BHRZ03] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. *The Parma Polyhedra Library User's Manual*. Department of Mathematics, University of Parma, Parma, Italy, April 2003.

[BP99] A. Barvinok and J. Pommersheim. An algorithmic theory of lattice points in polyhedra, 1999.

[Bri88] Michel Brion. Points entiers dans les polyèdres convexes. *Annales Scientifiques de l'École Normale Supérieure*, 4, 21(4):653–663, 1988.

[Che64] N.V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear equations. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 5(2):167–185, 1964.

[Che65] N.V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 5(2):228–233, 1965.

[Che68] N.V. Chernikova. Algorithm for discovering the set of all the solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.

[Cla96] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *International Conference on Supercomputing*, pages 278–285, 1996.

[Ehr77] Eugene Ehrhart. Polynômes arithmétiques et méthode des polyèdres en combinatoire. In *International Series of Numerical Mathematics*, volume vol.35. Birkhauser Verlag, Basel/Stuttgart, 1977.

[Fah96] Thomas Fahringer. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[Fah98] Thomas Fahringer. Efficient symbolic analysis for parallelizing compilers and performance estimators. *The Journal of Supercomputing*, 12(3):227–255, 1998.

[FS03] Thomas Fahringer and Bernhard Scholz. *Advanced Symbolic Analysis for Compilers*, volume 2628 of *Lecture Notes in Computer Science*. Springer, 2003. New Techniques and Algorithms for Symbolic Program Analysis and Optimization.

[GGL04] Armin Größlinger, Martin Griebl, and Christian Lengauer. Introducing non-linear parameters to the polyhedron model. In Michael Gerndt and Edmond Kereku, editors, *Proc. 11th Workshop on Compilers for Parallel Computers (CPC 2004)*, Research Report Series, pages 1–12. LRR-TUM, Technische Universität München, July 2004.

[GK94] Peter Gritzmann and Victor Klee. On the complexity of some basic problems in computational convexity: II. volume and mixed volumes. *Universität Trier, Mathematik/Informatik, Forschungsbericht*, 94-07, 1994.

[Gri00] Martin Griebl. On the mechanical tiling of space-time mapped loop nests. Technical Report MIP-0009, Fakultät für Mathematik und Informatik, Universität Passau, August 2000.

[Grö03] Armin Größlinger. Extending the polyhedron model to inequality systems with non-linear parameters using quantifier elimination. Diploma thesis, Universität Passau, September 2003.

[Leh] Lehrstuhl für Programmierung, Universität Passau. *The polyhedral loop parallelizer: LooPo*.

[LJL82] A. Lenstra, H. Lenstra Jr., and L. Lovasz. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[LW97] Vincent Loechner and Doran K. Wilde. Parameterized polyhedra and their vertices. *International Journal of Parallel Programming*, 25(6):525–549, 1997.

[MR80] T.H. Matheiss and David S. Rubin. A survey and comparison of methods for finding all vertices of convex polyhedral sets. *Mathematics of operations research*, 5(2):167–185, May 1980.

[MRTT53] T.S. Motzkin, H. Raiffa, G.L. Thompson, and R.M. Thrall. The double description method. *Contributions to the Theory of Games*, 2(28):51–73, 1953.

[Pug94] William Pugh. Counting solutions to presburger formulas: How and why. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–134, 1994.

[Sch86] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[Sho04] Victor Shoup. Ntl: A library for doing number theory, 2004. www.shoup.net/ntl.

[Ver94] H. Le Verge. A note on chernikova's algorithm. Technical Report 635, Rennes, France, 1994.

[VSB+04] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Analytical computation of ehrhart polynomials: enabling more compiler analyses and optimizations. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 248–258, New York, NY, USA, 2004. ACM Press.

[VWBC05] Sven Verdoolaege, Kevin Woods, Maurice Bruynooghe, and Ronald Cools. Computation and manipulation of enumerators of integer projections of parametric polytopes. Technical report, Katholieke Universiteit Leuven, Heverlee - Belgium, March 2005.

[Wil93] D. K. Wilde. A library for doing polyhedral operations. Technical Report RR-2157, 1993.

Index

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich diese Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet habe, sowie dass diese Diplomarbeit in gleicher oder in ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Passau, 30. Januar 2006

(Tilmann Rabl)