Master Thesis

Extreme points in medium and high dimensions

Author(s):
Helbling, Christian

Publication Date:
2010

Permanent Link:
https://doi.org/10.3929/ethz-a-006250404

Rights / License:
In Copyright - Non-Commercial Use Permitted
Extreme Points in Medium and High Dimensions

Master Thesis
Christian Helbling
November 17, 2010

Advisors: Dr. B. Gärtner, Prof. Dr. E. Welzl
Department of Computer Science, ETH Zürich
Abstract

The problem of finding the extreme points of a point set in higher dimensions is used in different applications and algorithms. In contrast to constructing the convex hull, this problem can be solved in polynomial time using linear programming. The practical part of this thesis is the implementation of an extreme point package for the CGAL library. In the theoretical part the special case of sparse points in high dimensions and the properties of an on-line setting are examined.

Keywords: Extreme points, linear programming, computational geometry, redundancy identification, convex hull, CGAL
Acknowledgements

I’d like to thank my advisor Bernd Gärtner for supporting me throughout the whole time in many ways.

I’d also like to thank all my friends for constantly keeping my moral up. Especially my house mates with whom I enjoyed many delicious dinners during this time.

And finally a big thank to Reto Conconi, Marc Bruggmann and Remo Gisi for proofreading the thesis.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Preliminaries</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Linear Programming</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 Algorithms</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2 Infeasibility Certificates</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Convex Combination Test</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Separating Hyperplanes</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Homogeneous Coordinates</td>
<td>10</td>
</tr>
<tr>
<td>2.5 Lexicographical Ordering</td>
<td>13</td>
</tr>
<tr>
<td>3 Algorithms</td>
<td>15</td>
</tr>
<tr>
<td>3.1 Simple Algorithm</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Dulá-Helgason Algorithm</td>
<td>16</td>
</tr>
<tr>
<td>4 Sparse Input</td>
<td>21</td>
</tr>
<tr>
<td>4.1 Representation of Sparse Points</td>
<td>21</td>
</tr>
<tr>
<td>4.2 One-Sparse Points</td>
<td>22</td>
</tr>
<tr>
<td>4.3 Preprocessing</td>
<td>24</td>
</tr>
<tr>
<td>4.4 The Dimension Graph</td>
<td>26</td>
</tr>
<tr>
<td>4.5 Sparse LP-Solvers</td>
<td>27</td>
</tr>
<tr>
<td>4.6 Divide and Conquer</td>
<td>27</td>
</tr>
<tr>
<td>5 On-Line Problem</td>
<td>29</td>
</tr>
<tr>
<td>5.1 Ignoring Non-Extreme Points</td>
<td>29</td>
</tr>
<tr>
<td>5.2 One New Point</td>
<td>32</td>
</tr>
<tr>
<td>5.3 Two or More New Points</td>
<td>33</td>
</tr>
<tr>
<td>5.4 Lots of New Points</td>
<td>36</td>
</tr>
</tbody>
</table>
6 Implementation 37
   6.1 CGAL Package ........................................... 38
      6.1.1 CGAL’s Philosophy .................................. 38
      6.1.2 Static Point Set .................................... 38
      6.1.3 On-Line Setting ..................................... 38
   6.2 No General Position Assumption ............................ 39
      6.2.1 Simple Algorithm .................................... 39
      6.2.2 Dulá-Helgason Algorithm ............................ 40
   6.3 Possible Improvements .................................... 40
      6.3.1 LP Hot Starts ......................................... 40
      6.3.2 Floating-point Filtering Technique .................... 41

7 Experimental Results 43
   7.1 Synthetic Data from Dulá and López ......................... 43
   7.2 Banking Data from Dulá and López .......................... 46
   7.3 Cyclic polytopes .......................................... 46

8 Conclusion 49

A CGAL Package Manuals 51
   A.1 User Manual ............................................. 51
   A.2 Reference Manual ......................................... 57

B Runtime Measurements 71

Bibliography 75
We start by defining the terms convex hull, extreme point and frame of a point set.

**Definition 1.1 (Convex hull)** The convex hull of some set $S \subseteq \mathbb{R}^d$ is the set of all finite convex combinations of $S$. Formally:

$$\text{conv}(S) := \left\{ \sum_{p \in S'} \lambda_p p \left| S' \subseteq S \text{ finite, } \lambda_p \geq 0 \text{ for each } p \text{ and } \sum_{p \in S'} \lambda_p = 1 \right\}$$

**Definition 1.2 (Extreme point, frame of a point set)** A point $p$ among a set of points $A$ in $d$ dimensions is called extreme if it is not contained in the convex hull of the other points.

A point which is not extreme (i.e. which is a convex combination of the other points) is called non-extreme.

The set of extreme points of $A$ is a subset of $A$ and is called the frame of $A$ and will be denoted by $\text{frame}(A)$. So formally we have:

$$p \in \text{frame}(A) \iff p \in A \text{ and } p \notin \text{conv}(A\setminus\{p\})$$

As a simple example, Figure 1.1 shows a set of 2-dimensional points where the extreme points are encircled and the convex hull is shaded.

The topic of this thesis is the problem of finding the frame of a $d$-dimensional point set. In particular the following aspects are covered:

**Sparse Input** In this scenario most of the coordinates of the input points are zero. In our model, any point may have at most a constant number $k$ of nonzero coordinates. The question is whether this problem is substantially easier than the dense problem where all $d$ coordinates of any point can be nonzero.
1. Introduction

Figure 1.1: Example: Convex Hull and Extreme Points in 2d

On-Line Problem In this case the frame of a growing point set is calculated repeatedly. Here we look at how results of previous computations can be exploited to speed up the next extreme point computation.

Implementation The practical part of the thesis is the implementation of two extreme point algorithms. Its focus lies on output-sensitivity and exact computation in the dense case. Some aspects of the on-line problem were also implemented. The code was written as a package for the CGAL library [1].

Applications

Finding the frame of a point set is useful in many applications. We only present a short listing here:

- Finding redundant constraints in linear programs
- Data Envelopment Analysis (DEA)
- Computational geometry
- Identifying outliers in multivariate statistical data
- Stochastic optimization

A more detailed description of these applications can be found in [10, p. 353ff].

Relation to the Convex Hull Problem

The problem of computing the convex hull of a $d$-dimensional point set is closely related to the extreme point problem. The connection is evident: the
extreme points are exactly the vertices of the convex hull. Therefore a way
to find the extreme points is to compute the convex hull first and then take
its vertices. In two and three dimensions this is in fact the way to go as
efficient $O(n \log(n))$ convex hull algorithms exist in 2d and 3d. However,
this approach does not scale to higher dimensions. The problem is that the
complexity of the convex hull itself can grow exponentially in the dimension.

More precisely, the convex hull of $n$ points in $d$ dimensions can have up to
$n^{[d/2]}$ facets (see for example [19, p. 15f] and the original Upper Bound The-
orem from McMullen [15]). Hence all convex hull algorithms are forced to
have an exponential worst-case runtime. Worst-case optimal, $d$-dimensional
convex hull algorithms exist. An example is Seidel’s Algorithm [17]. How-
ever, it is an open problem, whether there exists an output-sensitive algo-
rithm for the $d$-dimensional convex hull of any $d$-dimensional point set.
The Reverse-Search Algorithm from Avis and Fukuda [4] would be such
an output-sensitive algorithm but cannot handle degeneracies efficiently [3].

The reason that the algorithms which can handle degeneracies are not output-
sensitive is, that they operate on an incremental construction of the convex
hull. This means, that some representation of a partial hull has to be kept
internally and the complexity of this partial hull could be much worse than
the complexity of the final convex hull. So the exponential worst-case time
is not only due to a big output size, but also due to the complexity of in-
termediate results. Therefore we cannot get an efficient algorithm which
can also handle degeneracies by just modifying a $d$-dimensional convex hull
algorithm to output extreme points only.

On the other hand, there exists a very simple polynomial time frame algo-
rithm based on linear programming (see Chapter 3). So the two problems
are really distinct as the frame problem is polynomial-time solvable and the
convex hull problem not.

**Previous Work**

Dulá and Helgason presented in [10] an output-sensitive frame algorithm
which we will also be using in this thesis. Another output-sensitive algo-
rithm was found by Ottmann et al [16]. Chan improved this algorithm to
compute the $h$ extreme points of a point set with $n$ points in $O(n \log^{O(1)} h +
(nh)^{1-1/[d/2]+1} \log^{O(1)} n)$ time for any constant dimension [6].

As for the sparse frame problem, no results are known so far.
1. Introduction

**Structure of the Thesis**

Following this introduction, some important concepts needed throughout the whole thesis are dealt with in Chapter 2. Then two of the known frame algorithms are presented in Chapter 3. Chapter 4 discusses the case of sparse input data and Chapter 5 the case of an on-line setting. The implementation of the CGAL package will be outlined in Chapter 6. Finally, experimental results are presented in Chapter 7.
Chapter 2

Preliminaries

This chapter deals with some key concepts which are needed for the rest of the thesis. It starts with a short introduction of linear programming, which is the foundation of all algorithms we are dealing with. Based on that, a notation for convex combination tests is introduced, followed by the concept of so called separating hyperplanes. Furthermore, homogeneous coordinates are analyzed and finally lexicographical ordering is discussed.

2.1 Linear Programming

Linear programming is the problem of optimizing a linear function subject to a finite number of linear constraints. The type of linear programs we will be dealing with is the following, where \( x \in \mathbb{R}^n \), \( c \in \mathbb{R}^n \), and \( b \in \mathbb{R}^m \) are column vectors and \( A \in \mathbb{R}^{m \times n} \) is a matrix. The vector \( x \) is the unknown.

\[
\begin{align*}
\text{minimize} & \quad c^T x \\
\text{subject to} & \quad Ax = b \\
& \quad x \geq 0
\end{align*}
\]

(2.1)

Note that \( x \geq 0 \) means component-wise comparison (i.e. \( x_i \geq 0 \) for all \( 1 \leq i \leq k \) where \( k \) is the number of elements of \( x \)). We will use this notation throughout the thesis without further notice.

A solution \( x \) which satisfies the constraints is called a feasible solution. The function \( x \mapsto c^T x \) is called the objective function. In our case, the objective function can be ignored as we are only interested whether a feasible solution exists. Therefore we will just take the constant 0 as our objective function which implies that any feasible solution is an optimal solution. If there is no feasible solution to a linear program it is called infeasible.
As the term *linear program* comes up often, we will sometimes use the popular abbreviation *LP* for it.

We continue with a brief overview of algorithms used for solving linear programs followed by a short discussion about proving the nonexistence of a feasible solution using infeasibility certificates. Of course we cannot go into the full details here. For more comprehensive information on this topic we recommend the book *Linear Programming* from Chvátal [7].

### 2.1.1 Algorithms

The first algorithm to solve linear programming in general was Dantzig’s Simplex algorithm from 1947 [8]. Variations of it are still the most popular way of solving linear programs as they generally perform quite well in practice. However, worst-case examples with exponential runtime have been found for many of them [2, p. 3].

In 1979 Khachiyan proved the problem to be polynomial time solvable by using the so called ellipsoid method. Five years later, a significant step was made by Karmarkar who found an $O(n^{3.5}L)$ algorithm where $n$ is the number of variables and $L$ the number of bits in the input [14]. This improved the ellipsoid algorithm by a factor of $O(n^{2.5})$.

Hence, polynomial time algorithms for solving linear programs exist. However, the algorithms used more often have exponential worst-case runtimes but perform generally better in practice. This somewhat strange situation also reflects itself in this thesis as we will recognize linear programming as polynomial time solvable in the theoretical part and at the same time use a linear programming solver based on the Simplex algorithm in the implementation part.

To account for different LP-solvers we will use the following abstraction for runtime analysis.

**Definition 2.1 (Runtime for solving a linear program)** The expression $LP_{k,n}$ stands for the runtime used to solve a linear program with $k$ equality constraints and $n$ nonnegative variables.

### 2.1.2 Infeasibility Certificates

An easy way to prove that a linear program is not feasible are infeasibility certificates.

**Definition 2.2 (Infeasibility certificate)** An infeasibility certificate of a linear program in form (2.1) is a column vector $y \in \mathbb{R}^n$ which satisfies $A^T y \geq 0$ and $b^T y < 0$. 

2.2 Convex Combination Test

Farkas’ Lemma [12] proves that an infeasibility certificate exists if and only if the corresponding linear program is infeasible.

That an infeasibility certificate \( y \) proves the infeasibility of a linear program in form (2.1) is actually easy to see:

For all \( x \in \mathbb{R}^n \) with \( x \geq 0 \) we have:

\[
y^T A x \geq 0
\]

As all elements of \( y^T A \) and \( x \) are nonnegative. However, from the infeasibility we additionally know:

\[
y^T b < 0
\]

Therefore \( Ax \neq b \) for all candidates \( x \), which means that there is no feasible solution.

Most LP-solvers can output such an infeasibility certificate in case the linear program is not feasible. We will see that infeasibility certificates come handy in the algorithm from Dulá and Helgason as they can be used to get separating hyperplanes (see Section 2.3).

2.2 Convex Combination Test

A fundamental operation of our extreme point algorithms is to test whether some \( d \)-dimensional point \( p \) is a convex combination of the points \( A = \{a_1, a_2, ..., a_n\} \). This test can be done by solving the following linear program (with \( d + 1 \) equality constraints, \( n \) nonnegative variables and constant objective function).

\[
\begin{align*}
\sum_{i=1}^{n} a_i x_i &= p \\
\sum_{i=1}^{n} x_i &= 1 \\
x &\geq 0
\end{align*}
\] (2.2)

Let us introduce an abbreviation for this type of linear program as we will use it a lot.

**Definition 2.3 (Convex combination test)** For a set of points \( A = \{a_1, a_2, ..., a_n\} \) and a point \( p \), the expression \( \text{LP}(A, p) \) refers to the linear program (2.2).
2. Preliminaries

2.3 Separating Hyperplanes

A separating hyperplanes is defined like this:

**Definition 2.4 (Separating hyperplane)** A separating hyperplane of a set of points \( \mathcal{A} \) and a point \( p \notin \text{conv}(\mathcal{A}) \) is defined by \((z, z_0) = \text{sep}(\mathcal{A}, p)\), \( z \in \mathbb{R}^d \), \( z_0 \in \mathbb{R} \) such that:

\[
\langle a, z \rangle + z_0 \geq 0 \quad \text{for all} \quad a \in \mathcal{A} \\
\langle p, z \rangle + z_0 < 0
\]

Figure 2.1 shows an example of a 2-dimensional separating hyperplane which separates the encircled point from the others.

The following lemma shows that the infeasibility certificate of some LP(\( \mathcal{A}, p \)) is basically a separating hyperplane of \( \mathcal{A} \) and \( p \).

**Lemma 2.5 (Separating hyperplane from an infeasibility certificate)** The infeasibility certificate \( y = (y_1, y_2, \ldots, y_{d+1}) \) of an infeasible linear program LP(\( \mathcal{A}, p \)) defines a separating hyperplane \((z, z_0) = \text{sep}(\mathcal{A}, p)\) by \( z = (y_1, y_2, \ldots, y_d) \) and \( z_0 = y_{d+1} \).

**Proof** Let \( \mathcal{A} = \{a_1, a_2, \ldots, a_n\} \). We can write the equations from (2.2) for LP(\( \mathcal{A}, p \)) in matrix form like this:

\[
Ax = \begin{pmatrix}
    a_{11} & \cdots & a_{1n} \\
    \vdots & \ddots & \vdots \\
    a_{1d} & \cdots & a_{nd} \\
    1 & \cdots & 1
\end{pmatrix}
\begin{pmatrix}
    x_1 \\
    \vdots \\
    x_d \\
    1
\end{pmatrix}
= \begin{pmatrix}
    p_1 \\
    \vdots \\
    p_d \\
    1
\end{pmatrix} = b
\]
From Definition 2.2 we have:

\[ A^T y = \begin{pmatrix} a_{11} & \cdots & a_{1d} & 1 \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nd} & 1 \end{pmatrix} y \geq 0 \] (2.4)

Note that we have \( d + 1 \) inequalities here. If we split this up into the single inequalities we get the following constraint for every point \( a_i \in \mathcal{A} \):

\[ ( a_{i1} \cdots a_{id} 1 ) \begin{pmatrix} y_1 \\ \vdots \\ y_d \\ y_{d+1} \end{pmatrix} \geq 0 \] (2.5)

This meets exactly our demand on the separating hyperplane \((z, z_0)\) for all \( a_i \in \mathcal{A} \) as we see in the following equation:

\[ \langle a_i, z \rangle + z_0 = \sum_{k=1}^{d} a_{ik} z_k + z_0 = \sum_{k=1}^{d} a_{ik} y_k + y_{d+1} = ( a_{i1} \cdots a_{id} 1 ) \begin{pmatrix} y_1 \\ \vdots \\ y_d \\ y_{d+1} \end{pmatrix} \geq 0 \]

Similarly the inequality \( b^T y < 0 \) of Definition 2.2 yields:

\[ b^T y = ( p_1 \cdots p_d 1 ) \begin{pmatrix} y_1 \\ \vdots \\ y_d \\ y_{d+1} \end{pmatrix} < 0 \] (2.6)

Hence also the second demand from Definition 2.4 on the separating hyperplane \( z, z_0 \) is met:

\[ \langle p, z \rangle + z_0 = \sum_{k=1}^{d} p_k z_k + z_0 = \sum_{k=1}^{d} p_k y_k + y_{d+1} = ( p_1 \cdots p_d 1 ) \begin{pmatrix} y_1 \\ \vdots \\ y_d \\ y_{d+1} \end{pmatrix} < 0 \]

And therefore \((z, z_0)\) with \( z = (y_1, y_2, \ldots, y_d) \) and \( z_0 = y_{d+1} \) defines a separating hyperplane of \( \mathcal{A} \) and \( p \). \( \square \)
2. Preliminaries

2.4 Homogeneous Coordinates

Homogenization is a widely used tool in computational geometry. It is a system of coordinates which has one coordinate more than the Cartesian coordinate system.

Definition 2.6 (Homogeneous coordinates) A point \( p \) with Cartesian coordinates \( (p_1, p_2, \ldots, p_d) \) can be represented in the homogeneous system by the coordinates \( (p_1 h, p_2 h, \ldots, p_d h, h) \) for any positive homogenizing coordinate \( h \).

A point in the Cartesian system maps to a whole ray in the homogeneous system. The transformation from homogeneous space into Cartesian space is the intersection with the \( d \)-dimensional hyperplane where the homogenizing coordinate \( h \) is 1.

Homogenization has some nice properties in computational geometry (much like complex numbers have in electrical engineering). For example, the homogenizing coordinate can be seen as a common denominator. This implicitly treats all the coordinates as fractions and helps to get rid of unnecessary divisions. Many formulas become simpler by using homogeneous coordinates and it is even possible to represent so called points at infinity.

In the implementation we work directly on the homogeneous coordinates as this helps us avoid implicit divisions and is actually more natural. Let us analyze the impact on our linear programs. Remember that by

\[
\text{LP}(\{a_1, a_2, \ldots, a_n\}, p)
\]

we meant the following linear program up to this point:

\[
\begin{align*}
\sum_{i=1}^{n} a_{ik} x_i &= p_k & \text{for all } 1 \leq k \leq d \\
\sum_{i=1}^{n} x_i &= 1 \\
x &\geq 0
\end{align*}
\]

(2.7)

Using Definition 2.6 the point \( a_i \) with Cartesian coordinates \( (a_{i1}, \ldots, a_{id}) \) has the homogeneous coordinates \( (a_{i1} h_i, \ldots, a_{id} h_i, h_i) \) for some positive homogenizing coordinate \( h_i \). Similarly \( p \) is be represented by the homogeneous coordinates \( (p_1 h_p, \ldots, p_n h_p, h_p) \) for some positive \( h_p \).
Consider the following linear program constructed with homogeneous coordinates.

\[
\sum_{i=1}^{n} a_{ik} h_i x_i = p_k h_p
\]

for all \(1 \leq k \leq d\)

\[
\sum_{i=1}^{n} h_i x_i = h_p
\]  \hspace{1cm} (2.8)

\(x \geq 0\)

**Lemma 2.7 (Equivalence of the homogeneous LP under feasibility)** The linear programs 2.7 and 2.8 are equivalent in the sense that 2.7 is feasible if and only if 2.8 is feasible.

**Proof** A solution \(x^* = (x_1^*, ..., x_n^*)\) of 2.7 can be converted into a solution \(y^*\) of 2.8 like this:

\[
y^* = \left( \frac{h_p}{h_1} x_1^*, ..., \frac{h_p}{h_n} x_n^* \right)
\]

As all \(x_k^*\) and all the homogenizing coordinates are positive, the \(y_k^*\) are positive too. So we have \(y^* \geq 0\).

The two remaining constraints from 2.8 follow by substitution and the equations from 2.7:

\[
\sum_{i=1}^{n} a_{ik} h_i y_i^* = \sum_{i=1}^{n} a_{ik} h_p x_i^* = h_p \sum_{i=1}^{n} a_{ik} x_i^* = h_p p_k = p_k h_p
\]

\[
\sum_{i=1}^{n} h_i y_i^* = \sum_{i=1}^{n} h_p x_i^* = h_p \sum_{i=1}^{n} x_i^* = h_p
\]

In the other direction, a solution \(y^* = (y_1^*, ..., y_n^*)\) of 2.8 can be converted into a solution \(x^*\) of 2.7 in a similar way:

\[
x^* = \left( \frac{h_1}{h_p} y_1^*, ..., \frac{h_n}{h_p} y_n^* \right)
\]

As before, the nonnegativity constraints \(x^* \geq 0\) hold (positive \(y_k^*\) and homogenizing coordinates) and for the two remaining constraints of 2.7 we use substitution and the equations from 2.8:

\[
\sum_{i=1}^{n} a_{ik} x_i^* = \sum_{i=1}^{n} a_{ik} \frac{h_i}{h_p} y_i^* = \frac{1}{h_p} \sum_{i=1}^{n} a_{ik} h_i y_i^* = \frac{p_k h_p}{h_p} = p_k
\]
Therefore 2.7 is feasible if and only if 2.8 is feasible.

For the Simple Algorithm this is already enough as all we need from LP(A, p)
is the feasibility check. But in the Dulá-Helgason Algorithm we also need
a separating hyperplane in case p ̸∈ conv(A). The following lemma states
that this still works in exactly the same way.

**Lemma 2.8 (Separating hyperplane from an infeasibility certificate)** The in-
feasibility certificate \( y = (y_1, y_2, ..., y_d, 1) \) of an infeasible linear program LP(A, p)
in form (2.8) defines a separating hyperplane \( (z, z_0) = \text{sep}(A, p) \) by \( z = (y_1, y_2, ..., y_d) \)
and \( z_0 = y_d + 1 \).

**Proof** We adapt the proof of Lemma 2.5. By changing the linear program
equation (2.3) becomes:

\[
Ax = \begin{pmatrix}
  h_1 a_{11} & ... & h_n a_{1n} \\
  : & : & : \\
  h_1 a_{d1} & ... & h_n a_{dn}
\end{pmatrix} x = \begin{pmatrix}
  h_p p_1 \\
  : \\
  h_p p_d \\
  h_p
\end{pmatrix} = b
\]

Instead of equation (2.4) we have:

\[
A^T y = \begin{pmatrix}
  h_1 a_{11} & ... & h_1 a_{1d} & h_1 \\
  : & : & : & : \\
  h_n a_{n1} & ... & h_n a_{nd} & h_n
\end{pmatrix} y \geq 0
\]

Which turns equation (2.5) for a point \( a_i \in A \) into:

\[
\begin{pmatrix}
  h_i a_{i1} & ... & h_i a_{id} & h_i
\end{pmatrix} \begin{pmatrix}
  y_1 \\
  : \\
  y_d \\
  y_{d+1}
\end{pmatrix} = h_i( a_{i1} \cdots a_{id} \ 1 ) \begin{pmatrix}
  y_1 \\
  : \\
  y_d \\
  y_{d+1}
\end{pmatrix} \geq 0
\]

As \( h_i \) is the homogenizing coordinate of \( a_i \) it must be positive and we can
eliminate it from the inequality. By that we get the original inequality (2.5)
back. The rest of the proof for the position of the hyperplane relative to the
points \( a_i \in A \) stays the same.

In the second part of the proof equality (2.6) becomes:

\[
b^T y = \begin{pmatrix}
  h_p p_1 & ... & h_p p_d & h_p
\end{pmatrix} \begin{pmatrix}
  y_1 \\
  : \\
  y_d \\
  y_{d+1}
\end{pmatrix} = h_p( p_1 \cdots p_d \ 1 ) \begin{pmatrix}
  y_1 \\
  : \\
  y_d \\
  y_{d+1}
\end{pmatrix} < 0
\]
Again, we can eliminate the positive $h_p$ from the inequality and get the original equation (2.6) back. The rest of the proof is the same.

2.5 Lexicographical Ordering

A natural order on $d$-dimensional points is the lexicographical order:

**Definition 2.9 (Lexicographical order)** A $d$-dimensional point $a$ is lexicographically smaller than another $d$-dimensional point $b$ (denoted: $a \prec_{\text{lex}} b$) if there exists some $k$ satisfying:

$$
1 \leq k \leq d \\
a_i = b_i \\
\text{for all } 1 \leq i < k \\
a_k < b_k
$$

The lexicographically smallest point is defined like this:

**Definition 2.10 (Lexicographically smallest point)** The lexicographically smallest point of a point set $A$ is defined through:

$$
\min_{\text{lex}} p \in A \text{ such that } p \prec_{\text{lex}} p' \text{ for all } p' \in A \setminus \{p\}
$$

On a point set where all points differ, the lexicographical order is a strict total order without equalities and therefore a lexicographically smallest point exists and is unique.

The following lemma proves that the lexicographically smallest point is an extreme point. We will use this in the initialization of the Dulá-Helgason Algorithm for finding the first extreme point.

**Lemma 2.11 (Lexicographically smallest point is an extreme point)** The lexicographically smallest point $\min_{\text{lex}}(A)$ of a set of points $A$ is an extreme point.

**Proof** Let $m := \min_{\text{lex}}(A)$ be the lexicographically smallest point of $A$. If $A = \{m\}$ the lemma holds trivially. Otherwise let $m' := \min_{\text{lex}}(A \setminus \{m\})$.

As $m \prec_{\text{lex}} m'$ there exists some $k$ for which $m_k < m'_k$ holds and $m_i = m'_i$ for all $1 \leq i < k$.

Up to (and including) the $k$th dimension $m$ must have the smallest coordinate among all points from $A$ (or else another point would be lexicographically smaller which contradicts our choice of $m$ as the lexicographically smallest point).

Assume for the sake of contradiction that $m$ is non-extreme, i.e. $m \in \text{conv}(A \setminus \{m\})$. The point $m$ has to be some convex combination of the other points (see Definition 1.1). Consider the points with positive coefficient $\lambda_p$. In the first $k$
dimensions these points must all have the same coordinates as $m$. Otherwise $m$ cannot have the smallest coordinate in one of these dimensions and thus violating the fact from the previous paragraph. All the points having the same coordinates in dimension up to (and including) $k$ are lexicographically smaller than $m'$. But only $m$ is lexicographically smaller than $m'$ and $m \notin A\{m\}$. So there is no point with positive coefficient $\lambda_p$ which is a contradiction as the coefficients $\lambda_p$ of the convex combination have to sum up to 1.

Therefore no convex combination of the other points exists and $m$ has to be an extreme point. \hfill \Box
In this chapter we discuss two existing frame algorithms: a straightforward one which we will call the *Simple Algorithm* and an output-sensitive one from Dulá and Helgason [10]. Both of them were used in the implementation of the CGAL package (see Chapter 6). The pseudo code presented here follows mostly the paper from Dulá and López [9].

For now we will make two assumptions about our input points which will facilitate the discussion about the algorithms:

*No duplication:* No point occurs twice in the input.

*General position:* Any point \( p \) from a set of points \( \mathcal{P} \) on the boundary of its convex hull \( \text{conv}(\mathcal{P}) \) is an extreme point. Note that usually a stronger condition is meant by general position but this one is sufficient for our needs.

We will see in Chapter 6 how to get around these assumptions for the two implemented algorithms as we will not put such restrictions on our input.

### 3.1 Simple Algorithm

We start with a well-known frame-algorithm based on linear programming which we will refer to as the *Simple Algorithm*.

Let \( \mathcal{P} = \{p_1, ..., p_n\} \) be the set of input points. A point \( p_i \in \mathcal{P} \) is extreme exactly if it cannot be expressed as a convex combination of the other points. This can be decided easily by the linear program \( \text{LP}(\mathcal{P}\setminus\{p_i\}, p) \) introduced in Section 2.2.

The Simple Algorithm classifies all points using this method. This involves solving \( n \) linear programs with \( n - 1 \) nonnegative variables and \( d + 1 \) equality constraints each.
Algorithm 1 presents the pseudo code of the Simple Algorithm. Note that the condition in line 4 is decided by the feasibility of the linear program in line 3.

**Algorithm 1 Simple Algorithm**

**Input:** \( P = \{p_1, ..., p_n\} \)
1. \( F \leftarrow \emptyset \)
2. for \( i = 1 \) to \( n \) do
3. Solve LP(\( P \setminus \{p_i\} \), \( p_i \)).
4. if \( p_i \notin \text{conv}(P \setminus \{p_i\}) \) then
5. \( F \leftarrow F \cup \{p_i\} \)
6. end if
7. end for
**Output:** \( F = \text{frame}(P) \)

**Theorem 3.1 (Correctness of Algorithm 1)** Algorithm 1 is correct\(^1\), given the used linear programming solver is correct too.

**Proof** It is easy to see that the resulting set \( F \) is indeed the frame of \( P \) as every point is classified explicitly (and correctly) as extreme or non-extreme. Termination is also straightforward to see. \( \square \)

**Theorem 3.2 (Runtime of Algorithm 1)** Using Definition 2.1 for LP\(_{k,n}\), the runtime of Algorithm 1 is

\[ O(n \text{ LP}_{d+1,n-1}) \]

**Proof** For all \( n \) points the algorithm solves a linear program with \( n - 1 \) non-negative variables and \( d + 1 \) equality constraints. Everything else is \( O(n) \). \( \square \)

### 3.2 Dulá-Helgason Algorithm

In 1996 J.H. Dulá and R.V. Helgason designed an output-sensitive frame algorithm [10] in which the linear programs have at most \( m \) variables where \( m \) is the number of extreme points. The relative number of extreme points is called extreme point density.

**Definition 3.3 (Extreme point density)**

\[
\text{density}(\mathcal{A}) := \frac{|\text{frame}(\mathcal{A})|}{|\mathcal{A}|}
\]

\(^1\)By correctness we mean that an algorithm terminates and its output meets the specification.
The algorithm is similar to our Simple Algorithm in that it uses the same linear programs for checking convex combinations. However two additional observations are made. The first one is that if we have some point $p$ outside the convex hull of a point set $\mathcal{F}$ then a hyperplane separating $p$ from the $\mathcal{F}$ can be found as the infeasibility certificate of LP($\mathcal{F}, p$) (see Lemma 2.5). The second observation is that the point which is minimal in some arbitrary direction (i.e. the point with minimal scalar product to some $z \in \mathbb{R}^d$) is an extreme point. The reason for the latter is that all points which are minimal in a fixed direction lie on a face of the convex hull of the input points and thus on its boundary. Using our general position assumption, every such point is extreme.

Algorithm 2 Dulá-Helgason Algorithm

**Input:** $\mathcal{P} = \{p_1, ..., p_n\}$

1. $\mathcal{F} \leftarrow \{\text{min}_{\text{lex}}(\mathcal{P})\}$
2. $\mathcal{N} \leftarrow \emptyset$
3. for $i = 1$ to $n$ do
4.   if $p_i \notin \mathcal{F}$ then
5.     repeat
6.       Solve LP($\mathcal{F}, p_i$).
7.       if $p_i \in \text{conv}(\mathcal{F})$ then
8.         $\mathcal{N} \leftarrow \mathcal{N} \cup \{p_i\}$
9.       else
10.          $(z, z_0) = \text{sep}(\mathcal{F}, p_i)$
11.          $p^* \leftarrow \underset{p \in \mathcal{P}, f | > i}{\text{argmin}} \langle z, p \rangle$
12.         $\mathcal{F} \leftarrow \mathcal{F} \cup \{p^*\}$
13.       end if
14.   until $p_i \in \mathcal{F}$ or $p_i \in \mathcal{N}$
15. end if
16. end for

**Output:** $\mathcal{F} = \text{frame}(\mathcal{P})$

Algorithm 2 presents the pseudo code of the Dulá-Helgason Algorithm. At any point in time during execution of the algorithm, the set $\mathcal{F}$ contains the extreme points found so far and similarly the set $\mathcal{N}$ contains the points found to be non-extreme so far.

We start by initializing $\mathcal{F}$ with the lexicographically smallest point which is an extreme point (see Lemma 2.11). For all points $p_i \in \mathcal{P}$ we do the following:

If $p_i$ is not yet found to be extreme we solve LP($\mathcal{F}, p_i$). Based on the feasibility of this linear program the algorithm can decide the condition of line 7...
whether \( p_i \in \text{conv}(\mathcal{F}) \). If \( p_i \) is a convex combination of \( \mathcal{F} \), we classify it as non-extreme and immediately take the next point \( p_i \) (the repeat-until loop terminates as \( p_i \in \mathcal{N} \) which makes the condition in line 14 true). Otherwise we use the infeasibility certificate of LP(\( \mathcal{F}, p_i \)) as a hyperplane \((z,z_0)\) which separates all points \( \mathcal{F} \) from \( p_i \) (line 10). Now we look for some point \( p^* \in \mathcal{P} \) with minimal scalar product with \( z \). We will see in the following paragraph on correctness that this point is extreme and must be some point \( p_j \in \mathcal{P} \setminus \mathcal{F} \) where \( j \geq i \). As an extreme point we add \( p^* \) to \( \mathcal{F} \). If \( p^* = p_i \) the condition in line 14 becomes true (as \( p_i \in \mathcal{F} \)) and we take the next \( p_i \) in the outermost loop. Otherwise we repeat this process with the updated \( \mathcal{F} \).

**Theorem 3.4 (Correctness of Algorithm 2)** Algorithm 2 is correct, given the used linear programming solver is correct too.

**Proof** The following invariants hold:

1. All points of \( \mathcal{N} \) are non-extreme.
2. Inside the outermost loop, all points \( p_j \) with \( j < i \) are already classified, i.e. \( p_i \in \mathcal{F} \) or \( p_j \in \mathcal{N} \).
3. All points of \( \mathcal{F} \) are extreme.

Invariant 1 holds as the only time \( \mathcal{N} \) is changed is line 8 where it is obvious that \( p_i \) is non-extreme.

Invariant 2 holds also trivially as line 14 ensures that at the end of every iteration of the outermost loop the point \( p_i \) is classified.

Invariant 3 holds as the only time \( \mathcal{F} \) is changed is line 12 and the point \( p^* \) added there is an extreme point for the following reason: As stated earlier, any point with minimal scalar product to some fixed vector \( z \in \mathbb{R}^d \) is an extreme point (note that we are still using our general position assumption). In line 11 we choose the point \( p_j \) among all points \( \mathcal{P} \) which has the smallest scalar product with the vector \( z \) given from the separating hyperplane of \( \mathcal{F} \) and \( p_i \). We do not have to test the points from \( \mathcal{F} \) as any of them has a bigger scalar product with \( z \) than \( p_i \) has (this comes from Definition 2.4 of the separating hyperplane). We also do not have to test any point \( p_j \) with \( j < i \) as by Invariant 2 such a \( p_j \) is either in \( \mathcal{F} \), in which case the previous reasoning applies, or else \( p_j \) is in \( \mathcal{N} \) which means it is non-extreme and therefore cannot have the minimal scalar product with \( z \).

A corollary of Invariant 2 is that in the end all points of \( \mathcal{P} \) are classified. Together with Invariants 1 and 3 we can conclude that the returned \( \mathcal{F} \) is indeed the frame of \( \mathcal{P} \).

What remains to show is that the algorithm terminates. This is not difficult to see as in every iteration of the repeat-until loop, a new point gets classified. Note that in line 11 no point from \( \mathcal{F} \) or \( \mathcal{N} \) can be chosen for \( p^* \).
3.2. Dulá-Helgason Algorithm

Points from $\mathcal{F}$ are excluded explicitly and points from $\mathcal{N}$ implicitly by the condition $j \geq i$ and Invariant 2. As we only have a finite amount of points, the repeat-until loop must at some point classify $p_i$, and the outermost loop terminates trivially.

\textbf{Theorem 3.5 (Runtime of Algorithm 2)} Using Definition 2.1 for $LP_{k,n}$, the runtime of Algorithm 2 is

$$O(n \ LP_{d+1,m} + dnm)$$

where $m$ is the number of extreme points.

\textbf{Proof} The first part comes from the linear programs we have to solve. Every time we solve a linear program, some point gets classified. As we classify all points and the first point is classified without linear programming in the initialization, we solve $n - 1 \in O(n)$ linear programs. Their number of variables is bounded by the number of extreme points as $\mathcal{F} \subseteq \text{frame}(\mathcal{P})$ at any point in time.

The second part comes from finding the minimal scalar product in line 11. We execute this line at most $m$ times as every execution yields a new extreme point. One execution costs $O(dn)$ time as we have to calculate up to $n - 1$ scalar products, each of which need $O(d)$ multiplications and additions.

The computation of $\min_{i \in \mathcal{P}}(\mathcal{P})$ can be done in $O(dn)$ which is already dominated by the $O(dnm)$ of the second part. And as we classify one point in every iteration of the repeat-until loop, all the rest is done in $O(n)$ time. □

As the runtime heavily depends on the number of output points, the algorithm is indeed output-sensitive.
In this chapter we investigate the situation where the dimension $d$ is high but all our points are sparse, i.e. most of their coordinates are zero. We use the following definition for a general level of sparsity.

**Definition 4.1 (k-sparsity)** For a natural number $k$, a point is called *k*-sparse if it has no more than $k$ nonzero coordinates.

Similarly a set of points $P$ is called *k*-sparse if all of its points $p \in P$ are k-sparse.

The fundamental question of this chapter is whether there exists a frame algorithm for a $k$-sparse, $d$-dimensional point set with $n$ points, which is substantially better than any of the known algorithms which are not aware of sparsity.

For this and also for the next chapter we assume that the input contains no duplicate points. In fact this can be achieved quite easily by sorting the points lexicographically and then removing the duplicates.

### 4.1 Representation of Sparse Points

Sparse points are usually represented by a sorted list of pairs where every pair represents a nonzero coordinate by specifying its dimension and value. This list is sorted by dimension. We often need to know in what dimensions a point has nonzero coordinates. This is called the *support*.

**Definition 4.2 (Support)** The set of dimensions where a $d$-dimensional sparse point $p$ has nonzero coordinates is called the *support* of $p$ and denoted by

$$D(p) \subseteq \{1, 2, ..., d\}$$
Furthermore, we often need to select the points with the same support from some point set. Let us define a notation for that too.

**Definition 4.3 (D-support-fragment)** Given a set of points $\mathcal{P}$ and a support $D \subseteq \{1,2,...,d\}$, the set of all points $p \in \mathcal{P}$ which have support $D$ is called the D-support-fragment of $\mathcal{P}$ and written:

$$\mathcal{P}_D := \{ p \in \mathcal{P} | D(p) = D \}$$

For the value of the Cartesian coordinate in dimension $i$ of a point $p$ we continue using the notation $p_i$. However, we have to keep in mind that coordinate access is not anymore strictly constant and depends on the data structure used for storing sparse points. For example if we have a dimension/value list as described before, coordinate access can be done in $O(\log(|D(p)|))$ time using binary search. If a hash table with good parameters is used, almost constant coordinate access times are possible.

### 4.2 One-Sparse Points

The case $k = 1$ is easy. As every point has at most one non-zero coordinate we can have at most two extreme points in every dimension – the point with the maximal and the point with the minimal coordinate. All points in-between are clearly non-extreme. The total number of extreme points is therefore bounded by $2d$.

Figure 4.1 shows an example of a one-sparse point set in two dimensions. The extreme points are encircled.

Let $\mathcal{P}$ be the set of input points and assume the origin $O$ is part of $\mathcal{P}$. Then we can express the frame of $\mathcal{P}$ according to the following theorem.
4.2. One-Sparse Points

Theorem 4.4 (Frame of a one-sparse point set including the origin) In a one-sparse point set \( \mathcal{P} \) which includes the origin \( O \), the frame of \( \mathcal{P} \) is given as follows.

\[
\text{frame}(\mathcal{P}) \setminus \{O\} = \left\{ p \in \mathcal{P} \setminus \{O\} \mid p_k = \max_{q \in \mathcal{P} \setminus \{O\}} q_k \text{ or } p_k = \min_{q \in \mathcal{P} \setminus \{O\}} q_k \right\}
\]

\( O \in \text{frame}(\mathcal{P}) \iff \text{for all } 1 \leq k \leq d : \max_{p \in \mathcal{P}} p_k = 0 \text{ or } \min_{p \in \mathcal{P}} p_k = 0 \)

**Proof** A point \( p \neq O \) with the maximal (or minimal) coordinate in its nonzero dimension \( k \) is clearly extreme as no convex combination of the other points can get this high (or low) in dimension \( k \).

A point \( p \neq O \), which does not have the maximal or minimal coordinate in its nonzero dimension \( k \), is a convex combination of the origin and the point which has the maximal (if \( p_k > 0 \)) or the minimal (if \( p_k < 0 \)) coordinate in dimension \( k \).

For the origin, if there is any dimension \( k \) where the maximum coordinate is greater than 0 and the minimum coordinate is smaller than 0, then clearly the origin is a convex combination of the two points which have the maximal and minimal coordinates in dimension \( k \). Otherwise no convex combination of the other points exist that yield the origin, as in this case any convex combination of \( \mathcal{P} \setminus \{O\} \) has a nonzero coordinate in at least one dimension.

The resulting algorithm from this theorem runs in \( O(n + d) \) by just finding out the minimum and maximum coordinate in every dimension, and then going through all points to find out which ones are extreme (in constant time per point).

If the origin \( O \) is not part of the input, we have to distinguish between two cases: whether \( O \) is in the convex hull of \( \mathcal{P} \) or not. The origin is part of \( \text{conv}(\mathcal{P}) \) exactly if there exists a dimension \( i \) whose maximal coordinate is greater than zero and whose minimal coordinate is less than zero. In this case, let these two points with maximal and minimal coordinates in dimension \( i \) be called \( a \) and \( b \). Clearly, \( a \) and \( b \) are extreme points, and all other points with support \( \{i\} \) are non-extreme. The origin is a convex combination of \( a \) and \( b \). Hence, for all the points with another support, the situation is exactly the same as in Theorem 4.4. Therefore, we have the following formula for the case where the origin is not part of \( \mathcal{P} \) but part of \( \text{conv}(\mathcal{P}) \):

\[
\text{frame}(\mathcal{P}) = \left\{ p \in \mathcal{P} \mid p_k = \max_{q \in \mathcal{P} \setminus \{O\}} q_k \text{ or } p_k = \min_{q \in \mathcal{P} \setminus \{O\}} q_k \right\}
\]

Note, that this formula also works for all the points with support \( \{i\} \).
In our last case, the origin is not part of \( \text{conv}(P) \). This means that no convex combination can yield the origin. If we want to express a point \( p \) with support \{i\} as the convex combination of some other points of \( P \), we cannot use any point with another support \{j\}. Otherwise we wouldn’t be able to get the coordinate of dimension \( j \) back to zero. Therefore, only the points from the \{i\}-support-fragment matter. Hence, the maximal and minimal point of every dimension is extreme.

So we do not need anything more than the minima and maxima in the case \( O \notin P \) either and we see that the frame of a one-sparse point set can always be found in \( O(n + d) \).

### 4.3 Preprocessing

For \( k = 2 \) the situation is already nontrivial, as it covers the general (non-sparse) case in \( \mathbb{R}^2 \).

For simplicity we will not go beyond the two-sparse case in the rest of this chapter. However, most of the developed theory will also expand to the cases with bigger \( k \).

For now we also assume that all of our points have a support of exactly two dimensions. This makes things a bit simpler. Furthermore, we assume that for every dimension at least one point with nonzero coordinate exists.

Let \( P \) be the set of input points where all the supports have cardinality two. As a first observation, a point \( p \in P \) which is inside the (two-dimensional) convex hull of \( P \setminus \{p\} \) is clearly non-extreme. We call such a point trivially non-extreme.

#### Algorithm 3 Preprocessing

**Input:** \( P \) with \( |D(p)| = 2 \) for all \( p \in P \)

1. \( P' \leftarrow P \)
2. for all \( D = \{i, j\} \) with \( P_D \neq \emptyset \) do
3. Compute the two-dimensional convex hull \( \text{conv}(P_D) \) of \( P_D \)
4. for all \( p \in P_D \) do
5. if \( p \in \text{conv}(P_D \setminus \{p\}) \) then
6. \( P' \leftarrow P' \setminus \{p\} \)
7. end if
8. end for
9. end for

**Output:** \( P' \) with frame(\( P \)) \( \subseteq P' \subseteq P \) and \( p \notin P' \) for all trivially non-extreme points \( p \in P \)
Algorithm 3 filters such points out efficiently. Its correctness is obvious as all filtered out points are trivially non-extreme. And all trivially non-extreme points are found.

**Theorem 4.5 (Runtime of Algorithm 3)** For a set of n points, Algorithm 3 runs in time $O(n \log(n))$.

**Proof** The two dimensional convex hull of $m$ points can be computed in $O(m \log(m))$ using any of the standard two-dimensional convex hull algorithms, for example Graham scan. The queries in line 5 can also be done in $O(m \log(m))$ by a binary search on the lower and upper envelope of $\text{conv}(P_D)$. Note that a point $p$ is contained in $\text{conv}(P_D \setminus \{p\})$ exactly if it is not a vertex of $\text{conv}(P_D)$.

To find the support-fragments of $P$, we sort the points by support. This needs $O(kn \log(n))$ time, which is $O(n \log(n))$ for a constant $k$.

For every support-fragment with $m$ points we need $O(m \log(m))$ time as both the construction of the convex hull and the queries in line 5 can be done in $O(m \log(m))$ and the rest is in $O(m)$.

If there are $j$ support-fragments and their respective sizes are $m_1, m_2, ..., m_j$, we have:

$$\sum_{i=1}^{j} m_i = n$$

and our runtime for handling all support-fragments is in order of

$$\sum_{i=1}^{j} (m_i \log(m_i)) \leq \sum_{i=1}^{j} (m_i \log(n)) = \log(n) \sum_{i=1}^{j} m_i = n \log(n)$$

So our total runtime is $O(n \log(n))$. \hfill $\square$

Unfortunately, this preprocessing does not detect all non-extreme points as the following family of examples for an odd number of dimensions demonstrates:

$$P = (p_1 \ p_2 \ ... \ p_d) = \begin{pmatrix} 1 & d - 1 & 1 - d & \ldots & 1 - d & \ldots & 1 - d \\ d - 1 & 1 - d & 1 - d & \ldots & 1 - d & \ldots & 1 - d \\ 1 - d & 1 - d & 1 - d & \ldots & 1 - d & \ldots & 1 - d \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 1 - d & 1 - d & 1 - d & \ldots & 1 - d & \ldots & 1 - d \\ 1 & \ldots & \ldots & \ldots & 1 - d & \ldots & 1 - d \\ 1 & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \end{pmatrix}$$

Here the point $p_1$ is clearly a convex combination of the other points as
4. Sparse Input

\[ p_1 = \sum_{i=2}^{d} \frac{1}{d-1} p_i \]

But as \( p_1 \) is the only point in its supporting fragment, it is not filtered out by Algorithm 3. Similar examples can be constructed for even dimensions.

4.4 The Dimension Graph

The example in the previous section tells us that the connection of the dimensions is somehow important. Let us formalize this by defining a graph on the dimensions:

**Definition 4.6 (Dimension Graph)** The Dimension Graph of a \( k \)-sparse, \( d \)-dimensional point set \( \mathcal{P} \) is the undirected graph \( G = (V, E) \) where

\[ V := \{1, 2, \ldots, d\} \]

and

\[ E := \{\{i, j\} \mid \exists p \in \mathcal{P} \text{ s.t. } \{i, j\} \subseteq D(p)\} \]

In other words, the Dimension Graph of \( \mathcal{P} \) is the graph on all dimensions, where two dimensions are connected if there exists some point \( p \in \mathcal{P} \) having nonzero coordinates in both of them.

**Lemma 4.7 (Disconnected Dimension Graph)** If the Dimension Graph is not connected we can split the problem into smaller sub-problems in such a way, that the result is just the union of the result of the sub-problems. More precisely if we can partition the dimensions \( \{1, 2, \ldots, d\} \) into the sets \( D_1 \) and \( D_2 \) in such a way that for every point \( p \in \mathcal{P} \) the support \( D(p) \) lies either completely in \( D_1 \) or in \( D_2 \), the frame of \( \mathcal{P} \) is just the union of the frames of two sub-problems:

\[ \text{frame} (\mathcal{P}) = \text{frame} (\{ p \in \mathcal{P} | D(p) \subseteq D_1 \}) \cup \text{frame} (\{ p \in \mathcal{P} | D(p) \subseteq D_2 \}) \]

**Proof** Let \( \mathcal{P}_1 := \{ p \in \mathcal{P} | D(p) \subseteq D_1 \} \) and \( \mathcal{P}_2 := \{ p \in \mathcal{P} | D(p) \subseteq D_2 \} \) be the partition of \( \mathcal{P} \). It is easy to see that

\[ \text{frame} (\mathcal{P}) \subseteq \text{frame} (\mathcal{P}_1) \cup \text{frame} (\mathcal{P}_2) \]

as every extreme point \( p \) of \( \mathcal{P} \) is also an extreme point of one of \( \mathcal{P}_1 \) or \( \mathcal{P}_2 \).

For the other direction, note that all convex combinations of the points \( p \in \mathcal{P}_1 \) need to have its support in \( D_1 \) as those are the only dimensions where points from \( \mathcal{P}_1 \) have nonzero coordinates.
4.5 Sparse LP-Solvers

If we take an extreme point $p$ of $\mathcal{P}_1$, then $p$ cannot be expressed as convex combination of $\mathcal{P}_1 \setminus \{p\}$. As all convex combinations from $\mathcal{P}_2$ have support in $D_2$ and $D_1 \cap D_2 = \emptyset$, the point $p$ cannot be expressed as a convex combination of $\mathcal{P}_2 \cup \mathcal{P}_1 \setminus \{p\}$ either. Hence $p$ is also an extreme point of $\mathcal{P}$.

The same argument works for the extreme points of $\mathcal{P}_2$, and therefore we have:

$$\text{frame}(\mathcal{P}) \supseteq \text{frame}(\mathcal{P}_1) \cup \text{frame}(\mathcal{P}_2)$$

which concludes our proof.

4.5 Sparse LP-Solvers

A simple method of dealing with sparse input is to just take any algorithm for the dense problem, for example the Dulá-Helgason Algorithm, and use a sparse LP-solver for solving the linear programs.

Remember that the key part of the dense frame algorithms is the linear program $\text{LP}(\mathcal{A}, p)$. If we write this linear program in matrix notation like in (2.1), the matrix $\mathcal{A}$ has at most $k + 1$ nonzero entries per column. The vector $b$ has also at most $k + 1$ nonzero entries and the vector $c$ has all entries set to zero anyway.

A sparse LP-solver can take as input our sparse matrix $\mathcal{A}$ and sparse vectors $b$ and $c$ in a compact representation and make use of their sparsity. Taking a sparse LP-solver is certainly better than taking an LP-solver which is not aware of sparsity. However, sparse LP-solvers are not aware of the structure of the underlying problem (i.e. the $k$-sparse frame problem). Therefore, they cannot take full advantage of it.

4.6 Divide and Conquer

Lemma 4.7 gives us the idea to try a divide and conquer approach, as the conquer step is trivial in the case of a disconnected Dimension Graph.

Algorithm 4 shows the outline of the divide and conquer approach. Line 1 partitions the set $\mathcal{P}$ into disjoint subsets. Line 3 solves a sub-problem, which is to compute the frame of a subset. This is done by recursion until the result is trivial. Line 5 is called the conquer step, where the results of the sub-problems are merged to form the result of the original problem. The procedure MERGE is not specified further but has to be discussed, as it is the most important part of the divide and conquer approach. As already stated before, all MERGE has to do for a dimension-independent partitioning is to concatenate the frames of the subsets. But the general MERGE procedure
4. Sparse Input

**Algorithm 4 Divide and Conquer**

**Input:** $\mathcal{P}$

1. Partition $\mathcal{P}$ into $m \geq 2$ disjoint sets $\mathcal{P}_1 \cup \mathcal{P}_2 \cup \ldots \cup \mathcal{P}_m = \mathcal{P}$

2. for $i = 1$ to $m$ do
   3. $\mathcal{F}_i \leftarrow \text{frame}(\mathcal{P}_i)$ # recursive call

4. end for

5. $\mathcal{F} \leftarrow \text{MERGE}(\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_m)$

**Output:** $\mathcal{F} = \text{frame}(\mathcal{P})$

seems to be quite costly. Let us just try to avoid it, and see how far we can get without.

Let $\mathcal{P}$ be the original input set and look at this divide and conquer approach from bottom-up. We find that Algorithm 3 already computes the frames of all support-fragments of $\mathcal{P}$, i.e. it already solves the very basic sub-problems. After using that, we can merge any of these sub-problems which have independent dimensions.

In order to make the final MERGE task as easy as possible, we try to end up with the minimal number of sub-problems for our original $\mathcal{P}$. This is exactly the edge-coloring problem on the Dimension Graph $G$ of $\mathcal{P}$: Every support-fragment is represented by an edge of $G$ (remember that we still have $|D(p)| = 2$ for all points $p \in \mathcal{P}$) and its frame is already found by Algorithm 3. The minimal edge-coloring is what we are looking for: it corresponds to the minimal number of independent sets of support-fragments.

Vizing’s Theorem [18] states, that the chromatic index (i.e. the minimal number of colors needed for a valid edge-coloring) of a graph is either $\Delta$ or $\Delta + 1$, where $\Delta$ is the maximal vertex degree. To decide which of the two is the chromatic index is in fact NP-complete [13]. But for our case $\Delta + 1$ is sufficient, and such a coloring can be found with a greedy algorithm.

In our Dimension Graph, $\Delta$ is at most $d - 1$ which means we can always find an edge-coloring with $d$ colors. Hence, our bottom-up strategy provides a way to get very efficiently the frame of at most $d$ disjoint subsets of $\mathcal{P}$. Now we have a small number of already solved sub-problems for our main problem – finding the frame of $\mathcal{P}$ – and this is as far as we get with our basic tools.

For the final step we cannot get around using a general MERGE procedure. However, finding such a procedure which is efficient seems to be very tricky. Unfortunately the author did not make any breakthrough in this area.

And so the fundamental question – whether there exists an algorithm solving the sparse problem significantly faster than the dense problem – remains unsolved.
Chapter 5

On-Line Problem

This chapter deals with finding extreme points in an on-line setting. More precisely we have a set of points \( P \) and two types of operations:

- Add one or more points to \( P \)
- Compute the extreme points of \( P \)

The goal of this section is to find and analyze algorithms which make use of extreme point computations of a subset of \( P \).

Let \( \mathcal{P} \) be the current set of points, \( \mathcal{A} \) a subset of \( \mathcal{P} \) for which we already know the extreme points, and \( \mathcal{B} := \mathcal{P} \backslash \mathcal{A} \).

So we have \( \mathcal{P} = \mathcal{A} \cup \mathcal{B} \), and we already know \( \text{frame}(\mathcal{A}) \). Let us use the following variables for the cardinalities of the sets:

\[
\begin{align*}
  n & := |\mathcal{A}| \\
  n' & := |\text{frame}(\mathcal{A})| \\
  m & := |\mathcal{B}| \\
  m' & := |\text{frame}(\mathcal{B})|
\end{align*}
\]

Figure 5.1 shows two sets of two-dimensional points and their convex hulls. The extreme points of the unified set are encircled.

5.1 Ignoring Non-Extreme Points

A simple but in practice very important observation is that we can ignore all the non-extreme points of \( \mathcal{A} \). We will prove this in Theorem 5.2 for which we need the following lemma.
Lemma 5.1 (Convex hull of the frame) For any $d$-dimensional point set $A$ the following holds:

$$\text{conv}(A) = \text{conv}(\text{frame}(A))$$

Proof As $\text{frame}(A) \subseteq A$, all the convex combinations of $\text{frame}(A)$ are also convex combinations of $A$ which shows that $\text{conv}(A) \supseteq \text{conv}(\text{frame}(A))$.

It remains to show the other direction: $\text{conv}(A) \subseteq \text{conv}(\text{frame}(A))$. We do that incrementally:

Take any non-extreme point $p \in A$. If there is no such point, then $A = \text{frame}(A)$ and we are already done. As $p \in \text{conv}(A\setminus\{p\})$ there exists some convex combination

$$p = \sum_{q \in A\setminus\{p\}} \lambda_q q$$

where $\lambda_q \geq 0$ for each $q$ and $\sum_{q \in A\setminus\{p\}} \lambda_q = 1$

expressing $p$ with the points $A\setminus\{p\}$. If we take any point $p' \in \text{conv}(A)$, we can express $p'$ as a convex combination of $A\setminus\{p\}$ as follows: we take some convex combination of $A$ for $p'$ and replace the (possible) occurrence of $p$ with coefficient $\lambda'_p$ by $\lambda'_p \sum_{q \in A\setminus\{p\}} \lambda_q q$.

Now we know that

$$\text{conv}(A) \subseteq \text{conv}(A\setminus\{p\})$$

and we can express all points of $A$ as a convex combination of $A\setminus\{p\}$ (as $p \in \text{conv}(A) \subseteq \text{conv}(A\setminus\{p\})$ for all $p \in A$). We can now take another non-extreme point $q$ and repeat the process. This way all points from $\text{conv}(A)$ can be expressed as convex combinations of $A\setminus\{p, q\}$ which means

$$\text{conv}(A) \subseteq \text{conv}(A\setminus\{p, q\})$$
Repeating this process for all non-extreme points yields
\[ \text{conv}(A) \subseteq \text{conv}(\text{frame}(A)) \]
which is what we need to conclude the proof. \( \square \)

**Theorem 5.2 (Non-extreme points of a subset)** For any \( d \)-dimensional point sets \( A \) and \( B \) the following holds:
\[ \text{frame}(A \cup B) = \text{frame}(\text{frame}(A) \cup B) \]

**Proof** Take any extreme point \( p \) of \( A \cup B \), i.e. \( p \in \text{frame}(A \cup B) \). We have:
\[ p \notin \text{conv}((A \cup B)\setminus \{p\}) \]
As \( (\text{frame}(A) \cup B)\setminus \{p\} \subseteq (A \cup B)\setminus \{p\} \) we also have:
\[ \text{conv}((\text{frame}(A) \cup B)\setminus \{p\}) \subseteq \text{conv}((A \cup B)\setminus \{p\}) \]
and conclude that \( p \notin \text{conv}((\text{frame}(A) \cup B)\setminus \{p\}) \). As \( p \) is an extreme point of \( A \cup B \), it cannot be a convex combination of \( A\setminus \{p\} \) (i.e. \( p \) cannot be a non-extreme point of \( A \)) and therefore it must still be an element of \( \text{frame}(A) \cup B \). So we have:
\[ \text{frame}(A \cup B) \subseteq \text{frame}(\text{frame}(A) \cup B) \]
The other direction – \( \text{frame}(A \cup B) \supseteq \text{frame}(\text{frame}(A) \cup B) \) – means in words that no new extreme points can appear when we remove the non-extreme points from \( A \). By Lemma 5.1 all the non-extreme points from \( A \) can be expressed as convex combination of \( \text{frame}(A) \). This means, that by omitting these points, we still have the same convex hull:
\[ \text{conv}(A \cup B) = \text{conv}(\text{frame}(A) \cup B) \]
which proves that we cannot have any new extreme point as this must have increased the convex hull. So we have
\[ \text{frame}(A \cup B) \supseteq \text{frame}(\text{frame}(A) \cup B) \]
which concludes our proof. \( \square \)

The nice thing about this theorem is that it allows us to keep only the current extreme points after any extreme point computation in our increasing point set model. All currently non-extreme points cannot become extreme anymore and so we can discard them right away.
5.2 One New Point

Let us look at the situation where only one point \( b \) is new, i.e. \( m = |B| = 1 \) and \( B = \{ b \} \). Figure 5.2 shows a two-dimensional example of this situation. The point \( b \) is on the top right. The extreme points of the new set \( A \cup B \) are encircled. For every extreme point \( p \) of \( A \), a straight line is drawn which separates \( p \) from \( A \setminus \{ p \} \).

Algorithm 5 One New Point

**Input:** \( A' = \text{frame}(A), B = \{ b \}, \text{sep}(A' \setminus \{ a \}, a) \) for all \( a \in A' \)

1. Solve LP(\( A', b \)).
2. **if** \( b \in \text{conv}(A') \) **then**
3. \( F \leftarrow A' \)
4. **else**
5. \( F \leftarrow \{ b \} \)
6. **for all** \( a \in A' \) **do**
7. \( (z, z_0) = \text{sep}(A' \setminus \{ a \}, a) \)
8. **if** \( \langle z, b \rangle + z_0 \geq 0 \) **then**
9. \( F \leftarrow F \cup \{ a \} \)
10. **else**
11. Solve LP(\( A' \setminus \{ a \} \cup \{ b \}, a \)).
12. **if** \( p \notin A' \setminus \{ a \} \cup \{ b \} \) **then**
13. \( F \leftarrow F \cup \{ a \} \)
14. **end if**
15. **end if**
16. **end for**
17. **end if**

**Output:** \( F = \text{frame}(A \cup B) \)
Algorithm 5 solves this particular frame problem like this:

We have two possibilities: Either $b \in \text{conv}(A)$ or $b \notin \text{conv}(A)$. From Theorem 5.2 we know that we only have to test whether $b \in \text{conv(frame}(A))$. We can do this by solving $\text{LP(frame}(A), b)$.

If $b \in \text{conv(frame}(A)) = \text{conv}(A)$ we have $\text{frame}(P) = \text{frame}(A)$ and we are done.

Otherwise we know that $b \in \text{frame}(P)$ but some of the points in $\text{frame}(A)$ could lose there status as extreme point.

So in this case we have to check every point $a$ from $\text{frame}(A)$ whether it is still extreme. We can do this by solving $\text{LP(frame}(A) \setminus \{a\} \cup \{b\})$. However, if we maintain for every extreme point $a$ a hyperplane separating $a$ from $\mathcal{A} \setminus \{a\}$, a quick $O(d)$ test may already prove that $a$ stays extreme:

If $(z, z_0)$ separates $a$ from $\mathcal{A} \setminus \{a\}$ and $(z, b) + z_0 \leq 0$, then $(z, z_0)$ also separates $a$ from $\mathcal{A} \setminus \{a\} \cup \{b\}$ and by the reverse of Lemma 2.5 it defines an infeasibility certificate for $\text{LP(frame}(A) \setminus \{a\} \cup \{b\})$.

Unfortunately, if this condition is not true, $a$ may still be extreme. In that case we need to solve the linear program to find out.

**Correctness** The correctness of Algorithm 5 should be obvious by the description above.

**Theorem 5.3 (Runtime of Algorithm 5)** The worst-case runtime of Algorithm 5 is

$$O(n' \text{LP}_{d+1,n'})$$

**Proof** In the worst-case scenario we have to solve $n' + 1$ linear programs with $d + 1$ equality constraints and $n'$ variables – one in line 1 and up to $n'$ in line 11.

The hyperplane tests are done in $O(n'd)$ (which is dominated by $\text{LP}_{d+1,n'}$) and all the rest in $O(n')$. So we get a worst-case runtime of $O(n' \text{LP}_{d+1,n'})$. □

Of course this analysis is pessimistic. We expect the runtime to be much lower in many practical cases.

### 5.3 Two or More New Points

Figure 5.3 shows a two-dimensional example, where $\mathcal{B}$ consists of five points all of which lie outside the convex hull of $\mathcal{A}$ (the dark shaded area). The extreme points of the new set $\mathcal{A} \cup \mathcal{B}$ are encircled, and for every extreme point $p$ of $\mathcal{A}$, a straight line separating $p$ from $\mathcal{A} \setminus \{p\}$ is drawn.
Algorithm 6 solves the on-line frame problem for any number of new points in the spirit of Algorithm 5.

The major difference to Algorithm 5 is that it uses a modified Dulá-Helgason Algorithm in the end. This modified algorithm is defined as follows:

**Definition 5.4 (Dulá-Helgason Algorithm with pre-initialized $\mathcal{F}$)**

By $\text{DULÁHELGASONPREINIT}(\mathcal{A}', \mathcal{F}')$ we denote the modified Dulá-Helgason Algorithm with input set $\mathcal{A} := \mathcal{A}'$ and already initialized $\mathcal{F} := \mathcal{F}'$ where $\mathcal{F}' \subseteq \text{frame}(\mathcal{A}')$ must hold as a precondition.

**Algorithm 6 Two or More Mew Points**

**Input:** $\mathcal{A}' = \text{frame}(\mathcal{A})$, $\mathcal{B}$, $\text{sep}(\mathcal{A}' \setminus \{a\}, a)$ for all $a \in \mathcal{A}'$

1: $\mathcal{C} \leftarrow \emptyset$
2: $\mathcal{F} \leftarrow \emptyset$
3: for all $b \in \mathcal{B}$ do
4:  Solve $\text{LP}(\mathcal{A}', b)$.
5:  if $b \notin \text{conv}(\mathcal{A}')$ then
6:      $\mathcal{C} \leftarrow \mathcal{C} \cup \{b\}$
7:  end if
8: end for
9: for all $a \in \mathcal{A}'$ do
10:  $(z, z_0) = \text{sep}(\mathcal{A}' \setminus \{a\}, a)$
11:  if $\langle z, b \rangle + z_0 \geq 0$ for all $b \in \mathcal{C}$ then
12:     $\mathcal{F} \leftarrow \mathcal{F} \cup \{a\}$
13:  end if
14: end for
15: $\mathcal{F} \leftarrow \text{DULÁHELGASONPREINIT}(\mathcal{C} \cup \mathcal{A}', \mathcal{F})$

**Output:** $\mathcal{F} = \text{frame}(\mathcal{A} \cup \mathcal{B})$

The algorithm starts by checking every point $b \in \mathcal{B}$ for being a convex com-
5.3. Two or More New Points

Every point which is not a convex combination of \( \text{frame}(\mathcal{A}) \) gets added to the list of candidates \( C \).

Then, for all points \( a \in \text{frame}(\mathcal{A}) \) the hyperplane test is made like in Algorithm 5 to find out whether it is already clear that \( a \) stays extreme. Here, this test has to be made for all points of \( B \) which are not already found to be non-extreme.

So after the second for-loop all the points which are known to be extreme are in \( \mathcal{F} \) and the points which might be extreme are either in \( C \) or in \( \text{frame}(\mathcal{A}) \setminus \mathcal{F} \). Now the Dulá-Helgason Algorithm is run with the hint that the points from \( \mathcal{F} \) are already known to be extreme (see Definition 5.4).

**Correctness** The correctness of Algorithm 5 should be obvious by the description above.

**Theorem 5.5 (Runtime of Algorithm 6)** The worst-case runtime of Algorithm 6 is

\[
O((n' + m) \text{LP}_{d+1,n'+m} + (n' + m)^2d)
\]

**Proof** In a worst-case scenario, all points from \( B \) are outside the convex hull of \( \mathcal{A} \) and no hyperplane test proves any point from \( \mathcal{A} \) to stay extreme a priori. Also all points from \( \mathcal{A} \cup \mathcal{B} \) could be extreme, so that the output-sensitivity of the Dulá-Helgason Algorithm does not help either.

In this situation, the first for-loop solves \( m \) linear programs with \( d + 1 \) equality constraints and \( n' \) variables which yields a runtime of \( O(m \text{LP}_{d+1,n'}) \).

The second loop is executed \( n' \) times and the condition in line 11 may need up to \( m \) hyperplane tests. That makes \( O(n'md) \).

Finally, the Dulá-Helgason Algorithm on \( n' + m \) points with no already known extreme points and an extreme point density of 100\% needs \( O((n' + m) \text{LP}_{d+1,n'+m} + (n' + m)^2d) \) time. And this is also our total runtime, as it clearly dominates the runtime of the two loops.

As with Algorithm 5, we can have much better runtimes for specific scenarios in practical cases. In particular, if the resulting frame gets small, Algorithm 6 can take advantage of the output-sensitive nature of the Dulá-Helgason Algorithm.

However, for increasing numbers of new points, the probability sinks that the separating hyperplane tests of line 11 prove a point to stay extreme.
5.4 Lots of New Points

If we have significantly more new points than the cardinality of the old frame, i.e. if \(|B| \gg |\text{frame}(\mathcal{A})|\), the information that the points from frame(\mathcal{A}) lie in convex position is not worth much anymore. In this situation we can as well just run any algorithm from Chapter 3 on the set \(B \cup \text{frame}(\mathcal{A})\).
Chapter 6

Implementation

This chapter describes the implementation of frame algorithms as a package for the CGAL library. The algorithms used are the Simple Algorithm (Algorithm 1 as described in Section 3.1 on page 15) and the Dulá-Helgason Algorithm (Algorithm 2 as described in Section 3.2 on page 16). However, some modifications had to be applied to allow duplicate input points and point sets that are not in general position. In the implementation, homogeneous coordinates are used directly as explained in Section 2.4.

The algorithm from Ottmann, Schuierer and Soundaralakshmi [16] was also considered but not implemented. This algorithm is similar to the Dulá-Helgason Algorithm and is also output-sensitive. However it is not really suited for points which are not in general position. The reason is that the algorithm repeatedly classifies all points from a facet of the convex hull of the input points as extreme points. This is okay if the input points are in general position. But for degenerate point sets some of these points may not be extreme. Unless it is clear that all these points have to be extreme (for example by the number of them) we would have to find the extreme points among them by solving the problem a dimension smaller. And then the same problem can occur until we are down to one dimension. This recursion would add an additional factor of $d$ to the runtime. The second reason for not implementing it is that the Dulá-Helgason Algorithm performed better in the extensive comparison of Dulá and López [9].

Section 6.1 contains some background information on CGAL and an overview of the functionality of the built package. Section 6.2 discusses why we allow duplicated points and point sets not in general position, and explains how this is implemented. Finally, two possible improvements are described in Section 6.3.
6. Implementation

6.1 CGAL Package

6.1.1 CGAL’s Philosophy

CGAL [1] stands for Computational Geometry Algorithms Library and is an open source project devoted to efficient and reliable geometric algorithms. The project sets a high value on exact calculation of geometric predicates. Exact arithmetic is important for us as we are dealing with combinatorial algorithms in geometry, where a wrong decision due to rounding errors of some limited number representation could lead to a different program flow that completely changes the output.

The package created during this master thesis deals with extreme points in arbitrary dimensions and is therefore called Extreme_points.d. The package contains two algorithms for computing the extreme points of a static point set and a class for computing the extreme points of a point set which can grow dynamically. We only provide a brief overview of the package here. For more details, see the user manual in Appendix A.1 and the reference manual in Appendix A.2.

6.1.2 Static Point Set

The basic functionality of our package is to compute the extreme points of a static point set. As stated before, the two chosen algorithms are the Simple Algorithm and the Dulá-Helgason Algorithm. They are described in Chapter 3 and their implementation is discussed in the Sections 6.2.1 and 6.2.2.

6.1.3 On-Line Setting

Our package features a class which maintains a set of points which can be increased dynamically. It provides the following queries: get_extreme_points() and classify(). The query get_extreme_points() computes the extreme points of the current point set. The class remembers the result of the last extreme point computation and can therefore ignore all the points previously found not to be extreme (see Chapter 5). The query classify() classifies a point in reference to the convex hull of the current point set. A query point can either be an extreme point of the current point set, some other point inside the convex hull or it can be outside the convex hull. If the user assures that the query point is in the current point set, the query will be faster as it is clear that it cannot be an outside point.
6.2 No General Position Assumption

The papers [9] and [16] assume the input points to be in general position and refer to the technique of simulating simplicity [11] to cope with degenerate inputs. In [10] degenerate inputs are considered but seen as implementation problem.

In the spirit of CGAL’s philosophy and for usability reasons we do not make any general position assumptions. We do not assume the points in the input to be unique either, so any point could happen to appear multiple times in the input.

If we would apply the simulating simplicity technique, our algorithms would treat the points like they were very slightly moved, which can result in incorrect outcome. To illustrate this, just imagine that all input points are on a straight line. There are exactly two extreme points - the first and the last point on that line. But applying simulating simplicity would turn many of the points in between into extreme points as well.

For the case of duplicated points we must first define our desired behaviour. How should we treat a point that appears multiple times in the input? The frame problem specifies as input a set of points. Very strictly seen, a sequence containing duplicate points would not be a valid input. But in real-life applications we would like to handle this case too. The way we do this is as if only one point of them was given.

The following subsections discuss how these degenerate inputs are dealt with in the implementation of the two algorithms.

6.2.1 Simple Algorithm

The Simple Algorithm can handle points not in general position naturally. Remember that the algorithm tests all points \( p \in \mathcal{P} \) by solving \( \text{LP}(\mathcal{P}\setminus \{p\}, p) \). This linear program classifies every point correctly as long as it is not multiple times in the input.

An extreme point which appears in the input multiple times will not be detected as an extreme point as the LP-solver will say that it is a convex combination of itself (the point will appear in the left hand side of the linear program too). Therefore we have to eliminate duplicate points in a pre-processing step first. We do this by lexicographical sorting and comparing successive points.
6. Implementation

6.2.2 Dulá-Helgason Algorithm

If we run the Dulá-Helgason Algorithm on a point set not in general position it could happen that some points on the boundary of \( \text{conv}(\mathcal{P}) \) are tagged as extreme points but are in fact inner points as they lie completely inside some face of \( \text{conv}(\mathcal{P}) \). The problem is that the algorithm chooses any point with maximal inner product to \( z \) in line 11. To fix that we just have to make sure that the point chosen there is extreme. We can achieve this by taking the lexicographically smallest point among the points with maximal inner product. This way we chose an extreme point \( p^* \) among all the points from \( \mathcal{P} \) which lie on a common face of \( \text{conv}(\mathcal{P}) \) and therefore \( p^* \) is also an extreme point of all the points.

If a point appears multiple times in the input our algorithm should behave like this point was only given once. Having duplicated points is not a big problem here. The only thing to worry is that some duplicated point \( p \) will be in the output multiple times. Now we have two possibilities: either \( p \) is an extreme point or \( p \) is in the convex hull of the other points. In the latter case \( p \) is not part of the output so we are fine. For the former case let \( p_1 = \cdots = p_k = p \) be the duplicates of the extreme point \( p \). Without loss of generality \( p_1 \) is the point first detected as extreme point (by the correctness of the algorithm, one of them has to be detected as extreme point) and will be part of the output. After that, all the other duplicates \( p_i \) will be contained in the convex hull of the already found extreme points \( \mathcal{F} \) as \( p \in \mathcal{F} \). Hence they will not be marked as extreme points and that is exactly what we wanted. Therefore the Dulá-Helgason Algorithm is correct even if some points appear more than once in the input, and no preprocessing like in the Simple Algorithm is necessary.

6.3 Possible Improvements

6.3.1 LP Hot Starts

Subsequent linear programs which are solved in both algorithms do not change by much. For example in the Dulá-Helgason Algorithm it happens often that only the right-hand side changes. This condition can be exploited by a technique called “hot starts” with which substantial time savings can be achieved [9, p.8]. Unfortunately the CGAL LP-solver does not support this technique at this time. Also patching the solver to include this feature would have been too much work. Therefore “hot starts” are not used in the implementation.
6.3. Possible Improvements

6.3.2 Floating-point Filtering Technique

Another improvement is to use a floating-point filtering technique for calculating the minimal inner product in line 11 of the Dulá-Helgason Algorithm. In short this works as follows (see [5] for more details): Instead of calculating everything with an exact type, a fast but inexact type (like double) is tried first. Using error analysis, every value is replaced by an interval in which the value lies for sure. A comparison of two values can be decided directly if the intervals of two values do not intersect. Otherwise nothing can be said for sure and exact values have to be taken. In many cases the values are apart enough so that no exact computation is needed at all.

In our case filtering could be applied like this: First the intervals of all involved scalar products are calculated. The value with the smallest upper bound $u_{\text{min}}$ is our first candidate. If the lower bounds of all other values are greater than $u_{\text{min}}$, the minimum is found without using any exact arithmetic. Otherwise all the values with lower bound less than $u_{\text{min}}$ are candidates too and we have to calculate the exact scalar products among them to find the minimum.

As inexact floating point operations can be done very quickly compared to operations on exact types, this technique can save a big amount of time without sacrificing exactness.
In this chapter we analyze how the two implemented algorithms perform on various input data. In particular we look at output sensitivity, the influence of the number of input points and the influence of the dimension.

What needs to be mentioned beforehand is that the use of an exact LP-solver significantly affects the runtimes. With non-exact solvers, runtime improvements of a factor 40 or more are possible. The comparison of the two algorithms in this setting changes drastically (in favor of the Dulá-Helgason Algorithm). However, we will only look at runtimes resulting from the usage of an exact number type. As CGAL’s LP-solver is built and optimized for exact data types, an analysis using an inexact type would not make sense.

All the tests were run on a GNU/Linux system with an x86.64 AMD Athlon™ 3200+ CPU running at 2.2GHz and 2GB of RAM. The CGAL library was used in version 3.7.

The times measured for both algorithms are the total runtime and the amount of it which is spent on solving linear programs. The latter is denoted by “LPs” in the plots. For the Dulá-Helgason Algorithm the amount of time spent for calculating and comparing inner products (i.e. time spent in line 11 of Algorithm 2) is measured too and denoted by “IPs” in the plots. The raw measurement data is listed in Appendix B.

7.1 Synthetic Data from Dulá and López

The first set of input data is the one which was generated and used in [9] by Dulá and López. It features random point sets parameterized by three variables: the number of points \( n \), the dimension \( d \) and the extreme point
7. Experimental Results

Figure 7.1: Influence of $p$, $n = 10000$, $d = 20$

Figure 7.1 shows the influence of the extreme point density. “S” denotes the Simple Algorithm and “D” the Dulá-Helgason Algorithm. We can clearly see the output-sensitivity of the Dulá-Helgason Algorithm. Interestingly the Simple Algorithm spends the longest time for the input with 1% density where it needs more than twice as long as the Dulá-Helgason Algorithm. For $p = 25\%$ the runtimes are already similar. Here, the Dulá-Helgason Algorithm still spends significantly less time on solving LPs than the Simple Algorithm. About a third of the time is spent on calculating and comparing inner products. This is an indication that the filtering techniques discussed in Section 6.3.2 could really pay off for input sets whose extreme point density is not that low.

Figure 7.2 shows the influence of the dimension. We clearly see that the runtime of both algorithms is more than linear in $d$. As everything except solving the linear programs is clearly linear in $d$ this effect comes from our linear programming solver.

Figure 7.3 shows the influence of the number of input points. We observe an almost linear dependency but we have to be careful as we do not have many measuring points. Note that the constant extreme point density here means that the number of extreme points grows with $n$ too. This means that both algorithms should exhibit a quadratic behaviour. This can be observed clearly in Figure 7.6.
7.1. Synthetic Data from Dulá and López

Figure 7.2: Influence of $d$, $n = 10000$, $p = 15$

Figure 7.3: Influence of $n$, $d = 20$, $p = 25$
7. Experimental Results

### Table 7.1: Banking Data Characteristics

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cardinality ( n )</th>
<th>Dimension ( d )</th>
<th>Extreme point density ( p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4971</td>
<td>7</td>
<td>6.94%</td>
</tr>
<tr>
<td>2</td>
<td>12456</td>
<td>8</td>
<td>5.74%</td>
</tr>
<tr>
<td>3</td>
<td>19939</td>
<td>11</td>
<td>24.88%</td>
</tr>
</tbody>
</table>

![Figure 7.4: Banking Data](image)

#### 7.2 Banking Data from Dulá and López

Our next data set comes also from Dulá and López [9] and is interesting for the reason that it features real life data from US commercial-banks. Table 7.1 shows the characteristics of the inputs. As before \( n \) is the number of input points, \( d \) the dimension and \( p \) the extreme point density.

Figure 7.4 shows the performance of both algorithms on the 3 test cases. “S” denotes the Simple Algorithm and “D” the Dulá-Helgason Algorithm. The third test case shows that also in real life applications the Simple Algorithm can have a similar runtime.

#### 7.3 Cyclic polytopes

A third set of input data was generated as vertices of cyclic polytopes with parameters chosen uniformly at random between 0 and 1. A cyclic polytope
7.3. Cyclic polytopes

Figure 7.5: Cyclic Polytopes: Influence of $d$, $n = 10000$

is defined as follows.

**Definition 7.1 (Cyclic polytope)** The $d$-dimensional cyclic polytope with parameters $t_1, t_2, ..., t_n \in \mathbb{R}$, $n > d$ is defined as the following convex hull:

$$C_d(t_1, t_2, ..., t_n) := \text{conv}(x(t_1), x(t_2), ..., x(t_n))$$

Where $x(t)$ is the moment curve in $\mathbb{R}^d$ defined by:

$$x(t) := \begin{pmatrix} t \\ t^2 \\ \vdots \\ t^d \end{pmatrix} \in \mathbb{R}^d$$

All the points $x(t_i)$ are vertices of the cyclic polytope $C_d(t_1, t_2, ..., t_n)$ (i.e. they lie in convex position, see [19, p. 11ff] for the proof). Therefore our input sets have an extreme point density of 100% this time.

The variable parameters for the generated sets are the number of points $n$ and the dimension $d$.

Figure 7.5 shows the influence of the dimension. Compared to the tests from Figure 7.2 the results are less dramatic here but the tendency is the same. In this scenario the Simple Algorithm performs significantly better than the Dulá-Helgason Algorithm, which is what we expected, as the extreme point density of 100% is a worst-case scenario for an output-sensitive
algorithm. Interestingly, the Simple Algorithm seems to perform better with cyclic polytopes than with the banking data from Dulá and López. This is an indication that the performance of the underlying LP-solver also depends on the nature of the input sets. Therefore, comparisons across different types of input sets have to be made with care.

Figure 7.6 shows the influence of the number of input points. Here we can see clearly the quadratic tendency we expected to see already in Figure 7.3 (remember that the number of extreme points grows here too). For increasing \( n \) the runtime of the Simple Algorithm really outperforms the one from the Dulá-Helgason Algorithm in this scenario – for \( n = 30000 \) it is twice as fast. However we see also that the Dulá-Helgason Algorithm still spends less time in solving linear programs. This comes from the fact, that the linear programs from the Dulá-Helgason Algorithm have only as much variables as extreme points are found so far. So in this scenario, the algorithm starts with a one-variable LP and then one variable gets added each time until it is a \( n - 1 \) variable LP. In contrast, the Simple Algorithm solves \( n \) full LPs with \( n - 1 \) variables each.

Both Figure 7.6 and Figure 7.3 show also the big potential the floating-point filters discussed in Section 6.3.2 have, as a substantial amount of time the Dulá-Helgason Algorithm spends is calculating and comparing inner products.
Chapter 8

Conclusion

In higher dimensions the extreme point problem is substantially easier than the convex hull problem as in contrast to the latter it can be solved in polynomial time by using linear programming.

The thesis presents the implementation of a package for the CGAL library featuring a straightforward and an output-sensitive frame algorithm. Exact arithmetic is used and no assumptions are made for the input points.

In an on-line setting where extreme point computations are made repeatedly on an increasing point set, every point which is found to be non-extreme can be ignored in any future extreme point computation.

Unfortunately no algorithm has been found which solves the sparse problem significantly faster than the dense problem.
Appendix A

CGAL Package Manuals

A.1 User Manual

Extreme_points_d

Separate Build

November 17, 2010
Chapter 1

dD Extreme points

Christian Helbling

Contents

1.1 Introduction .............................................. 1
1.2 Static extreme points computations ......................... 1
1.3 Extreme points computations for a dynamically increasing point set .......... 3
1.4 Demo ........................................... 5
1.5 Classified Reference Pages .................................. 7
1.6 Alphabetical List of Reference Pages ........................ 8

1.1 Introduction

A subset \( S \subseteq \mathbb{R}^d \) is convex if for any two points \( p \) and \( q \) in the set the line segment with endpoints \( p \) and \( q \) is contained in \( S \). The convex hull of a set \( S \) is the smallest convex set containing \( S \). The convex hull of a finite set of points \( A \in \mathbb{R}^d \) is a convex polytope with vertices in \( A \). A point in \( A \) is an extreme point (with respect to \( A \)) iff it is a vertex of the convex hull of \( A \). This also means that a point in \( A \) is an extreme point iff it is not contained in the convex hull of all the other points in \( A \).

This chapter describes the functions and class provided in CGAL for computing the extreme points of arbitrary dimensional point sets. There are two ways to compute extreme points of arbitrary dimensional point sets in CGAL: using a static algorithm or using the dynamic class `Extreme_points_d<Traits>`. The class `Extreme_points_d<Traits>` maintains a semi-dynamic point set and answers extreme point queries.

1.2 Static extreme points computations

The function `extreme_points_d_simple` computes the extreme points of a d-dimensional point set using a straightforward algorithm based on linear programming.

For point sets with low extreme point density Dulá and Helgason invented an output-sensitive algorithm [DH96]. The function `extreme_points_d_dula_helgason` provides an implementation of this algorithm.
Example

The following program computes the extreme points of a set of 500 random points chosen from a 10-dimensional iso box uniformly at random. The same results could be achieved by substituting the function `CGAL::extreme_points_d_dula_helgason` by `CGAL::extreme_points_d_simple`.

Note that although `double` is used here as the field number type for the Kernel does not mean that the algorithm will not be exact. Internally an appropriate exact number type is used for intermediate results.

```cpp
#include <CGAL/point_generators_d.h>
#include <CGAL/algorithm.h>
#include <vector>
#include <iostream>
#include <CGAL/Cartesian_d.h>
#include <CGAL/Extreme_points_d.h>

typedef CGAL::Cartesian_d<double> Kernel_d;
typedef Kernel_d::Point_d Point_d;

int main() {
    const int D = 10;
    const int N = 500;

    // D dimensional points with coordinates in the range [-10, 10]
    CGAL::Random_points_in_iso_box_d<Point_d> gen (D, 10);

    // generate N points randomly in the D dimensional iso box
    // and copy them to a vector
    std::vector<Point_d> points;
    CGAL::copy_n( gen, N, std::back_inserter(points) );

    // compute the extreme points
    std::vector<Point_d> extreme_points;
    CGAL::extreme_points_d_dula_helgason(points.begin(), points.end(),
                                         std::back_inserter(extreme_points));

    std::cout<<"Found "<<extreme_points.size()
             " extreme points:"<<std::endl;
    for (std::vector<Point_d>::iterator it=extreme_points.begin();
         it!=extreme_points.end();
         ++it) {
        std::cout<<*it<<std::endl;
    }
    return 0;
}
```

File: examples/Extreme_points_d/extreme_points_d_dula_helgason.cpp
1.3 Extreme points computations for a dynamically increasing point set

For extreme point queries on a growing point set the class `Extreme_points_d<Traits>` can be used. This class maintains a set of points in an arbitrary (but fixed) dimension. Extreme point computations are done lazily (i.e. only on extreme point queries) and the result of the last extreme point computation is kept and used in the next computation.

The class can also classify query points in relation to the convex hull of the current point set. Such a query point is either one of the extreme points, some other point inside the convex hull or completely outside the convex hull (which means it would be an extreme point when added to the point set).

Example

The following program demonstrates the usage of `Extreme_points_d<Traits>`. Randomly chosen points are added to the current point set in multiple batches. In between, the extreme points of the current set are calculated and some classify queries are made.

```cpp
#include <CGAL/Cartesian_d.h>
#include <CGAL/point_generators_d.h>
#include <CGAL/Extreme_points_d.h>
#include <CGAL/Extreme_points_traits_d.h>
#include <vector>
#include <iostream>
#include <cassert>

typedef CGAL::Cartesian_d<double> Kernel_d;
typedef Kernel_d::Point_d Point_d;
typedef CGAL::Extreme_points_traits_d<Point_d> EP_Traits_d;

int main() {
    const int D = 5; // dimension
    const int N = 100; // number of points for every batch
    const int BATCHES = 3; // number of batches

    CGAL::Extreme_points_d<EP_Traits_d> ep(D);

    // Generator for D-dimensional points with coordinates
    // in the range [-10, 10]
    CGAL::Random_points_in_iso_box_d<Point_d> gen (D, 10.);

    for (int i=0;i<BATCHES;++i) {
        std::cout<<"Inserting the points:"<<std::endl;
        for (int j=0;j<N;++j) {
            Point_d p = *gen++;
            std::cout<<p<<"<"<<std::endl;
            points.push_back(p);
        }
        // add these points to the current point set maintained by ep
        ep.insert(points.begin(), points.end());
    }
    return 0;
}
```
// compute the extreme points
std::cout<<"\nExtreme points of the current set: "<<std::endl;
std::vector<Point_d> extreme_points;
ep.get_extreme_points(std::back_inserter(extreme_points));
for (std::vector<Point_d>::iterator it=extreme_points.begin();
it!=extreme_points.end(); it++) {
    std::cout<<*it<<std::endl;
}

// we can use classify to see whether
// some specific point was extreme
if (ep.classify(points[0], true) == CGAL::EXTREME_POINT) {
    std::cout<<"The point "<<points[0]
<<" is an extreme point."<<std::endl;
} else {
    std::cout<<"The point "<<points[0]
<<" is not an extreme point."<<std::endl;
}

// we can also classify some other random point
Point_d p = *gen++;
switch (ep.classify(p)) {
    case CGAL::INTERNAL_POINT:
        std::cout<<"The point "<<p<<" is inside the "
        "convex hull of the current point set 
        "(but not an extreme point)."<<std::endl;
        break;
    case CGAL::EXTREME_POINT:
        // the chance that this happens is practically zero.
        std::cout<<"The point "<<p<<" is an extreme point 
        "of the current point set."<<std::endl;
        break;
    case CGAL::EXTERNAL_POINT:
        std::cout<<"The point "<<p<<" is outside the "
        "convex hull of the current point set." 
        "<<std::endl;
        break;
    default:
        std::cerr<<"Extreme_points_d<Point_d>::classify "
        "returned unexpected answer!"<<std::endl;
        assert(0);
        std::cout<<std::endl;
}
return 0;
}
1.4 Demo

This package comes with an interactive demo of Extreme points $d$<Traits$>. For simplicity and for better interaction and imagination the demo operates only with 2-dimensional points. Source code: <CGAL_ROOT>/demo/Extreme_points_d/Extreme_points_2.cpp).

![Demo of Extreme points $d$<Traits$> in 2 dimensions](image)

Figure 1.1: Demo of Extreme points $d$<Traits$> in 2 dimensions
A.2 Reference Manual

dD Extreme Points
Reference Manual

Christian Helbling

1.5 Classified Reference Pages

Concepts
ExtremePointsTraits\text{\texttt{d}} \hspace{1cm} \text{page 12}

Traits Classes
\texttt{CGAL::Extreme\_points\_traits\_d\langle Point\rangle} \hspace{1cm} \text{page 17}

Classes
\texttt{CGAL::Extreme\_points\_d\langle Traits\rangle} \hspace{1cm} \text{page 15}
\texttt{CGAL::Extreme\_points\_options\_d} \hspace{1cm} \text{page 18}

Global Functions
\texttt{CGAL::extreme\_points\_d} \hspace{1cm} \text{page 9}
\texttt{CGAL::extreme\_points\_d\_dula\_helgason} \hspace{1cm} \text{page 10}
\texttt{CGAL::extreme\_points\_d\_simple} \hspace{1cm} \text{page 11}

Enum
\texttt{CGAL::Extreme\_point\_classification} \hspace{1cm} \text{page 14}
\texttt{CGAL::Extreme\_point\_algorithm\_d} \hspace{1cm} \text{page 13}
1.6 Alphabetical List of Reference Pages

- `Extreme_point_algorithm_d` ................................................................. page 13
- `Extreme_point_classification` ......................................................... page 14
- `extreme_points_d` ........................................................................ page 9
- `extreme_points_d_dula_helgason` ..................................................... page 10
- `extreme_points_d_simple` ............................................................... page 11
- `Extreme_points_d<Traits>` ............................................................... page 15
- `Extreme_points_options_d` ............................................................... page 18
- `ExtremePointsTraits_d` ................................................................. page 12
- `Extreme_points_traits_d<Point>` ....................................................... page 17
CGAL::extreme_points_d

Definition
The function `extreme_points_d` computes the extreme points of the given set of input points.

```cpp
#include <CGAL/Extreme_points_d.h>

template <class InputIterator, class OutputIterator, class ExtremePointsTraits_d>
OutputIterator extreme_points_d(InputIterator first, InputIterator beyond, OutputIterator result, ExtremePointsTraits_d ep_traits = Default_traits)
```

computes the extreme points of the point set in the range `[first, beyond)`. The resulting sequence of extreme points is placed starting at position `result`, and the past-the-end iterator for the resulting sequence is returned.

The default traits class `Default_traits` is `Extreme_points_traits_d::Point` where `Point` is `InputIterator::value_type`.

Requirements
`InputIterator::value_type` and `OutputIterator::value_type` are equivalent to `ExtremePointsTraits_d::Point`.

See Also
- CGAL::extreme_points_d_simple ........................................ page 11
- CGAL::extreme_points_d_dula_helgason ................................ page 10
- CGAL::Extreme_points_d<Traits> ........................................ page 15

Implementation
At the moment this is just `extreme_points_d_dula_helgason`. However, the idea is that this function chooses the most appropriate algorithm based on some heuristics.

Example
See the example of `extreme_points_d_dula_helgason` as its interface is exactly the same as the one of `extreme_points_d`.

`Extreme_points_d/extreme_points_d_dula_helgason.cpp`

9
Function

CGAL::extreme_points_d_dula_helgason

Definition

The function `extreme_points_d_dula_helgason` computes the extreme points of the given set of input points.

```cpp
#include <CGAL/Extreme_points_d.h>

template <class InputIterator, class OutputIterator, class ExtremePointsTraits>
OutputIterator extreme_points_d_dula_helgason(InputIterator first, InputIterator beyond, OutputIterator result, ExtremePointsTraits ep_traits = Default_traits)
```

computes the extreme points of the point set in the range `[first, beyond)`. The resulting sequence of extreme points is placed starting at position `result`, and the past-the-end iterator for the resulting sequence is returned.

The default traits class `Default_traits` is `Extreme_points_traits_d<Point>` where `Point` is `InputIterator::value_type`.

Requirements

`InputIterator::value_type` and `OutputIterator::value_type` are equivalent to `ExtremePointsTraits_d::Point`.

See Also

- `CGAL::Extreme_points_d<Traits>` ........................................ page 15
- `CGAL::extreme_points_d<simple>` .................................. page 11
- `CGAL::extreme_points_d` ............................................. page 9

Implementation

This function implements the extreme points algorithm from Dulá and Helgason [DH96] as also described in [DL08] and [He10].

This algorithm requires $O(d(m+n \cdot LP_{a+b}))$ time in the worst case where $n$ is the number of input points, $m$ the number of extreme points, $d$ the dimension and $LP_{a+b}$ the runtime for solving a linear program with $a$ equality constraints and $b$ nonnegative variables using CGAL’s QP solver package.

Example

`Extreme_points_d/extreme_points_d_dula_helgason.cpp`
CGAL::extreme_points_d_simple

Definition
The function `extreme_points_d_simple` computes the extreme points of the given set of input points.

```cpp
#include <CGAL/Extreme_points_d.h>

template <class InputIterator, class OutputIterator, class ExtremePointsTraits>
OutputIterator extreme_points_d_simple(InputIterator first,
                                        InputIterator beyond,
                                        OutputIterator result,
                                        ExtremePointsTraits ep_traits = Default_traits)
```

computes the extreme points of the point set in the range `[first, beyond)`. The resulting sequence of extreme points is placed starting at position `result`, and the past-the-end iterator for the resulting sequence is returned.

The default traits class `Default_traits` is `Extreme_points_traits_d<Point>` where `Point` is `InputIterator::value_type`

Requirements
`InputIterator::value_type` and `OutputIterator::value_type` are equivalent to `ExtremePointsTraits::Point`.

See Also
- `CGAL::Extreme_points_d<Traits>` on page 15
- `CGAL::extreme_points_d_dula_helgason` on page 10
- `CGAL::extreme_points_d_simple` on page 9

Implementation
This function implements a straightforward implementation of the extreme points algorithm using linear programming. Every point is tested for being an extreme point by a linear program involving all the other points. The algorithm is described in more detail in [Hel10].

This algorithm requires $O(n \cdot LP_{a,b} + 1, n - 1)$ time where $n$ is the number of input points, $d$ the dimension and $LP_{a,b}$ the runtime for solving a linear program with $a$ equality constraints and $b$ nonnegative variables using CGAL’s QP solver package.

Example
See the example of `extreme_points_d_dula_helgason` as its interface is exactly the same as the one of `extreme_points_d_simple`:

```cpp
Extreme_points_d/extreme_points_d_dula_helgason.cpp
```
**ExtremePointsTraits\_d**

**Definition**

Requirements of the traits class to be used with the functions `extreme_points\_d`, `extreme_points\_d\_dula\_helgason`, `extreme_points\_d\_simple` and the class `Extreme_points\_d`.

**Types**

- `ExtremePointsTraits\_d:: Point`: The point type on which the extreme point algorithm operates.
- `ExtremePointsTraits\_d:: RT`: The `RingNumberType` used for the homogeneous coordinates of the input points.
- `ExtremePointsTraits\_d:: Less\_lexicographically`: Binary predicate object type comparing `Points` lexicographically. Must provide `bool operator() (Point p, Point q)` where `true` is returned if `p` is lexicographically smaller than `q`.
- `ExtremePointsTraits\_d:: Homogeneous\_begin`: Function object type that provides `RandomAccessIterator operator() (Point &p)`, which returns a random access iterator over the homogeneous coordinates of `p` pointing to the zeroth homogeneous coordinate $h_0$ of `p`. The value type of this random access iterator (i.e. the type of the homogeneous coordinates) must be `RT`.

**Has Models**

`CGAL::Extreme_points_traits\_d<\text{Point}>`
CGAL::Extreme_point_algorithm_d

#include <CGAL/Extreme_points_options_d.h>


This is an enumeration type used to specify an extreme point algorithm in `Extreme_points_options_d`.

The following options are available:

**EP_CHOOSE_APPROPRIATE**

This is the default value of the algorithm in `Extreme_points_options_d`, and it lets the implementation choose the algorithm that it thinks is most appropriate for current situation.

**EP_SIMPLE**

This is the straightforward algorithm `extreme_points_d_simple`. If most of the input points are extreme points, this algorithm can be faster than `EP_DULA_HELGASON`.

**EP_DULA_HELGASON**

This is the output-sensitive algorithm from Dulá and Helgason [DH96] `extreme_points_d_dula_helgason`. If a small fraction of the input points are extreme points, this algorithm performs significantly better than `EP_SIMPLE`.

See Also

[CGAL::Extreme_points_options_d](#)
Enum CGAL::Extreme_point_classification

```
#include <CGAL/Extreme_points_d.h>

enum Extreme_point_classification {INTERNAL_POINT, EXTREME_POINT, EXTERNAL_POINT};
```

Enum to classify a query point in relation to the convex hull of some point set. The query point can be either completely outside the convex hull (EXTERNAL_POINT), an extreme point (EXTREME_POINT) or some point inside the convex hull (INTERNAL_POINT). Note that internal points can also be located at the boundary of the convex hull.

See Also

`CGAL::Extreme_points_d<Traits>`
Definition

The class `CGAL::Extreme_points_d<Traits>` holds a set of $d$ dimensional points and answers extreme point queries. The point set can be enlarged dynamically. Extreme point computations are done lazily (i.e. only when a query has to be answered) and the result of the last computation is kept. There is also the possibility to classify points relative to the convex hull of the current point set (i.e. to tell whether they are inside, outside or an extreme point).

```cpp
#include <CGAL/Extreme_points_d.h>
```

Types

The following types come directly from the traits class given as the template argument which must be a model of the concept `ExtremePointsTraits_d`.

- `Extreme_points_d<Traits>::Point`: The type of the input points.
- `Extreme_points_d<Traits>::Less_lexicographically`: The lexicographic compare functor for `Point`.
- `Extreme_points_d<Traits>::RT`: The number type, which is the ring type of the input points.

Creation

```cpp
Extreme_points_d<Traits> ep( int d, Extreme_points_options_d ep_options = Extreme_points_options_d());
```

Constructor for extreme points computations in $d$ dimensions. The optional argument `ep_options` can be used to set some options (see `Extreme_points_options_d`).

Operations

- `int ep.dimension()`: Returns the dimension of the points
- `void ep.clear()`: Clears the point set
- `void ep.insert( const Point p)`: Adds the point $p$ to the point set

```cpp
template <typename InputIterator>
```

15
void ep.insert( InputIterator first, InputIterator beyond)
Adds all the points from the range [first,beyond) to the point set

template <class OutputIterator>
OutputIterator ep.get_extreme_points( OutputIterator result)
Calculates the extreme points of the current point set. The resulting sequence of extreme points is placed starting at position result, and the past-the-end iterator for the resulting sequence is returned.

enum Extreme_point_classification

ep.classify( Point p, bool is_input_point=false)
Classifies point p relative to the convex hull of the current point set. If p is an input point the argument is_input_point may be set to true which speeds up the query.
Precondition: p is an input point or is_input_point=false.

Requirements

Traits is a model of the concept ExtremePointsTraits_d.

See Also

CGAL::extreme_points_d .......................................................... page 9
CGAL::extreme_points_d_delaunay ............................................ page 10
CGAL::extreme_points_d_simple .............................................. page 11
CGAL::Extreme_points_traits_d<Point>

Definition

The class `Extreme_points_traits_d<Point>` serves as a traits class for the class `Extreme_points_d<Traits>` and for the functions `extreme_points_d`, `extreme_points_d_dula_helgason` and `extreme_points_d_simple`. This is the default traits class for these two functions.

Note that this class is implemented by template specialization, so it can be used only with specific types of `Point` (see Requirements).

```
#include <CGAL/Extreme_points_traits_d.h>
```

Is Model for the Concepts

ExtremePointsTraits_d ............................................................ page 12

Requirements

This class requires that `Point` is one of `CGAL::Point_2<Kernel>`, `CGAL::Point_3<Kernel>` and `CGAL::Point_d<Kernel>`.

See Also

CGAL::extreme_points_d_dula_helgason ......................................... page 10
CGAL::extreme_points_d_simple .................................................. page 11
CGAL::extreme_points_d ............................................................. page 9
CGAL::Extreme_points_d<Traits> ................................................... page 15
CGAL::Extreme_points_options_d

Definition
The class `Extreme_points_options_d` is used for passing options to the class `Extreme_points_d<Traits>`. Currently the only option is which algorithm is chosen for the extreme point computations.

Some more options can be added in the future.

```cpp
#include <CGAL/Extreme_points_options_d.h>
```

Creation

```cpp
Extreme_points_options_d ep_options( Extreme_point_algorithm_d algo = EP_DULA_HELGASON);
```

Constructs an instance of `Extreme_points_options_d`. If `algo` is specified it is set as the chosen extreme point algorithm.

Operations

```cpp
void ep_options.set_algorithm( Extreme_point_algorithm_d algo)
```

Sets the algorithm used for extreme point computations to `algo`. For more information see `Extreme_point_algorithm_d`.

```cpp
Extreme_point_algorithm_d
```

```cpp
ep_options.get_algorithm() 
```

Returns the algorithm used for extreme point computations.

See Also

- `CGAL::Extreme_points_d<Traits>` page 15
- `CGAL::Extreme_point_algorithm_d` page 13
Bibliography


Index

Pages on which definitions are given are presented in boldface.

classify
Extreme\_points\_d, 16

clear
Extreme\_points\_d, 15

dimension
Extreme\_points\_d, 15

\texttt{EP\_CHOOSE\_APPROPRIATE}, 13
\texttt{EP\_DULA\_HELGASON}, 13
\texttt{EP\_SIMPLE}, 13

\texttt{EXTERNAL\_POINT}, 14

\textit{extreme point}, 1
\texttt{EXTREME\_POINT}, 14

\texttt{Extreme\_point\_algorithm\_d}, 13
\texttt{Extreme\_point\_classification}, 14
\texttt{Extreme\_points\_d}, 15–16

\texttt{extreme\_points\_d}, 9
\texttt{extreme\_points\_d\_dula\_helgason}, 10

\texttt{traits class}
\texttt{default}, 17

\texttt{extreme\_points\_d\_simple}, 11

\texttt{traits class}
\texttt{default}, 17

\texttt{Extreme\_points\_options\_d}, 18
\texttt{Extreme\_points\_traits\_d}, 17
\texttt{Extreme\_points\_Traits\_d}, 12

\texttt{get\_algorithm}
\texttt{Extreme\_points\_options\_d}, 18

\texttt{get\_extreme\_points}
\texttt{Extreme\_points\_d}, 16

\texttt{insert}
\texttt{Extreme\_points\_d}, 15

\texttt{INTERNAL\_POINT}, 14

\texttt{set\_algorithm}
\texttt{Extreme\_points\_options\_d}, 18
Appendix B

Runtime Measurements

Legend

n  Number of points
 d  Dimension
 p  Extreme point density
 LPs Time used to solve linear programs
 IPs Time used to calculate and compare inner products

Table B.1: Synthetic Data from Dulá and López

<table>
<thead>
<tr>
<th>n</th>
<th>d</th>
<th>p[%]</th>
<th>Simple Algorithm</th>
<th></th>
<th></th>
<th>Runtime [s]</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>LPs</td>
<td>Rest</td>
<td>Total</td>
<td>LPs</td>
<td>IPs</td>
<td>Rest</td>
<td>Total</td>
</tr>
<tr>
<td>2500</td>
<td>5</td>
<td>1</td>
<td>4.02</td>
<td>0.00</td>
<td>4.02</td>
<td>1.88</td>
<td>0.31</td>
<td>0.02</td>
<td>2.21</td>
</tr>
<tr>
<td>2500</td>
<td>5</td>
<td>15</td>
<td>3.88</td>
<td>0.00</td>
<td>3.88</td>
<td>2.27</td>
<td>3.54</td>
<td>0.01</td>
<td>5.81</td>
</tr>
<tr>
<td>2500</td>
<td>5</td>
<td>25</td>
<td>4.22</td>
<td>0.00</td>
<td>4.22</td>
<td>2.67</td>
<td>5.91</td>
<td>0.01</td>
<td>8.59</td>
</tr>
<tr>
<td>2500</td>
<td>10</td>
<td>1</td>
<td>22.08</td>
<td>0.00</td>
<td>22.08</td>
<td>13.93</td>
<td>0.50</td>
<td>0.04</td>
<td>14.47</td>
</tr>
<tr>
<td>2500</td>
<td>10</td>
<td>15</td>
<td>20.13</td>
<td>0.01</td>
<td>20.14</td>
<td>14.49</td>
<td>6.25</td>
<td>0.02</td>
<td>20.76</td>
</tr>
<tr>
<td>2500</td>
<td>10</td>
<td>25</td>
<td>19.73</td>
<td>0.00</td>
<td>19.73</td>
<td>15.14</td>
<td>9.43</td>
<td>0.05</td>
<td>24.63</td>
</tr>
<tr>
<td>2500</td>
<td>15</td>
<td>1</td>
<td>84.42</td>
<td>0.00</td>
<td>84.42</td>
<td>43.61</td>
<td>0.83</td>
<td>0.08</td>
<td>44.52</td>
</tr>
<tr>
<td>2500</td>
<td>15</td>
<td>15</td>
<td>64.05</td>
<td>0.00</td>
<td>64.05</td>
<td>46.85</td>
<td>9.39</td>
<td>0.08</td>
<td>56.32</td>
</tr>
<tr>
<td>2500</td>
<td>15</td>
<td>25</td>
<td>67.95</td>
<td>0.00</td>
<td>67.95</td>
<td>56.81</td>
<td>14.72</td>
<td>0.07</td>
<td>71.60</td>
</tr>
<tr>
<td>2500</td>
<td>20</td>
<td>1</td>
<td>337.63</td>
<td>0.00</td>
<td>337.63</td>
<td>94.67</td>
<td>1.25</td>
<td>0.14</td>
<td>96.06</td>
</tr>
<tr>
<td>2500</td>
<td>20</td>
<td>15</td>
<td>158.20</td>
<td>0.00</td>
<td>158.20</td>
<td>128.70</td>
<td>12.67</td>
<td>0.18</td>
<td>141.55</td>
</tr>
<tr>
<td>2500</td>
<td>20</td>
<td>25</td>
<td>153.20</td>
<td>0.00</td>
<td>153.21</td>
<td>130.50</td>
<td>21.37</td>
<td>0.17</td>
<td>152.04</td>
</tr>
</tbody>
</table>
Table B.1: Synthetic Data from Dulá and López (continued)

| n    | d | p[%] | Simple Algorithm |            | Runtime [s] |            |            |
|------|---|------|------------------|------------|-------------|------------|
|      |   |      | LPs | Rest | Total | LPs | IPs | Rest | Total |
| 5000 | 5 | 1    | 13.58 | 0.01 | 13.59 | 3.83 | 1.13 | 0.03 | 4.99 |
| 5000 | 5 | 15   | 13.36 | 0.01 | 13.37 | 4.98 | 14.54 | 0.03 | 19.56 |
| 5000 | 5 | 25   | 14.62 | 0.01 | 14.62 | 6.20 | 23.88 | 0.03 | 30.11 |
| 5000 | 10| 1    | 73.61 | 0.00 | 73.61 | 25.94 | 1.93 | 0.06 | 27.92 |
| 5000 | 10| 15   | 67.81 | 0.01 | 67.82 | 30.36 | 24.99 | 0.08 | 55.43 |
| 5000 | 10| 25   | 71.76 | 0.00 | 71.76 | 33.99 | 40.38 | 0.09 | 74.47 |
| 5000 | 15| 1    | 211.18 | 0.01 | 211.19 | 94.22 | 3.10 | 0.08 | 97.40 |
| 5000 | 15| 15   | 171.91 | 0.00 | 171.92 | 108.90 | 4.53 | 0.15 | 113.56 |
| 5000 | 15| 25   | 180.75 | 0.01 | 180.76 | 108.90 | 4.53 | 0.19 | 113.56 |
| 5000 | 20| 1    | 612.48 | 0.01 | 612.48 | 264.07 | 4.53 | 0.30 | 308.91 |
| 5000 | 20| 15   | 395.29 | 0.00 | 395.29 | 258.62 | 4.53 | 0.34 | 308.07 |
| 5000 | 20| 25   | 400.68 | 0.00 | 400.68 | 285.80 | 4.53 | 0.37 | 370.09 |
| 7500 | 5 | 1    | 51.60 | 0.02 | 51.62 | 5.80  | 2.46 | 0.11 | 8.37 |
| 7500 | 5 | 15   | 50.17 | 0.01 | 50.18 | 8.14  | 3.33 | 0.07 | 41.74 |
| 7500 | 5 | 25   | 51.86 | 0.02 | 51.87 | 10.51 | 5.14 | 0.08 | 64.72 |
| 7500 | 10| 1    | 157.09 | 0.01 | 157.10 | 37.83 | 4.67 | 0.10 | 42.60 |
| 7500 | 10| 15   | 152.07 | 0.01 | 152.08 | 46.97 | 54.91 | 0.15 | 102.03 |
| 7500 | 10| 25   | 156.14 | 0.02 | 156.16 | 54.37 | 89.65 | 0.17 | 144.18 |
| 7500 | 15| 1    | 368.08 | 0.02 | 368.10 | 139.21 | 6.57 | 0.26 | 146.04 |
| 7500 | 15| 15   | 344.69 | 0.01 | 344.70 | 173.70 | 80.90 | 0.24 | 254.84 |
| 7500 | 15| 25   | 339.05 | 0.00 | 339.06 | 173.39 | 134.63 | 0.32 | 308.34 |
| 7500 | 20| 1    | 1012.33 | 0.01 | 1012.34 | 373.37 | 9.47 | 0.51 | 383.35 |
| 7500 | 20| 15   | 739.17 | 0.01 | 739.17 | 397.70 | 112.47 | 0.50 | 510.69 |
| 7500 | 20| 25   | 743.03 | 0.02 | 743.05 | 440.60 | 187.97 | 0.62 | 629.19 |
| 10000| 5 | 1    | 102.15 | 0.03 | 102.18 | 7.73  | 4.05 | 0.07 | 11.85 |
| 10000| 5 | 15   | 103.07 | 0.03 | 103.10 | 11.82 | 59.86 | 0.10 | 71.79 |
| 10000| 5 | 25   | 107.05 | 0.03 | 107.08 | 15.54 | 95.76 | 0.12 | 111.42 |
| 10000| 10| 1    | 266.06 | 0.03 | 266.09 | 50.68 | 7.30 | 0.14 | 58.12 |
| 10000| 10| 15   | 250.03 | 0.03 | 250.06 | 63.56 | 95.08 | 0.26 | 158.90 |
| 10000| 10| 25   | 264.77 | 0.02 | 264.79 | 79.28 | 158.52 | 0.30 | 238.11 |
| 10000| 15| 1    | 582.26 | 0.02 | 582.27 | 189.91 | 11.81 | 0.26 | 201.99 |
| 10000| 15| 15   | 519.76 | 0.02 | 519.77 | 213.69 | 145.75 | 0.44 | 359.88 |
| 10000| 15| 25   | 532.84 | 0.03 | 532.87 | 244.08 | 243.94 | 0.49 | 488.52 |
| 10000| 20| 1    | 1414.70 | 0.02 | 1414.72 | 507.37 | 15.56 | 0.68 | 523.61 |
| 10000| 20| 15   | 1121.15 | 0.02 | 1121.17 | 536.02 | 195.96 | 0.83 | 732.81 |
| 10000| 20| 25   | 1162.93 | 0.02 | 1162.95 | 621.88 | 326.86 | 1.08 | 949.83 |
### Table B.2: Banking Data from Dulá and López

<table>
<thead>
<tr>
<th>#</th>
<th>n</th>
<th>d</th>
<th>p[%]</th>
<th>Simple Algorithm</th>
<th></th>
<th></th>
<th></th>
<th>Dulá-Helgason Algorithm</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LPs</td>
<td>Rest</td>
<td>Total</td>
<td>LPs</td>
<td>IPs</td>
<td>Rest</td>
<td>Total</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4971</td>
<td>7</td>
<td>6.94</td>
<td>43.36</td>
<td>0.01</td>
<td>43.37</td>
<td>19.77</td>
<td>9.17</td>
<td>0.04</td>
<td>28.98</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12456</td>
<td>8</td>
<td>5.74</td>
<td>358.90</td>
<td>0.03</td>
<td>358.93</td>
<td>86.07</td>
<td>52.68</td>
<td>0.14</td>
<td>138.89</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>19939</td>
<td>11</td>
<td>24.88</td>
<td>1693.90</td>
<td>0.06</td>
<td>1693.96</td>
<td>962.63</td>
<td>673.23</td>
<td>0.83</td>
<td>1636.69</td>
<td></td>
</tr>
</tbody>
</table>

### Table B.3: Cyclic Polytopes

<table>
<thead>
<tr>
<th>n</th>
<th>d</th>
<th>p[%]</th>
<th>Simple Algorithm</th>
<th></th>
<th></th>
<th></th>
<th>Dulá-Helgason Algorithm</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>LPs</td>
<td>Rest</td>
<td>Total</td>
<td>LPs</td>
<td>IPs</td>
<td>Rest</td>
<td>Total</td>
<td></td>
</tr>
<tr>
<td>2500</td>
<td>5</td>
<td>4.98</td>
<td>0.00</td>
<td>4.98</td>
<td>3.75</td>
<td>20.53</td>
<td>0.02</td>
<td>24.30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2500</td>
<td>10</td>
<td>12.76</td>
<td>0.01</td>
<td>12.77</td>
<td>10.09</td>
<td>35.15</td>
<td>0.03</td>
<td>45.27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2500</td>
<td>15</td>
<td>101.89</td>
<td>0.00</td>
<td>101.89</td>
<td>60.87</td>
<td>202.92</td>
<td>0.27</td>
<td>263.26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2500</td>
<td>20</td>
<td>386.42</td>
<td>0.01</td>
<td>386.42</td>
<td>263.08</td>
<td>436.47</td>
<td>0.83</td>
<td>523.72</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2500</td>
<td>25</td>
<td>649.94</td>
<td>0.01</td>
<td>649.94</td>
<td>549.49</td>
<td>488.42</td>
<td>0.98</td>
<td>993.89</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table B.3: Cyclic Polytopes (continued)

<table>
<thead>
<tr>
<th>n</th>
<th>d</th>
<th>Simple Algorithm</th>
<th>Dulá-Helgason Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>LPs</td>
<td>Rest</td>
</tr>
<tr>
<td>7500</td>
<td>5</td>
<td>58.70</td>
<td>0.02</td>
</tr>
<tr>
<td>7500</td>
<td>10</td>
<td>137.85</td>
<td>0.02</td>
</tr>
<tr>
<td>7500</td>
<td>15</td>
<td>236.03</td>
<td>0.01</td>
</tr>
<tr>
<td>7500</td>
<td>20</td>
<td>394.23</td>
<td>0.02</td>
</tr>
<tr>
<td>7500</td>
<td>25</td>
<td>621.42</td>
<td>0.01</td>
</tr>
<tr>
<td>10000</td>
<td>5</td>
<td>118.84</td>
<td>0.02</td>
</tr>
<tr>
<td>10000</td>
<td>10</td>
<td>246.76</td>
<td>0.03</td>
</tr>
<tr>
<td>10000</td>
<td>15</td>
<td>404.48</td>
<td>0.02</td>
</tr>
<tr>
<td>10000</td>
<td>20</td>
<td>670.47</td>
<td>0.03</td>
</tr>
<tr>
<td>10000</td>
<td>25</td>
<td>1040.56</td>
<td>0.04</td>
</tr>
<tr>
<td>10000</td>
<td>30</td>
<td>1483.82</td>
<td>0.02</td>
</tr>
<tr>
<td>10000</td>
<td>35</td>
<td>1991.80</td>
<td>0.02</td>
</tr>
<tr>
<td>10000</td>
<td>40</td>
<td>2546.52</td>
<td>0.02</td>
</tr>
<tr>
<td>12500</td>
<td>20</td>
<td>985.88</td>
<td>0.04</td>
</tr>
<tr>
<td>15000</td>
<td>20</td>
<td>1399.83</td>
<td>0.04</td>
</tr>
<tr>
<td>17500</td>
<td>20</td>
<td>1897.45</td>
<td>0.06</td>
</tr>
<tr>
<td>20000</td>
<td>20</td>
<td>2428.18</td>
<td>0.06</td>
</tr>
<tr>
<td>22500</td>
<td>20</td>
<td>3023.31</td>
<td>0.07</td>
</tr>
<tr>
<td>25000</td>
<td>20</td>
<td>3655.58</td>
<td>0.08</td>
</tr>
<tr>
<td>27500</td>
<td>20</td>
<td>4417.09</td>
<td>0.07</td>
</tr>
<tr>
<td>30000</td>
<td>20</td>
<td>5179.43</td>
<td>0.11</td>
</tr>
</tbody>
</table>
Bibliography


