Computation on Polyhedral Arrangements and its Applications on Lifting Regions

By: Peijun Xiao

Faculty advisor: Matthias Köppe

Senior thesis

June 2017

UNIVERSITY OF CALIFORNIA, DAVIS

COLLEGE OF LETTERS AND SCIENCE

DEPARTMENT OF MATHEMATICS

Contents

Abstract

In this thesis, we introduce the definition of polyhedral arrangements and develop a computational method to arrange the polyhedra. Furthermore, we explore possible ways to define a polyhedral dissection of a polyhedral arrangement, and we provide the definitions of polyhedral complexes and polyhedral triangulations. We review regular triangulations of point configurations. We show that there exist a hierarchy of polyhedral arrangements. Every polyhedral complex is a polyhedral dissection; every polyhedral triangulation is a polyhedral complex. At the end of the thesis, we apply polyhedral arrangements to the study of lifting regions, a topic in the theory of cut-generating functions. We provide Sage code to compute the volumns of lifting regions that are assigned to a torus.

Acknowledgments

I would like to thank my thesis advisor, Matthias Köppe, for giving me this research opportunity and for all of his encouragement, advice, and support throughout this project. I would also like to thank Yuan Zhou, a graduating Ph.D. student in the Department of Mathematics at UC Davis, for her generous support and help throughout the project. I would like to thank Shuidie Yao for her encouragement and support throughout the research.

CHAPTER 1

A Hierarchy of Polyhedral Arrangements

1.1. Polyhedral Arrangements

DEFINITION 1.1.1. *A **polyhedral arrangement** A is a collection of polyhedra in $\mathbb{R}^n$.*

The collection of polyhedra for a polyhedral arrangement does not need to be a finite set. We call a polyhedron p of a polyhedral arrangement A an **element** of the polyhedral arrangement. A more well-studied arrangement in literature is the hyperplane arrangement. A hyperplane arrangement is defined as a collection of affine hyperplanes in a vector space $V \cong K^n$ where K is a field [**Sta04**]. Similarly, the collection of affine hyperplanes for a hyperplane arrangement can possibly be infinite [**Par12**]. In optimization, hyperplane arrangements are applied in algorithm design for the least distance problem [**FP09**]. In this thesis, polyhedral arrangements are used as a geometrical tool to study lifting regions, a topic in the theory of cut-generating functions. This topic will be introduced in Chapter 4.

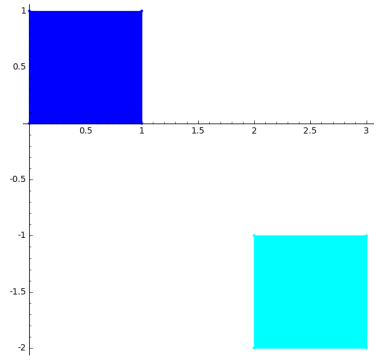
1.2. Polyhedral Dissections

In the study of geometry, one is interested in decomposing a polyhedron efficiently. For example, given a d-dimensional cube, a **dissection** of the cube is defined as a decomposition of the cube into d-dimensional simplices whose interiors are pairwise disjoint but that do not necessarily intersect in a common face [**Sen13**]. Inspired by the dissection of a d-dimensional cube, we are interested in developing a definition of the dissection of more than one polyhedron.

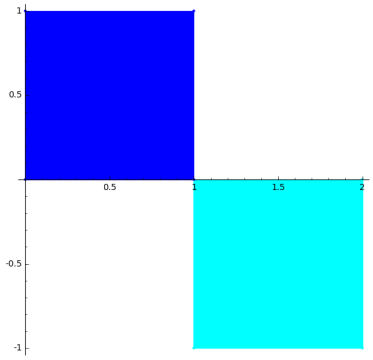
First, we need to clarify some terms. Given some geometrical objects X and Y:

(i) a dissection **consisting of** X and Y means that X and Y are elements of the dissection.

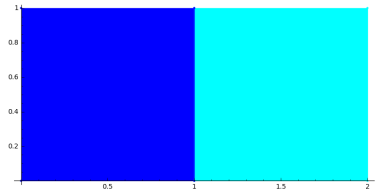
(ii) a dissection **of** X means that the union of the elements of the dissection is X.



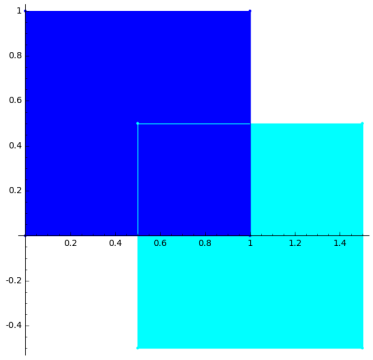
(a) Intersection has dimension -1



(b) Intersection has dimension 0



(c) Intersection has dimension 1



(d) Intersection has dimension 2

FIGURE 1.1. Cases that two square have different intersections

For example, the dissection **of** a d-dimensional cube **consists of** some d-dimensional simplices. In this chapter, we are interested in defining when a polyhedral arrangement is called a polyhedral dissection **of** itself.

EXAMPLE 1.2.1. *The easiest example to start with is a polyhedral arrangement consisting of two squares p and q. There are four cases to discuss:*

(1) Square p and square q does not intersect. The polyhedral arrangement in Figure 1.1a is an example of this case.

(2) Two squares intersect, and their intersection has dimension 0, i.e., a point. The polyhedral arrangement in Figure 1.1b is an example of this case.

(3) Two squares intersect, and their intersection has dimension 1, i.e., a line segment. The polyhedral arrangement in Figure 1.1c is an example of this case.

(4) Two squares intersect, and their intersection has dimension 2. The polyhedral arrangement in Figure 1.1d is an example of this case.

For Case (1), obviously, the polyhedral arrangement is a dissection of itself, since the squares do not intersect with each other. The polyhedral arrangements are dissections of themselves in both Case (2) and Case (3), because those cases occur only if the squares intersect at their boundaries. The squares are still "complete" if we "tear" the squares apart from their intersection. However, we can't "tear" the squares apart from each other in Case (4). Therefore, the polyhedral arrangement is not a polyhedral dissection of itself in Case (4).

By observation of the dimensions of the intersections, assume all the elements from a polyhedral arrangement have the **same** dimension, we derive a criterion to check if the polyhedral arrangement is a polyhedral dissection of itself: if any two different elements p and q from the polyhedral arrangement satisfies

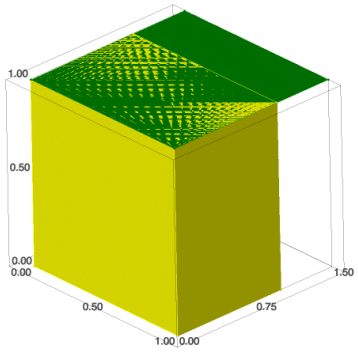
$$\dim(p \cap q) < \min(\dim(p))$$

If there exists two elements p and q from the polyhedral arrangement that have **different** dimensions, then the following condition is one possible way to determine if a polyhedral arrangement A is a polyhedral dissection of itself: every pair of different elements p, q of A satisfies

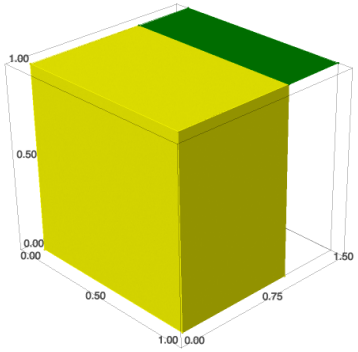
$$\dim(p \cap q) < \min(\dim(p), \dim(q))$$

We refer this condition as the **dimension condition**.

EXAMPLE 1.2.2. *The elements from each polyhedral arrangement in Figure 1.2a and Figure 1.2b have different dimensions. The polyhedral arrangement in Figure 1.2a is constructed with the unit cube colored in yellow and a wide rectangle with vertices* $(0,0,1), (0,3/2,1), (1,0,1), (1,3/2,1)$ *colored in green. The intersection of the wide rectangle and the cube is the upper face (has dimension 2) of the cube. The relative interior of the upper face is in the relative interior of the wide rectangle. These two objects are "glued" together, hence, the polyhedral arrangement is not a polyhedral dissection of itself.*



(a) A cube with a wide rectangle



(b) A cube with a thin rectangle

FIGURE 1.2. Elements have different dimensions

Consider another polyhedral arrangement in Figure 1.2b which is constructed with the unit cube colored in yellow and a thin rectangle with vertices $(0,1,1),(0,3/2,1),(1,1,1),(1,3/2,1)$ *colored in green. Since the intersection of the cube and the thin rectangle is the edge which has dimension 1 less than 2, therefore, the polyhedral arrangement is a polyhedral dissection of itself.*

Rather than using the **dimension condition**, we found that using the notion of the intersection of the relative interiors of the polyhedra is a better way to define a polyhedral dissection.

LEMMA 1.2.1. *Given two polyhedra* p *and* q*, denote the relative interior of a polyhedron* p *as* $\text{relint}(p)$*, then (i) implies (ii):*

(i) $\text{relint}(p) \cap \text{relint}(q) = \emptyset$*;*

(ii) $\dim(p \cap q) < \min(\dim(p), \dim(q))$.

By Lemma 1.2.1, we can replace our previous **dimension condition** by the new **relative interior condition**: a polyhedral arrangement A is a polyhedral dissection of itself if any two different elements p, q of A satisfies $\text{relint}(p) \cap \text{relint}(q) = \emptyset$.

If we use the **relative interior condition**, we can take care of situations that the **dimension condition** can't take care of.

EXAMPLE 1.2.3. *Consider the polyhedral arrangement in Figure 1.3. The polyhedral arrangement is constructed by a line segment with end points* $(1,1/2),(2,1/2)$ *and another line segment with end*

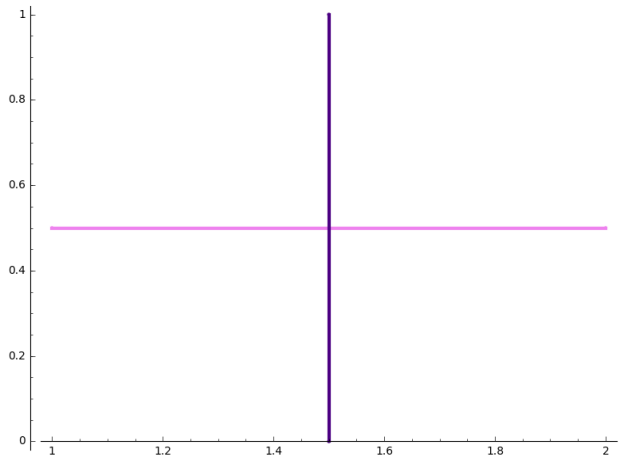


FIGURE 1.3. This is not a polyhedral dissection consisting of two line segments

points $(3/2, 0), (3/2, 1)$. *Both of the line segments have dimension 1, and their intersection is a point at* $(2/3, 1/2)$ *which has dimension 0. This implies that the polyhedral arrangement satisfies the* ***dimension condition****. However, this polyhedral arrangement violates the* ***relative interior condition****, because the intersection of the lines is a point rather than an empty set. The* ***relative interior condition*** *determines that this polyhedral arrangement is not a polyhedral dissection.*

In general, if there exists some elements of a polyhedral arrangement A that have different dimensions, the we define a polyhedral dissection of A as the following:

DEFINITION 1.2.1. *A polyhedral arrangement* A *in* $\mathbb{R}^n$ *is called a* ***polyhedral dissection*** *of itself if every pair of different elements* p, q *of the polyhedral arrangement satisfying*

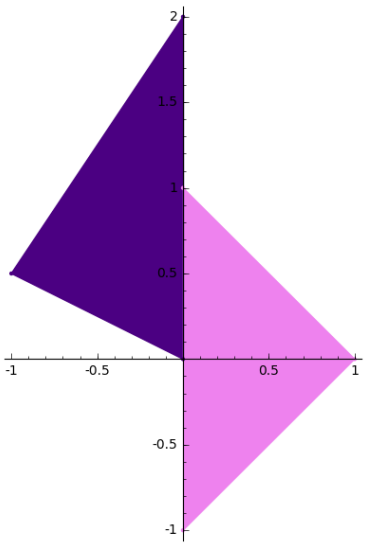
$$\text{relint}(p) \cap \text{relint}(q) = \emptyset$$

1.3. Polyhedral Complexes

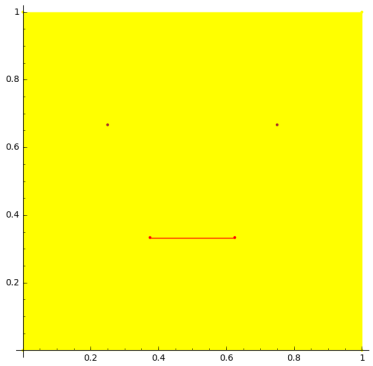
Recall that a polytope is in the form $p = \{x \in \mathbb{R}^d : Ax \leq \boldsymbol{b}\}$ where $A \in \mathbb{R}^{n \times d}$ and a vector $\boldsymbol{b} \in \mathbb{R}^n$.

DEFINITION 1.3.1. *[**AZ14**] A face of a polytope* p *is subset* $f \subseteq p$ *of the form*

$$f = p \cap \{x \in \mathbb{R}^d : a^T x = b\}$$



(a) A polyhedral dissection of two triangles



(b) A polyhedral arrangement of a square, two points and a line inside the square is not a polyhedral dissection

FIGURE 1.4. Example and non-example of polyhedral dissections

where $a^T x \leq b$ is a linear inequality which is valid for all points $x \in p$.

LEMMA 1.3.1. *Every face of a polytope is itself a polytope.*

PROOF. Given a polytope $p = \{x \in \mathbb{R}^d : Ax \leq \boldsymbol{b}\}$ where $A \in \mathbb{R}^{n \times d}$ and a vector $\boldsymbol{b} \in \mathbb{R}^n$, and a face $f = p \cap \{x \in \mathbb{R}^d : a^T x = b\}$. Then construct $A' \in R^{(n+2) \times d}$ with the first n rows of A' are the rows of A, and the last two rows of A' are the vectors a and $-a$. Also construct $\boldsymbol{b'} \in \mathbb{R}^{n+2}$ with the first n entries are the entries of $\boldsymbol{b}$, and the last two entries are b and $-b$. Then the face f is also a polytope in the form $f = \{x \in \mathbb{R}^d : A'x \leq \boldsymbol{b'}\}$. □

We can use the same technique in the proof of Lemma 1.3.1 to prove that every face of a polyhedron is itself a polyhedron.

Recall a finite **abstract simplicial complex** is a finite set A together with a collection Δ of subsets of A such that if $X \in \Delta$ and $Y \subseteq X$, then $Y \in \Delta$. Every element $\delta \in \Delta$ is called a **simplex** of Δ. A simplex of an abstract simplicial complex is a maximal simplex if it is maximal with respect to set inclusion [**Koz08**].

A geometric version of a simplex is **geometric n-simplex**, which is defined as a convex hull of a set A of $n+1$ affine independent points in $\mathbb{R}^N$, for some $N \geq n$ [**Koz08**]. The convex hulls of the subsets of A are called **subsimplices** of σ [**Koz08**]. Notice that simplices are the simplest of polyhedra which are points, line segments, triangles, tetrahedra, etc. [**DRS10**]. The geometric version of an abstract simplicial complex is called a **geometric simplicial complex**. A geometric simplicial complex K in $\mathbb{R}^N$ is a collection of simplices in $\mathbb{R}^N$ such that every subsimplex of a simplex of K is a simplex of K and the intersection of any two simplices of K is a subsimplex of each of them [**DRS10**].

Since we are studying geometrical objects in this thesis, from now on, all simplicial complexes mean geometric simplicial complexes, and all simplices mean geometric n-simplexes.

We change "subsimplex" to "face" and "simplex" to "polyhedron" in the definition of geometric n-simplex to derive the definition of a polyhedral complex as the following:

DEFINITION 1.3.2. *[**BHK16**] A **polyhedral complex** C is a polyhedral arrangement in $\mathbb{R}^n$ such that*

(i) every face of a polyhedron in C is itself a polyhedron in C;

(ii) the intersection of any two polyhedra p and q in C is a face of p and of q;

A polyhedron of C is called a **face** of the polyhedral complex [**BHK16**]. Notice that by Lemma 1.3.1, every face of a polyhedron is a polyhedron, and property (i) of Definition 1.3.2 states that every face of a polyhedron in a polyhedral complex is a polyhedron in the complex. Therefore, our definition of a face of a polyhedral complex does not conflict with the definition of a face of a polyhedron. Property (ii) of Definition 1.3.2 is called **face-to-face** [**BHK16**].

Every polyhedron has an empty face. A face of a polyhedral complex C is called a vertex if it has dimension 0. A face of C of dimension i is an i-face of C [**BHK16**]. A face of polyhedral complex is called **maximal** if it is maximal with respect to set inclusion. With this definition of maximal faces, we define a polyhedral complex C to be **pure** if all maximal faces have the same dimension. The polyhedral complexes in Figure 1.7 are pure. The polyhedral complex in Figure 3.2e is not pure.

1.4. Polyhedral Triangulations

Triangulations is an important topic in computational geometry and have many applications in algebra, computer science, combinatorics and optimization [**DRS10**]. To study triangulations, besides the concepts of simplicial complexes introduced in section 1.3, we also need to study point configurations. Recall a **point configuration** is a finite collection of points $A = \{a_1, a_2, \cdots, a_d\}$ in $\mathbb{R}^n$ with non-repeated labels [**DRS10**].

DEFINITION 1.4.1. *[**DRS10**] A triangulation of a point configuration A in $\mathbb{R}^n$ is a collection of n-simplices where the vertices of the simplices are points in A that satisfies the following properties:*

(1) (Union Property) The union of all these simplices equals $\text{conv}(A)$.

(2) (Intersection Property) Any pair of these simplices intersects in a common face.

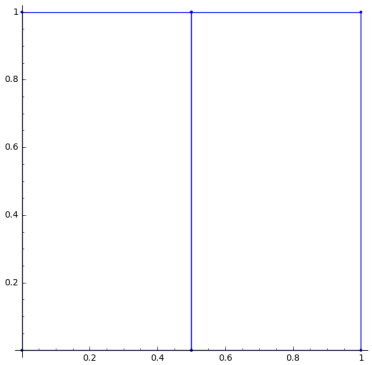
Notice that this definition of triangulation is of a point configuration, rather than of a polyhedral arrangement. Our goal is to develop a definition of a polyhedral triangulation of a polyhedral arrangement.

By observation, the n-simplices in Definition 1.4 are analogs of maximal faces of a polyhedral complex. The intersection property in Definition 1.4 is an analog of the face-to-face property of a polyhedral complex. Inspired by those analogs, we derive the definition of polyhedral triangulation of a polyhedral arrangement as the following:

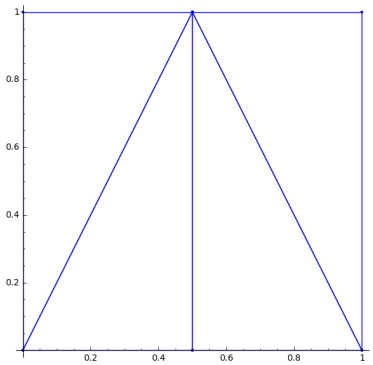
DEFINITION 1.4.2. *A **polyhedral triangulation** T of a polyhedral arrangement A in $\mathbb{R}^n$ is a polyhedral complex C of A such that every maximal face of C is a simplex.*

Since the faces of a simplex are still simplices, then every face of a polyhedral triangulation is simplex. By observation, the collection of the simplices from a polyhedral triangulation is a simplicial complex.

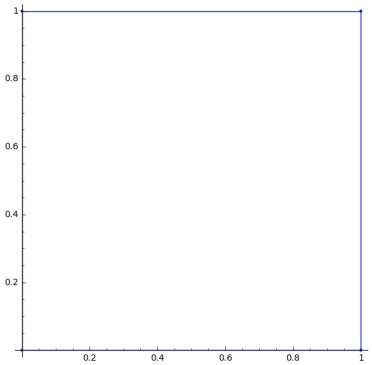
Notice that a polyhedral triangulation of a polyhedral arrangement is not the same as a polyhedral triangulation of the unions of polyhedra of the same polyhedral arrangement.



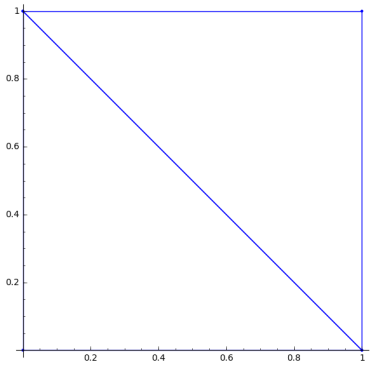
(a) A polyhedral arrangement A of two thin rectangles



(b) A polyhedral triangulation of A in Figure 1.5a



(c) The union of elements of A in Figure 1.5a



(d) A polyhedral triangulation of the square in Figure 1.5c

FIGURE 1.5. Examples of a polyhedral triangulation of a polyhedral arrangement and a polyhedral triangulation of the union of elements for the same polyhedral arrangement

EXAMPLE 1.4.1. *Figure 1.5a gives a polyhedral arrangement with two thin rectangles as elements. Figure 1.5c is a square. It is the union of the elements of the same polyhedral arrangement. Figure 1.5b and Figure 1.5d show that a polyhedral triangulation of a polyhedral arrangement is different from a polyhedral triangulation of the union of the elements of the same polyhedral arrangement.*

1.5. A Hierarchy of Polyhedral Arrangements

LEMMA 1.5.1. *Every polyhedral complex is a polyhedral dissection.*

PROOF. Given a polyhedral complex C. Consider condition (ii) in the definition of a polyhedral complex: the intersection of any two polyhedra p and q in C is either empty or is a face of p and

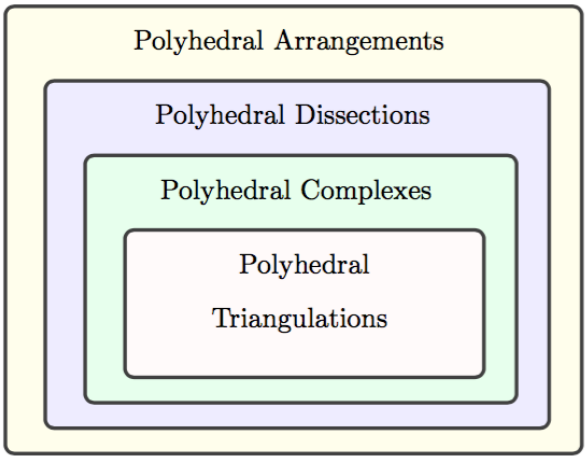


FIGURE 1.6. A hierarchy of polyhedral arrangements

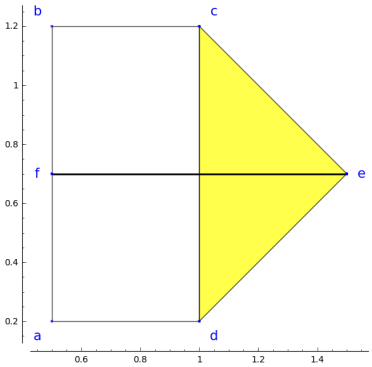
of q. To show C is a polyhedral dissection, we want to show that if p and q are different polyhedra in C, then $\text{relint}(p) \cap \text{relint}(q) = \emptyset$.

Suppose p and q are different polyhedra in C. If the intersection of them is empty, then obviously $\text{relint}(p)$ does not intersect with $\text{relint}(q)$. If the intersection i is a face of p and of q, since p and q are different, then we know the intersection i intersects with p and q at the corresponding supporting hyperplanes of p and of q by the definition of a face of a polytope in Definition 1.3.1. Since the supporting hyperplanes are not in the relative interior of p (of q), then $\text{relint}(p) \cap \text{relint}(q) = \emptyset$. □

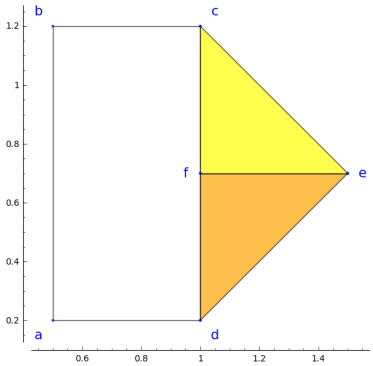
By our definition of polyhedral dissections, polyhedral complexes, and polyhedral triangulations with Lemma 1.5, we conclude that there is a hierarchy of polyhedral arrangements as Figure 1.6 shows.

EXAMPLE 1.5.1. *Figure 1.7a is a polyhedral arrangement with three elements: a white rectangle, a yellow triangle, and a thick black line segment that crosses through the triangle and the rectangle. Because of the line segment, the polyhedral arrangement is not a polyhedral dissection.*

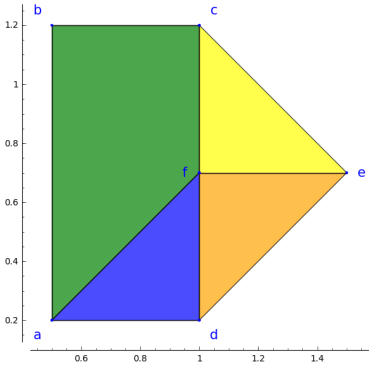
The polyhedral arrangement in Figure 1.7b has three elements: a white rectangle, a yellow triangle and an orange triangle. This polyhedral arrangement is a polyhedral dissection, but not a polyhedral complex, because the rectangle intersects with the yellow triangle at the line segment with endpoints



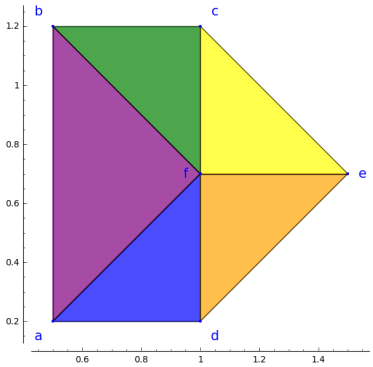
(a) a polyhedral arrangement



(b) a polyhedral dissection, but not a polyhedral complex



(c) a polyhedral complex, but not a polyhedral triangulation



(d) a polyhedral triangulation

FIGURE 1.7. Examples of polyhedral arrangements, polyhedral dissections, polyhedral complexes and polyhedral triangulations

c and f, but the line segment is not a face of the rectangle. Therefore, this polyhedral dissection violates the face-to-face condition to be a polyhedral complex.

The polyhedral arrangement in Figure 1.7c consists of three triangles, a trapezoid and their faces. This polyhedral arrangement is a polyhedral complex, because it satisfies the face-to-face condition. However, it is not a polyhedral triangulation, since the trapezoid is not a simplex.

The polyhedral arrangement in Figure 1.7d consists of five triangles and their faces. This polyhedral arrangement is a polyhedral triangulation, because the polyhedral arrangement is a polyhedral complex, and all elements in the polyhedral complex are simplices.

CHAPTER 2

Computation on Polyhedral Arrangements

2.1. Bucket Method

In the study of polyhedral arrangements, we have to do lots of calculations on the elements, such as checking if two elements from a polyhedral arrangement intersect or not or finding all the polyhedra from the polyhedral arrangement that contain a given point.

To optimize the speed of those calculations, we need to find a proper data structure to efficiently store and retrieve the information of the elements of polyhedral arrangements. Recall that in the study of data structures, **bucket sort** is an efficient method to distribute the elements of a 1-D array into numbers of buckets. Once the data structure is initialized, the time to retrieve the information of an element of the array from the bucket is $\mathcal{O}(1)$.

Since a polytope can be represented as a convex hull of some points in $\mathbb{R}^n$, then the buckets in our data structure have to be hypercubes in $\mathbb{R}^n$ to store a polytope by its represented points. By convention, we refer to those hypercubes as buckets, and we refer to the hypercube that is composed of the buckets as a grid.

Given a polyhedral arrangement, we set up a grid based on the sizes of the elements and the number of the elements. We then specify the width of the buckets and partition the grid into buckets. Subsection 2.2.1 provides more details on how a grid is set up. After we set up the grid, we assign coordinates to label the buckets of the grid.

EXAMPLE 2.1.1. *A real life example of a grid partitioned by high dimensional buckets is the Rubik's cube in Figure 2.1. The Rubik's cube is a grid composed of 27 buckets. We denote the lower-left red bucket as* $(0, 0, 0)$ *and the upper-right blue-yellow bucket as* $(2, 2, 2)$.

FIGURE 2.1. The Rubik's cube are partitioned into 27 3-D buckets.

Once we set up the grid, the next step to initialize the data structure is to find a fast way to distribute the information of the elements of the polyhedral arrangement into the buckets. Instead of using searching methods that are based on comparisons between the information of the elements, a more efficient way to distribute the information of elements is based on **digital tree search** [**Knu98**]. Given a table of records, digital tree search forms a binary tree. While the argument is not in the table, a new node that containing the argument is inserted into the tree in the appropriate place [**Knu98**]. The advantages of digital tree search are saving memory space at expense running time and easy to process the search.

Based on **digital tree search**, we introduce the so-called **bucket method** to initialize the data structure in storing information of elements from a polyhedral arrangement. The bucket method is illustrated in algorithm 1. We proceed the bucket method on every element of a polyhedral arrangement and make hash tables to store the information of the elements. Once we finish running the bucket method on every element of the polyhedral arrangement, we say that the data structure is initialized.

2.2. Sage Code

2.2.1. Set up a grid.

Based on the dimension of the space, the class called 'Grid' constructs a square, a cube, or a hypercube composed of buckets that contain all the elements of the polyhedral arrangement. The class 'Grid' is used for bucket method in the class 'PolyhedralArrangement'. Notice that a grid in the class 'Grid' does not need to be a standard hypercube $[0,1]^d$ for some dimension d. The grid can be a hypercube in $[a,b]^d$ for some $a, b \in \mathbb{R}$.

```
   input  : A polytope p, selection (default as None)
 1 if selection has not been assigned then
 2 |  selection ← all buckets of the grid;
 3 end
 4 lower_left ← the lower-left bucket of the selection;
 5 upper_right ← the upper-right bucket of the selection;
 6 if selection intersects with p then
 7 |  if lower_left == upper_right then
 8 |  |  add lower_left as a key to bucket_contents and p as the value;
 9 |  |  add p as a key to polyhedral_buckets and lower_left as the value;
10 |  else
11 |  |  choose the longest side of selection;
12 |  |  cut selection into two hyperplanes through the longest side;
13 |  |  obtain two new selections, selection_A and selection_B;
14 |  |  call Bucket_Method (selection_A);
15 |  |  call Bucket_Method (selection_B);
16 |  end
17 end
```

Algorithm 1: Bucket_Method(p, Selection)

(1) '__init__()' takes the following five parameters to construct a 'Grid' instance:

   (a) the number of the polyhedra n;

   (b) the ambient dimension d of the polyhedra;

   (c) the coordinates of the lower-left point of the grid;

   (d) the width w of each bucket;

   (e) the side length s of the grid.

(2) 'cut_selection' is a method for bucket method. The inputs of the method are lower-left bucket and the upper-right bucket of a selection. The method finds the longest side of the selection, and cuts the selection through the longest side into two new selections. If the sides have equal lengths, then this method cuts the selection through the side on the first dimension. For example, choose the entire Rubik's cube in Figure 2.1 as a selection. The lower-left bucket is $(0, 0, 0)$, and the upper-right bucket is $(2, 2, 2)$. We pass this selection to 'cut_selection', and the method returns a selection with lower-left bucket $(0, 0, 0)$ and

upper-right bucket $(0, 2, 2)$, and another bucket with lower-left bucket $(1, 0, 0)$ and upper-right bucket $(2, 2, 2)$.

2.2.2. Construct a polyhedral arrangement.

The class called 'PolyhedralArrangement' constructs a polyhedral arrangement from a list, tuple, or a set of polyhedra. In the following paragraphs of this subsection, "elements" means the elements of the polyhedral arrangement. All the elements are required to have the same ambient dimension.

Construction:

(1) '__init__()' constructs a polyhedral arrangement of its elements and sets up a 'Grid' instance, grid.

By default, the grid has the same ambient dimension as the elements of the polyhedral arrangement. Denote the ambient dimension as d. The lower-left coordinates of the grid is the coordinates of the lower-left point among the union of vertices of each element. For each element of the polyhedral arrangement, we calculate the maximum and the minimum of each coordinate, and find the offset of the maximum and the minimum. The largest offset we have found is set to be the side length of the grid. Finally, the width of each bucket in the grid is a rational number approximated from $\frac{s}{\sqrt[d]{n}}$. Experiments are needed to show that this number of buckets will give the best performance time for the bucket method.

(2) 'collection_dict()' returns a dictionary whose keys are the dimension of the elements and whose values are the elements that have that dimension.

(3) 'grid_contents()' calls the bucket method method and returns a dictionary whose keys are buckets and whose values are a set of elements that intersect with the buckets.

(4) 'polyhedron_buckets()' calls the bucket method and returns a dictionary whose keys are the elements and whose values are a set of buckets that intersect with the elements.

(5) 'in_buckets(p)' returns a set of buckets that the element p intersects with.

(6) 'update_dictionaries_for_a_polyhedron(p, selection)' is the bucket method on an element p. In the step of checking if a selection intersects with the element p, 'update_dictionaries_for_a_polyhedron(p, selection)' calls '_set_up_linear_programming_for_intersection (p, selection)' to set up an linear programming problem (LP) in Sage. If the LP has no feasible solutions, then the selection and p do not intersect.

Once the LP is set up, every time the new selection is passed into the search, 'update_dictionaries_for_a_polyhedron(p, selection)' calls '_update_variables_bounds()' to use the lower-left bucket and the upper-right bucket of the new selection to update variables bounds of the existing LP. This warm-starting step of the LP is crucial, because it significantly reduces the running time for the bucket method method.

(7) 'update_dictionaries()' runs the bucket method for each element of the polyhedral arrangement and updates the two dictionaries 'polyhedron_buckets' and 'grid_contents'.

Computation:

(1) '__contains__(p)' checks if a polyhedron p is an element of the polyhedron arrangement.

(2) 'intersect(p, q)' checks if two elements p and q intersects with each other.

(3) 'is_polyhedron_in_polyhedron(p, q)' checks if an element p is in another element q of the polyhedral arrangement. An element p is said to be **in** another element q if all vertices of p are contained in q. Notice that p does not need to be in the polyhedral arrangement.

(4) 'point_in_polyhedra(pt)' returns a list of polyhedra from the elements of the polyhedral arrangement that contain the given point pt.

(5) 'polyhedron_contains_point(p, pt)' checks if an element of the polyhedral arrangement p contains a given point pt.

2.3. Experiments

2.3.1. Check Intersection.

To test the efficiency of bucket method method, we perform experiments in checking if two polyhedra intersect. First we randomly generate 100 polyhedra with a fixed ambient dimension d. Then we randomly choose 100 pairs of polyhedra from the generated polyhedra. We check if any pairs of polyhedra p and q intersect with each other or not by the following three methods:

(1) Use Sage function 'p.intersection(q)' to compute the intersection and check if the intersection is an empty polytope. Denote this method as **Sage_intersection**;

(2) Use the inequalities and equations of the representations of p and q to construct a linear programming problem with objective function 0, and check if the linear programming is non-feasible. Denote this method as **lp_intersection**;

(3) Construct a 'PolyhedralArrangement' instance with the 100 pairs of polyhedra as the elements of the polyhedral arrangement. After running the bucket method method, obtain two sets of buckets that contain p and q respectively. If the sets of buckets don't intersect, then p and q do not intersect. Otherwise, use the linear programming method stated before to check if p and q intersect. In this method, we assume that the data structure has already been initialized; i.e., we do not count the time for constructing the polyhedral arrangement and running the bucket method method in analyzing the performance for this method. We denote this method as **PA_intersection**. PA stands for polyhedral arrangement.

TABLE 2.1. Intersections of 100 pairs of polyhedra

100 pairs	**dimension**	1	**dimension**	2	**dimension**	3	**dimension**	4
	buckets	83	**buckets**	100	**buckets**	125	**buckets**	256
methods	**CPU time**		**CPU time**		**CPU time**		**CPU time**	
	mean	**variance**	**mean**	**variance**	**mean**	**variance**	**mean**	**variance**
Sage_intersection	0.0340	0.0000	0.0767	0.0001	0.1847	0.0009	1.3504	0.0179
lp_intersect	0.0345	0.0000	0.0662	0.0001	0.1359	0.0008	0.4468	0.0007
PA_intersect	0.1650	0.0002	0.1562	0.0008	0.1969	0.0002	0.2859	0.0002
construct PA	2.7576	0.8699	2.0534	0.3591	2.7476	0.3748	2.0434	0.0213

We also want to know the time it takes to initializing the data structure. At the first columns in Table 2.1, the row "construct PA" stands for constructing an 'PolyhedralArrangement' instance and initializing the data structure. For the following columns, we have the results for four experiments on four sets of polyhedra with ambient dimensions varied from 1 to 4.

In the result of each experiments, we include the number of buckets used in constructing the polyhedral arrangements by setting the width of each bucket as $\sqrt[d]{n}$. n is the number of elements in the polyhedral arrangement, and d is the ambient dimension of the polyhedra.

By observation, **Sage_intersection** is the slowest method. For ambient dimension 1 to 3, **lp_intersection** performs better than **PA_intersection**. **PA_intersection** only needs roughly half of the time than **lp_intersection** to check intersection when the ambient dimension is 4. However, we observe that it takes significant time in initializing the data structure.

Next, we raise the testing number of polyhedra from 100 to 200, and then from 200 to 300. We also raise the number of pairs of polyhedra to tested from 100 to 200, and then from 200 to 300. Since **Sage_intersection** has been observed as the slowest method, we do not include this method in our experiments.

TABLE 2.2. Intersections of 200 pairs of polyhedra

200 pairs	**dimension** 1		**dimension** 2		**dimension** 3		**dimension** 4		**dimension** 5	
	buckets 140		**buckets** 187		**buckets** 216		**buckets** 256		**buckets** 243	
methods	**CPU time**		**CPU time**		**CPU time**		**CPU time**		**CPU time**	
	mean	**variance**	**mean**	**variance**	**mean**	**variance**	**mean**	**variance**	**mean**	**variance**
lp_intersect	0.0626	0.0000	0.1488	0.0012	0.2651	0.0007	0.9595	0.0194	7.1631	1.0527
PA_intersect	0.5940	0.0010	0.5044	0.0016	0.5476	0.0008	0.6687	0.0098	0.2078	0.0028
construct PA	15.5813	3.0829	9.3947	0.3217	12.6268	0.3164	4.7964	0.0438	3.5028	0.0222

TABLE 2.3. Intersections of 300 pairs of polyhedra

300 pairs	**dimension** 1		**dimension** 2		**dimension** 3		**dimension** 4		**dimension** 5	
	buckets 198		**buckets** 289		**buckets** 343		**buckets** 625		**buckets** 1025	
methods	**CPU time**		**CPU time**		**CPU time**		**CPU time**		**CPU time**	
	mean	**variance**	**mean**	**variance**	**mean**	**variance**	**mean**	**variance**	**mean**	**variance**
lp_intersect	0.1062	0.0000	0.1912	0.0004	0.4199	0.0002	1.6217	0.0023	10.4348	1.1544
PA_intersect	1.2680	0.0102	1.0153	0.0008	0.9858	0.0017	1.2327	0.1888	0.1641	0.0002
construct PA	46.8352	27.2894	24.5346	2.3484	36.1479	3.0201	14.4418	0.4312	6.7237	0.1002

From Table 2.2 and Table 2.3, we have similar observations: **PA_intersection** performs better than **lp_intersection** when the ambient dimensions is equal or greater than 4.

2.3.2. Find point containment.

Another experiment we have done is to check which polyhedra from a large set of polyhedra contain a given point. Similar to the experiment in 2.3.1, we randomly generate 300 polyhedra in a hypercube $[0, 1]^d$ with a fixed ambient dimension d. Then we randomly choose 300 points in the

hypercube. In the experiment, we compare the CPU-time in running the following two methods on those polyhedra and points:

(1) Use Sage function 'p.contains(pt)' check if a polyhedron p from the collection of polyhedra contains the point pt. Iterate this step over all polyhedra in the collection. Denote this method as **Sage_contain**;

(2) Construct a 'PolyhedralArrangement' instance with the 300 pairs of polyhedra as the elements of the polyhedral arrangement. First we calculate which bucket contains the given point pt. Denote this bucket as bk. After initializing the data structure, we look for which polyhedra intersect with bk by looking up bk's values in the dictionary "grid_contents". Denote this set of polyhedra that intersect with bk as **possible_polyhedra**. Finally, we use the Sage function 'p.contains(pt)' to check if a polyhedron p from **possible_polyhedra** contains the point pt. We denote this method as **PA_contain**. PA stands for polyhedral arrangement.

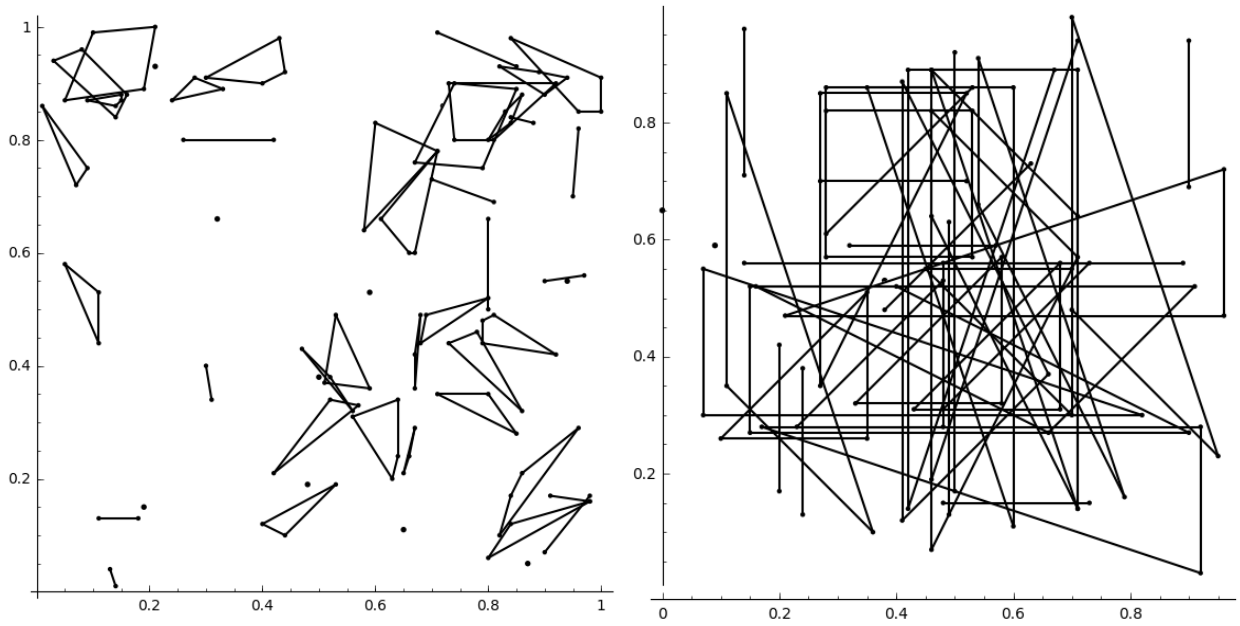
TABLE 2.4. Find containment with **size_range** $\frac{1}{2}$

300 polyhedra	**dimension**	1	**dimension**	2	**dimension**	3	**dimension**	4
size_range = 1 / 2	**buckets**	148	**buckets**	324	**buckets**	343	**buckets**	625
methods	**CPU time**		**CPU time**		**CPU time**		**CPU time**	
	mean	**variance**	**mean**	**variance**	**mean**	**variance**	**mean**	**variance**
Sage_contain	3.2529	0.0086	7.7579	0.1700	7.8048	0.0343	9.5154	1.0115
PA_contain	0.6065	0.0128	0.4847	0.0007	0.6808	0.0308	0.9927	0.0268
construct PA	9.7630	6.2639	16.3038	0.5057	14.0755	0.5256	15.6958	0.9973

TABLE 2.5. Find containment with **size_range** $\frac{1}{10}$

300 polyhedra	**dimension**	1	**dimension**	2	**dimension**	3	**dimension**	4
size_range = 1 / 10	**buckets**	198	**buckets**	289	**buckets**	343	**buckets**	625
methods	**CPU time**		**CPU time**		**CPU time**		**CPU time**	
	mean	**variance**	**mean**	**variance**	**mean**	**variance**	**mean**	**variance**
Sage_contain	4.4914	0.0986	7.1304	0.0153	8.1357	0.1549	9.2963	0.0495
PA_contain	0.2765	0.0019	0.1185	0.0000	0.1181	0.0002	0.1060	0.0000
construct PA	8.7568	2.1597	5.3033	0.4453	5.8926	0.3821	4.4077	0.1187

In this experiment, we introduce a new parameter **size_range**, which restricts the size of the randomly generated polyhedra. Denote **size_range** as s. To randomly generate a polyhedron, we first randomly pick a point in the hypercube as the center of the polyhedron. Denote the center point as $\mathbf{c} = (c_1, c_2, \cdots, c_d)$. Then we pick vertices such that every coordinate v_i of each vertex is randomly chosen from $[c_i - s, c_i + s] \cap [0, 1]$. In other words, the vertices are chosen from a



(a) 50 polyhedra generated with **size_range** $\frac{1}{10}$

(b) 50 polyhedra generated with **size_range** $\frac{1}{2}$

FIGURE 2.2. Examples of how **size_range** affects the size of polyhedra in $[0,1]^2$

hypercube that is centered at c with the side of length $2s$. We then use those vertices to generate a polyhedron. Repeat the process until we get 300 randomly generated polyhedra.

Figure 2.2 gives an example on how **size_range** affects the sizes of 50 polyhedra in $[0,1]^2$. By observation, the smaller the **size_range** is, the more likely that the polyhedra separated from each other in the $[0,1]^d$ hypercube. From Table 2.4 and Table 2.5, we observe that the smaller the **size_range** is, the less time it takes to initialize the data structure.

In general, **Sage_contain** is slower than **PA_contain**. The gaps between the CPU-time for these two methods grow larger as the ambient dimension of the polyhedra increases. As we just discussed, it takes less time to initialize **PA_contain** when **size_range** is small. Therefore, the time we save in using **PA_contain** instead of **Sage_contain** can easily to compensate the time we spend on initializing **PA_contain** when **size_range** is small. This result is not surprising, because our bucket method performs well in arranging polyhedra when the polyhedra are small or have big gaps among each other.

Overall, if a set of polyhedra is randomly generated, then **PA_contain** is a faster method than **Sage_contain** to check which polyhedra from the set contain a given point.

CHAPTER 3

Regular Triangulations

3.1. Regular Triangulations of Point Configurations

Recall that a **point configuration** is a finite collection of points $A = \{a_1, a_2, \cdots, a_n\}$ in $\mathbb{R}^d$ with non-repeated labels [**DRS10**]. [**DRS10**] shows that every point configuration has at least one triangulation. In particular, every point configuration has so-called regular triangulations [**DRS10**].

To construct a regular triangulation, first we pick a **height function** $\omega : A \to \mathbb{R}$ such that every $a_i \in A$ is assigned with a weight ω_i. Then the **lifted point configuration** [**DRS10**] in $\mathbb{R}^{n+1}$ is represented in the form

$$A^\omega := \begin{bmatrix} a_1 & a_2 & \cdots & a_n \\ \omega_1 & \omega_2 & \cdots & \omega_n \end{bmatrix}$$

Literally, a lifted point configuration is a point configuration being lifted to one dimension higher by assigning weights ω to all the points in the point configuration. We denote the lifted point configuration of A as A^ω.

Given a point configuration A, we compute a face structure of the polytope $p = \text{conv}(A^w)$. A **lower face** of a lifted point configuration A^ω is a face of p that is visible from below of p [**DRS10**]. We also define a **regular subdivision** of a point configuration A produced by w to be the set of lower faces of the lifted point configuration A^ω [**DRS10**].

To get a triangulation of a point configuration A, we first lift up the point configuration by weight functions ω, then we project the polytope $p = \text{conv}(A^w)$ down to get its **lower face**. Notice that the collection of projected lower faces satisfies the three properties in Definition 1.4.2 for triangulation of point configuration, if we assign generic weights to the point configuration, and the projections of the lower faces form a triangulation of a point configuration [**DRS10**].

We say a triangulation of a point configuration A in $\mathbb{R}^n$ is called **regular** if it can be obtained by projecting the lower envelope of a lifting of A to $\mathbb{R}^{n+1}$. Notice that a regular triangulation of a point configuration A is a particular case of a regular subdivision of A when the weights are generic).

3.2. Polyhedral Triangulations of Polyhedral Arrangements

Inspired by the idea of regular triangulation of a point configuration, we want to explore if we could use the same ways to construct a polyhedral triangulation of a polyhedral arrangement. First, for each element of the polyhedral arrangement, we take its vertices to form a point configuration, and construct a regular triangulation for the point configuration. We repeat this step for all the elements of the arrangement. Finally, we take the union of the regular triangulations. However, as Figure 3.1 shows, this way of construction does not guarantee a polyhedral triangulation of a polyhedral arrangement.

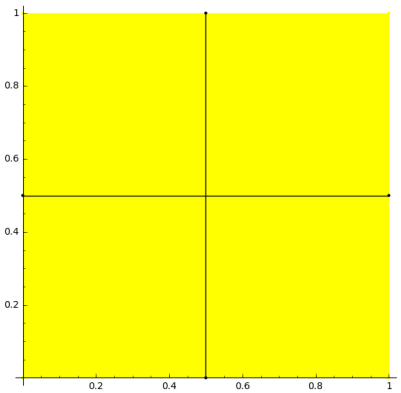
EXAMPLE 3.2.1. *Figure 3.1a gives a polyhedral arrangement of a square and two line segments. Figure 3.1b shows the union of regular triangulations of point configurations of the vertices lists of elements of 3.1a. Figure 3.1b is composed of two triangles and two line segments.*

Figure 3.1b is not a polyhedral triangulation of Figure 3.1a, because the intersection point $(1/2, 1/2)$ *of the two line segments is not included. This violates the* ***face-to-face*** *condition of a polyhedral complex. By Definition 1.4.2, every polyhedral triangulation of a polyhedral arrangement has to be a polyhedral complex of the polyhedral arrangement.*

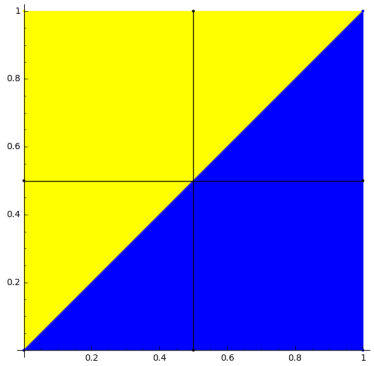
We have not found a way to make a polyhedral arrangement into a polyhedral triangulation. More study is needed on this topic in the future.

3.3. Sage Code and Examples

The code to compute a regular triangulation of a point configuratoin is in the class 'PolyhedralArrangement'.



(a) A polyhedral arrangement of a square and two line segments



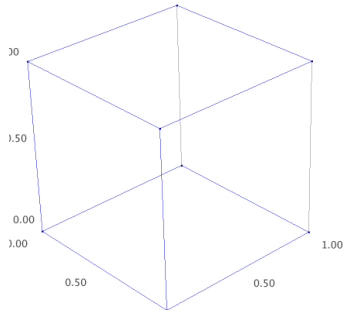
(b) The union of regular triangulations of some point configurations

FIGURE 3.1. Examples of regular triangulations of point configurations do not guarantee a polyhedral triangulation of a polyhedral arrangement. Notice that each point configuration for the regular triangulations in Figure 3.1b is the list of the vertices of an element of the polyhedral arrangement in Figure 3.1a

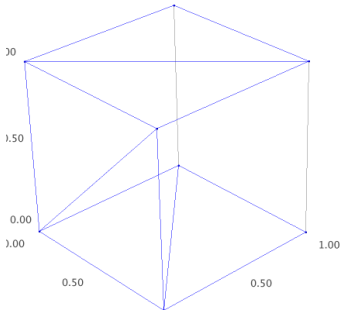
1. 'regular_triangulation(pt_config, dim)' takes two inputs: a point configuration 'pt_config', and the dimension 'dim' of the polyhedron that we want to triangulate. The method first computes the weight of each point in the point configuration plus some perturbation to each weight. Then the method constructs a polyhedron as same as the original polyhedron except that the weight gives an extra dimension to each vertex of the original polyhedron. Finally, the method projects the new polyhedron onto the faces with dimension 'dim'. Those faces are the lower faces of the lifted point configuration. The union of those faces forms a regular triangulation of the point configuration.

2. 'weight(p_1, p_2)' computes the weight of two points p_1 and p_2. The weight is defined as the square of the Euclidean distance between p_1 and p_2.

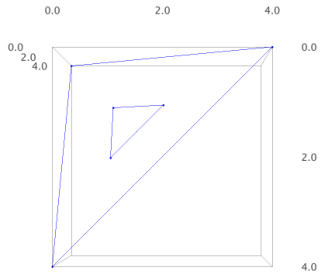
3. '_small_perturbation(list)' first finds the minimal value of the elements in a list. Denote the minimum as m. Then the method constructs a list of random rational numbers in $[m * 10^{-8}, m * 10^{-5}]$ as the perturbation list of the input list. This method is used in 'regular_triangulation' to assign perturbations to the weight function.



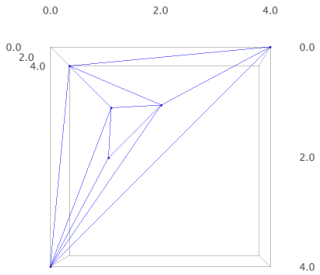
(a) A polyhedral complex composed by three squares from top, bottom and left



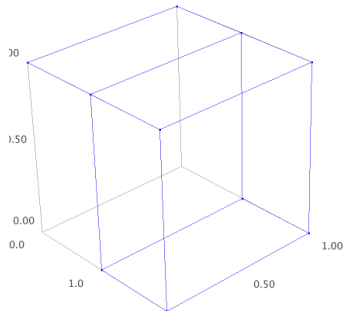
(b) The triangulations of the point configurations of the polyhedra in polyhedral arrangement in Figure 3.2a



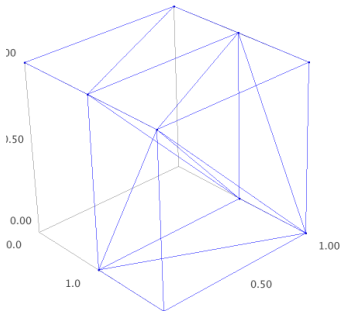
(c) A polyhedral arrangement of embedded triangles in 3D (Example 2.2.5 in [**DRS10**])



(d) The triangulations of the point configurations of the polyhedra in the polyhedral arrangement in Figure 3.2c



(e) A cube and a square forms a non-pure polyhedral complex



(f) The triangulations of the point configurations of the polyhedra of polyhedral complex in Figure 3.2e

FIGURE 3.2. Examples of triangulations of the point configurations of the polyhedra in some polyhedral arrangements

CHAPTER 4

Applications on Volumes of Lifting Regions

4.1. Background

In the study of integer programming, a convex set is called **lattice-free** if there is no integral points in its interior [**BC09**]. A lattice-free convex set is **maximal** if it is inclusion-maximal in the class of lattice-free convex sets [**Wag11**].

EXAMPLE 4.1.1. *The triangle in Figure 4.1a has vertices* $(0,1),(-3,0),(3,0)$. *This triangle is lattice-free, but not maximal lattice-free, because it is contained in the triangle in Figure 4.1b. The triangle in Figure 4.1b is maximal lattice-free, because it is maximal with respect to set inclusion.*

In the theory of cut-generating functions, we are interested in studying the so-called **lifting regions**. To define a lifting region of a maximal lattice-free convex set $B \in \mathbb{R}^n$ corresponding to a $f \in B$, we define $P_F(f,B) = \text{conv}(f \cup F)$ for each facet F of B [**AB14**].

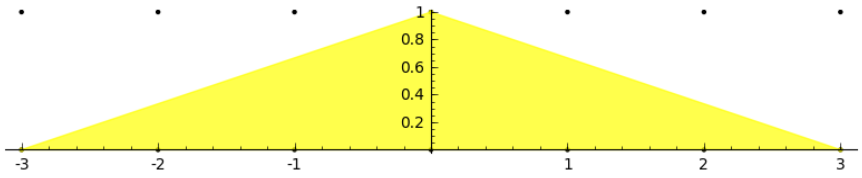
For each $z \in F \cap \mathbb{Z}^n$ and each facet F, we define a **spindle**

$$S_{F,z}(f,B) = P_F(f,B) \cap (z + f - P_F(f,B))$$

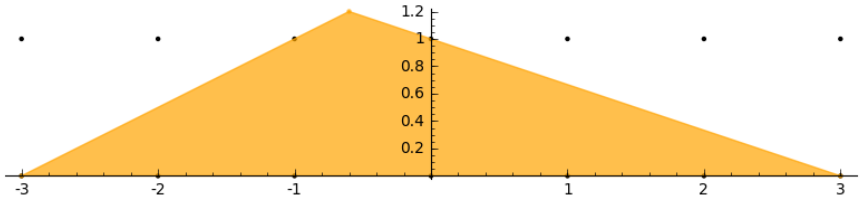
In addition, we define the union of all spindles arising from the facet F as

$$R_F(f,B) = \bigcup_{z \in F \cap \mathbb{Z}^n} S_{F,z}(f,B)$$

We refer the set of all facets of B as Facets(B).



(a) A triangle that is lattice-free but not maximal



(b) A triangle that is maximal lattice-free

FIGURE 4.1. Examples of lattice-free triangles

DEFINITION 4.1.1. *[**AB14**] If B is a maximal lattice-free polytope and $f \in B$, then the **lifting region** $R(f, B)$ associated with the point f is defined as*

$$R(f, B) = \bigcup_{F \in \text{Facets}(B)} R_F(f, B)$$

Equipped with the natural Lebesgue measure that assigns volume 1 to torus $\mathbb{T}^n = \mathbb{R}^n/\mathbb{Z}^n$ [**BCK12**], researchers are interested in the volume of the lifting region $R(f, B)/\mathbb{Z}^n$, i.e., the region $R(f, B)$ is sent onto the torus $\mathbb{T}^n$ [**AB14**]. We denote the volume of the lifting region $R(f, B)/\mathbb{Z}^n$ as $\text{vol}_{\mathbb{T}^n}(R(f, B)/\mathbb{Z}^n)$.

By observation, a compact set $X \subseteq \mathbb{R}^n$ covers $\mathbb{R}^n$ by lattice translations, i.e., $X + \mathbb{Z}^n = \mathbb{R}^n$, if and only if $\text{vol}_{\mathbb{T}^n}(X/\mathbb{Z}^n) = 1$ [**BCK12**]. Researchers have been interested in what kinds of maximal lattice-free convex sets B with what values of f will give $\text{vol}_{\mathbb{T}^n}(R(f, B)/\mathbb{Z}^n) = 1$. Another question about the volumes of the lifting regions is whether there exists a relation between f and $\text{vol}_{\mathbb{T}^n}(R(f, B)/\mathbb{Z}^n)$ for a given B.

THEOREM 4.1.1. *[**AB14**] Let $B \subset \mathbb{R}^n$ be a maximal lattice-free polytope. The function $f \to \text{vol}_{\mathbb{T}^n}(R(f, B)/\mathbb{Z}^n)$, acting from B to $\mathbb{R}$, is the restriction of an affine function.*

Theorem 4.1.1 is a generalization of Theorem 4 in [**BCK12**], which states that if $B \subset \mathbb{R}^n$ is a maximal lattice-free **simplicial** polytope and $f \in B$, then $\text{vol}_{\mathbb{T}^n}(R(f,B)/\mathbb{Z}^n)$ is an affine function of the coordinates of f.

In [**BCK12**], Corollary 1 states that the set $\{\ f \in B \mid \text{vol}_{\mathbb{T}^n}(R(f,B)/\mathbb{Z}^n) = 1\ \}$ is a face of B. This implies that the minimum and maximum of $\text{vol}_{\mathbb{T}^n}(R(f,B)/\mathbb{Z}^n)$ are found when f is on some vertices of B.

4.2. Sage Code and Examples

(1) 'is_lattice_free(B)' checks if a polyhedron B is lattice-free. This method first calls B.integral_points() in Sage to obtain the integral points on B, then checks if none of those points is in the interior of B.

(2) 'is_maximal_lattice_free(B)' checks if a polyhedron B is maximal lattice-free by checking that B is lattice-free and every facet of B has at least one lattice point in its relative interior.

(3) 'lifting_region(B, f)' constructs the lifting region $R(f,B)$ in the ways as Definition 4.1.1 states.

(4) 'pick_f(B)' returns a random f in B.

(5) 'translate_lifting_region(B, lr)' constructs $R(B,f)/\mathbb{Z}^n$ on the unit cube $[0,1]^d$, where d is the ambient dimension of B. This method returns $R(B,f)/\mathbb{Z}^n$ as a 'PolyhedralArrangement' instance. The method first makes some lattice translations on $R(f,B)$ and truncates them to the unit cube by taking non-empty intersections with the cube.

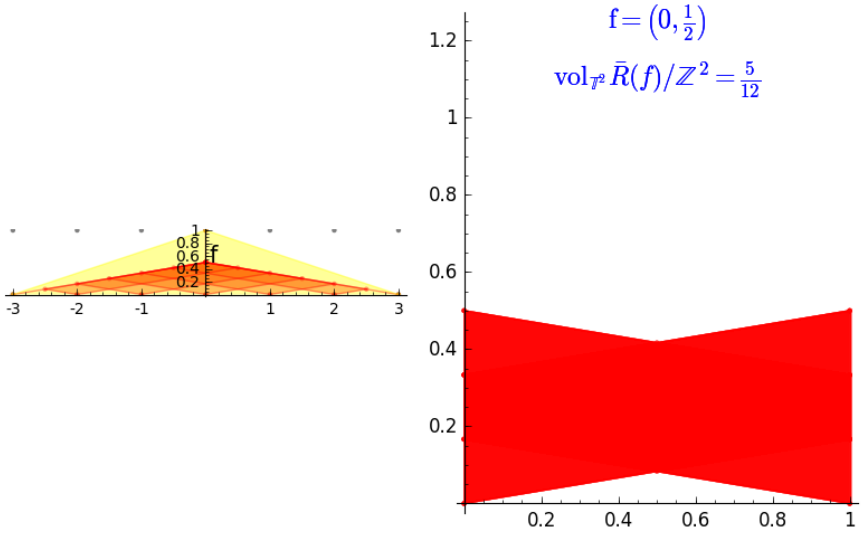
(6) 'find_volume_of_lifting_region(B, lr)' computes $\text{vol}_{\mathbb{T}^n}(R(B,f)/\mathbb{Z}^n)$ by using inclusion-exclusion property on the volumes of the elements in the polyhedral arrangement constructed by 'translate_lifting_region(B, lr)'.

(7) 'lifting_graphics(B, f=None)' plots the lifting region lr of $R(B,f)$. If f is not given, then the method picks an f by calling 'pick_f(B)'. The method also plots $R(B,f)/\mathbb{Z}^n$ which is

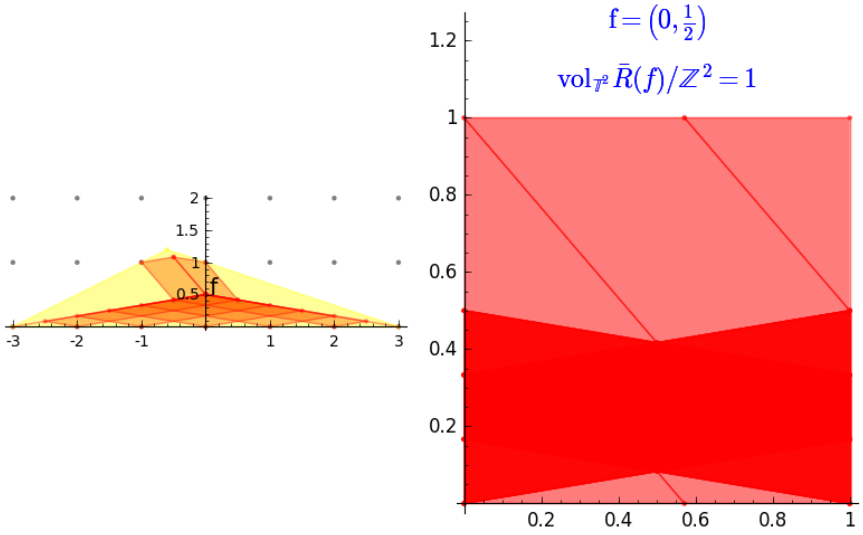
constructed by 'translate_lifting_region(B, lr)'. The method shows the value of f and the corresponding volume, $\mathrm{vol}_{\mathbb{T}^n}(R(B,f)/\mathbb{Z}^n)$ in the graph.

Readers can use the Sage code to construct lifting regions and the translated lifting regions that intersected with a unit cube. Readers can also use the code to calculate the volumes of the lifting regions that are sent to the torus to better study lifting regions in the theory of cut-generating functions.

In the following pages, we will provide the lifting graphics on some 2-D and 3-D examples. For 2-D examples, the lattice-free convex set B is colored in yellow. The lifting regions are colored in orange, and the translated lifting regions that interesect with the unit cube are colored in red. For 3-D examples, the boundary of the lattice-free convex set B is colored in blue. The lifting regions are all colored in red. The gray points are the integral points of the bounding box of B.

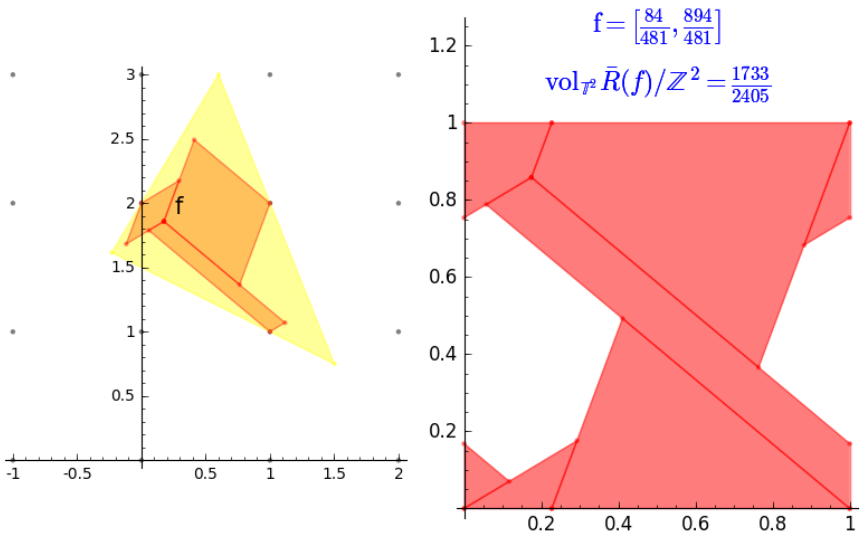


(a) Lifting graphics for the convex set in Figure 4.1a

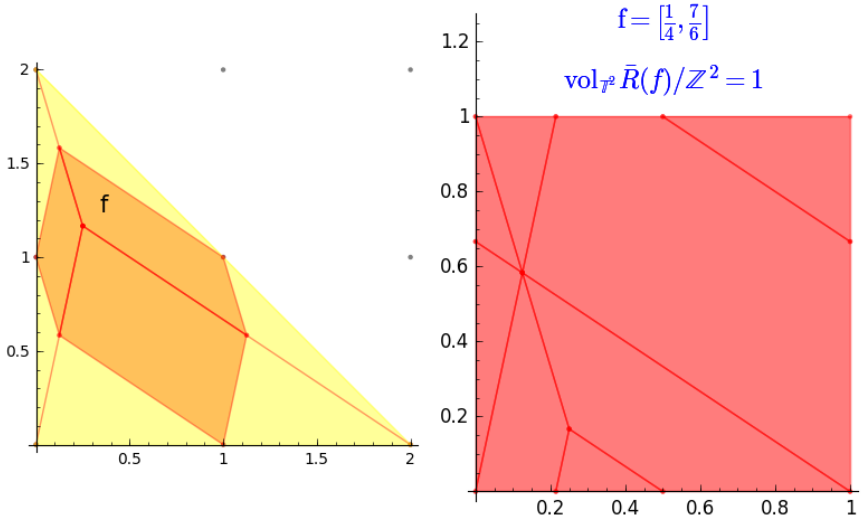


(b) Lifting graphics for the convex set in Figure 4.1b

FIGURE 4.2. Lifting graphics for some 2-D lattice-free polytopes in Figure 4.1

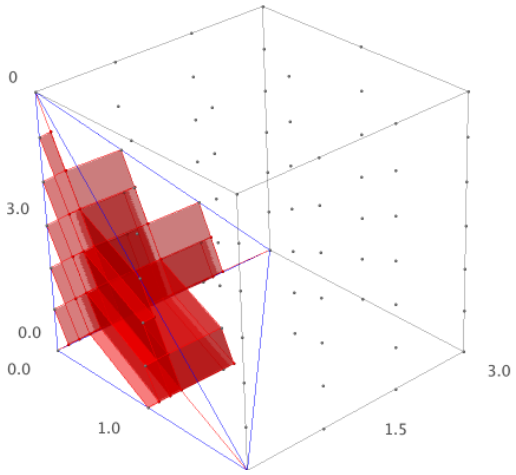


(a) Lifting graphics for the example in Figure 1(a) in [**BCK12**]

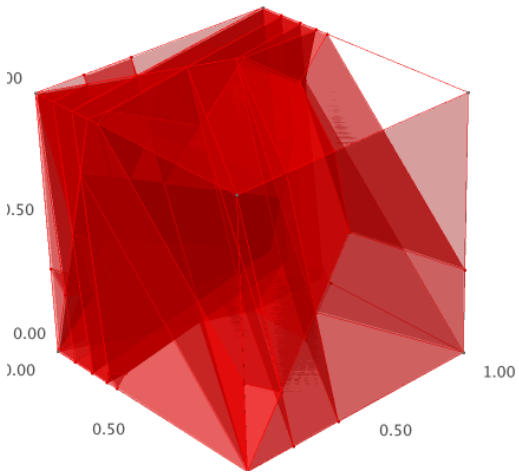


(b) Lifting graphics for the example in Figure 1(b) in [**BCK12**]

FIGURE 4.3. Lifting graphics for some 2-D polytopes from literature

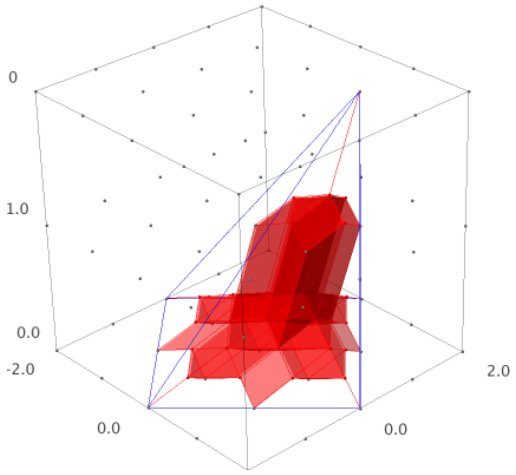


(a) Lifting graphics for a simplex denoted as M1 in Theorem 2.2 in [**Wag11**] with randomly generated f

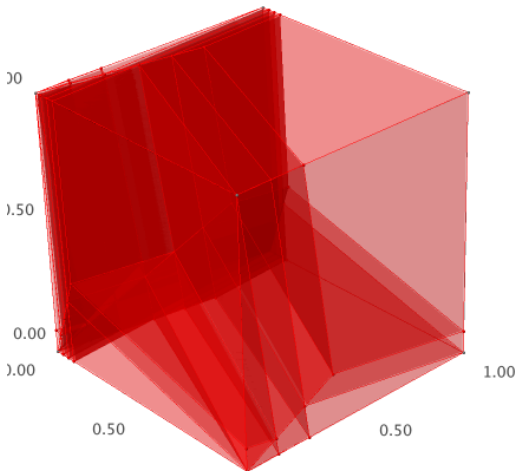


(b) Translated lifting regions of the convex set in Figure 4.4a that interesect with the unit cube

FIGURE 4.4. Lifting graphics for a 3-D simplex in [**Wag11**]



(a) Lifting graphics for a pyramid denoted as M8 in Theorem 2.2 in [**Wag11**] with randomly generated f



(b) Translated lifting regions of the convex set in Figure 4.5a that interesect with the unit cube

FIGURE 4.5. Lifting graphics for a pyramid in [**Wag11**]

Bibliography

[AB14] G. Averkov and A. Basu, *On the Unique-lifting Property*, Proceedings of IPCO (2014), 76—-87.

[AZ14] M. Aigner and G. M. Ziegler, *Proofs from the book: Fifth edition*, 2014.

[BC09] V. Borozan and G. Cornuéjols, *Minimal Valid Inequalities for Integer Constraints*, Mathematics of Operations Research **34** (2009), no. 3, 538–546.

[BCK12] A. Basu, G. Cornuéjols, and M. Köppe, *Unique Minimal Liftings for Simplicial Polytopes*, Mathematics of Operations Research **37** (2012), no. 2, 346–355.

[BHK16] A. Basu, R. Hildebrand, and M. Köppe, *Light on the infinite group relaxation I: foundations and taxonomy*, 4or **14** (2016), no. 1, 1–40, 1410.8584.

[DRS10] J. A. De Loera, J. Rambau, and F. Santos, *Triangulations*, Algorithms and Computation in Mathematics, vol. 25, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[FP09] C. A. Floudas and P. M. P. M. Pardalos, *Encyclopedia of optimization*, Springer, 2009.

[Knu98] D. E. Knuth, *The Art of Computer Programming Volume 3. Sorting and Searching*, Addison Wesley **3** (1998), 829.

[Koz08] D. Kozlov, *Combinatorial Algebraic Topology*, vol. 21, 2008.

[Par12] L. Paris, *K(π,1) conjecture for Artin groups*, arXiv.org **23** (2012), no. 2, 361–415, 1211.7339.

[Sch98] A. Schrijver, *Theory of linear and integer programming*, Wiley, 1998.

[Sen13] M. Senechal, *Shaping space: Exploring polyhedra in nature, art, and the geometrical imagination*, vol. 1, Springer New York, New York, NY, 2013.

[Sta04] R. Stanley, *An introduction to hyperplane arrangements*, Lecture notes, IAS/Park City Mathematics Institute (2004), 110.

[Wag11] C. Wagner, *Maximal lattice-free polyhedra : finiteness and an explicit description in dimension three*, (2011), arXiv:1010.1077v2.