Raymond and Beverly Sackler
Faculty of Exact Sciences
The Blavatnik School of Computer Science

Lines Tangent to Four Polytopes in $\mathbb{R}^3$

Thesis submitted in partial fulfillment of the requirements for the M.Sc.

degree in the School of Computer Science, Tel-Aviv University

by

Asaf Porat

This work has been carried out at Tel-Aviv University
under the supervision of Prof. Dan Halperin

May 2012

Acknowledgments

Many people had great influence on this thesis and its author during the research period. I deeply thank my advisors, Dr. Efi Fogel and Prof. Dan Halperin, for their help in guidance, support, and encouragement, and for introducing me to the field of applied computational geometry. Special thanks are given to Efi for providing the basis for the player software, which enabled the creation of the 3D figures of this thesis.

I would also like to thank all other members of the applied computational geometry lab at the computer science school of Tel-Aviv University who provided support and useful suggestions. Special thanks are given to Michael Hemmer for introducing and helping with the field of computational algebra.

I also thank Linqiao Zhang who provided us with Redburn's code that was used for the experiments. Zhang used it as part of an implementation of an algorithm that constructs the visibility skeleton.

Finally, I would like to express my love and gratitude to my beloved family; to my dear spouse Ganit for her support and patience over the last two years.

Work on the thesis has been supported in part by the 7th Framework Programme for Research of the European Commission, under FET-Open grant number 255827 (CGL—Computational Geometry Learning), by the German-Israeli Foundation (grant no. 969/07), and by the Hermann Minkowski–Minerva Center for Geometry at Tel Aviv University.

Abstract

We present a method for computing all lines tangent to four geometric objects taken from a set of n geometric objects in three-dimensional Euclidean space. The problem of finding all lines tangent to four geometric objects arises in many fields of computation such as computer graphics (visibility computations), computational geometry (line transversal), robotics and automation (assembly planning), and computer vision.

The first type of geometric objects that we handle are line segments. The number of lines tangent to four line segments is either 0, 1, 2, 3, 4, or infinite. We present an exact implementation of an efficient output-sensitive algorithm, such that given a set $\mathcal{S} = \{S_1, S_2, ..., S_n\}$ of n line segments, it finds all the lines tangent to at least four line segments of $\mathcal{S}$. We do not assume general position. Namely, the algorithm and its implementation are robust and the code properly handles all degenerate cases, e. g., a line segment may degenerate to a point, several segments may intersect, be co-planar, parallel, concurrent, lie on the same supporting line, or even overlap. Additionally, we enhance the output sensitive algorithm and its implementation to solve the problem where the given objects are convex polytopes. Given a set $\mathcal{P} = \{P_1, P_2, ..., P_k\}$ of k strictly pairwise disjoint convex polytopes, with a total number of n edges, the algorithm finds all the lines tangent to at least four elements of $\mathcal{P}$.

Theoretical bounds for the algorithms are $O((n^3+I)\log n)$ running time, and $O(n \log n + J)$ working storage, where n is the input size, I is the output size, and J is the maximum number of intersection in a single arrangement; J is bounded by $O(n^2)$. I is bounded by $O(n^4)$ when the input is of n line segments, and by $O(n^2k^2)$ when the input is of k polytopes with n edges in total.

Contents

1 Introduction **1**

2 Preliminaries **9**

2.1 Definitions . . . 9

2.2 Representation . . . 10

2.3 CGAL and the *2D Arrangements* Package . . . 11

3 From $\mathbb{R}^3$ to Two-Dimensional Surfaces **13**

3.1 Directions Are Linearly Independent . . . 13

3.2 Directions Are Not Linearly Independent . . . 15

3.2.1 Directions of L_1 and L_2 are Linearly Independent . . . 15

3.2.2 Directions of L_1 and L_2 Are Dependent . . . 17

3.2.3 S_1 and S_2 Intersect . . . 17

3.2.4 S_1 and S_2 Are Collinear . . . 17

4 Algorithmic and Implementation Details **19**

4.1 Algorithm Overview . . . 19

4.1.1 The Processing of Arrangements in the Plane . . . 21

4.1.2 The Processing of Arrangements on the Sphere . . . 22

4.1.3 The Processing of Collinear Line Segments . . . 23

4.1.4 Complexity Analysis . . . 24

4.2 Implementation with CGAL . . . 24

4.3 Application Interface . . . 25

4.3.1 Output Elements . . . 26

5 Lines Tangent to Four Strictly Disjoint Polytopes **29**

5.1 Algorithm Overview . . . 29

5.2 Constructing the Arrangement . . . 30

5.3 Removing Cells From the Arrangement . . . 33

5.3.1 Degeneracies . . . 33

5.4 Complexity Analysis 34
5.5 Implementation with CGAL. 34

6 Experiments 37
6.1 Grid 37
6.1.1 Transformed Grid 38
6.2 Random Input 39

7 Conclusions and Future Work 41

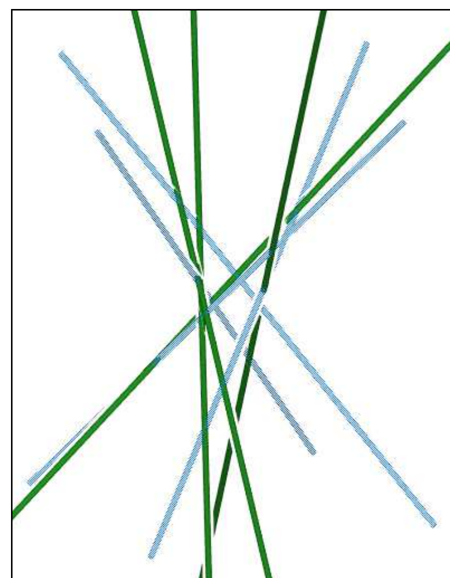
List of Figures

1.1 Hyperbolic paraboloid and hyperboloid of one sheet. 2
1.2 Infinite lines tangent to four lines 3

3.1 The mapping of a hyperboloid of one sheet 13
3.2 Mappings of three line segments in various configurations 15
3.3 Mappings of three coplanar line segments 16

6.1 Two grids parallel to the $z = 0$ plane. 38

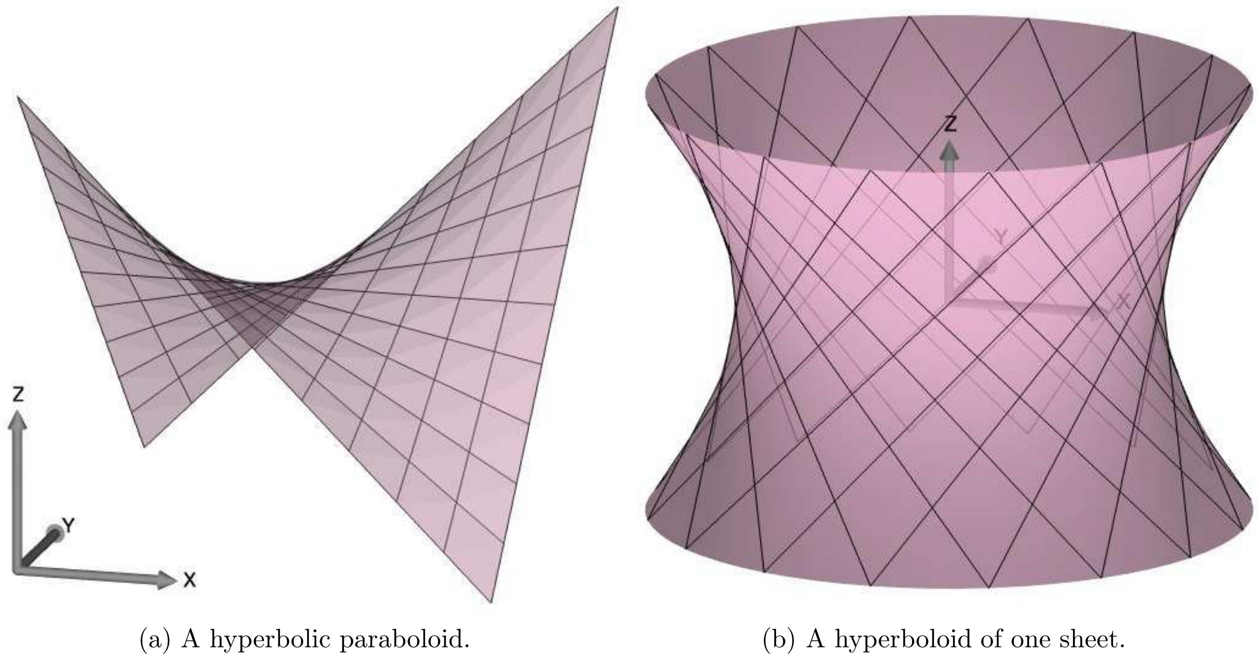
1 Introduction



In this thesis we study lines in three-dimensional Euclidean space. Our interest focuses at lines tangent to four geometric objects. These objects can be line segments, or convex bounded polyhedra (referred to as convex polytopes) including convex polygons[1]. When the input consists of line segments, we refer to the problem as the *lines-tangent-to-segments* problem, or LTS for short. When the input consists of polytopes, we refer to the problem as the *lines-tangent-to-polytopes* problem, or LTP for short. The figure to the right depicts four lines (drawn in green) tangent to four line segments (drawn in blue with a halftone pattern). LTS and LTP are fundamental problems that arise in a variety of domains and in many fields of computation such as computer graphics and computer vision (visibility computations), computational geometry (line transversal), and robotics and automation (assembly planning). Computing visibility information, for example, is crucial to many problems in computer graphics, vision, and robotics, such as computing umbra and penumbra cast by a light source [DDE$^+$09].

The lines tangent to a single line in $\mathbb{R}^3$ have three degrees of freedom; those tangent to two lines have two degrees of freedom; those tangent to three lines have one degree of freedom. The number of lines tangent to four lines in $\mathbb{R}^3$ is 0, 1, 2, or infinite. The problem of finding the line transversals to line segments (the set of lines tangent to all given line segments) was studied by H. Brönnimann, et al. [BEL$^+$05]. They showed that the number

[1] By definition in three or higher dimensions any point of intersection between two lines is also a tangency point.



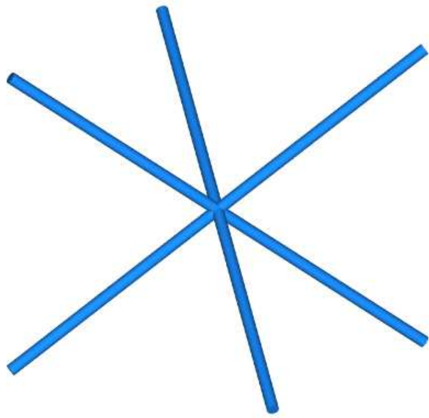
(a) A hyperbolic paraboloid. (b) A hyperboloid of one sheet.

Figure 1.1: Three skew lines in $\mathbb{R}^3$ form one of these ruled surfaces. They are all on the same ruling. All the lines on the other ruling are tangent to the three lines.

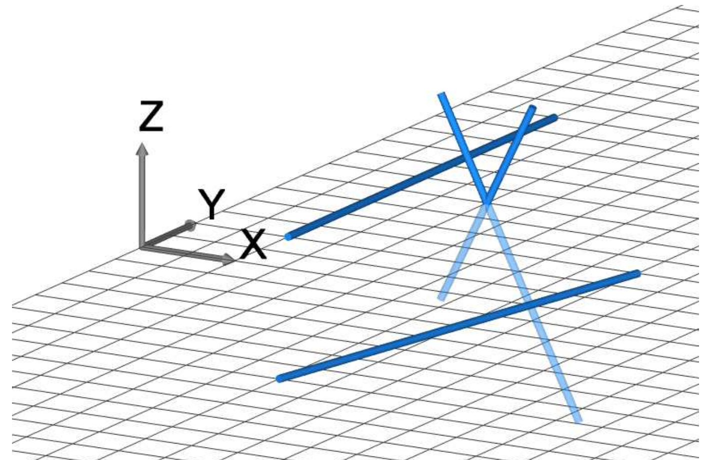
of lines tangent to four arbitrary line segments in $\mathbb{R}^3$ is 0, 1, 2, 3, 4, or infinite. The latter may happen only if the segments lie in one of the following configurations:[2] (i) The four line segments are coplanar. (ii) Three line segments lie in the same plane P, which is pierced by the fourth segment. (iii) Two line segments lie in the same plane P, while the other two pierce P at the same point; see Figure 1.2b. (iv) At least three line segments intersect at the same point; see Figure 1.2a. (v) At least two line segments overlap. (vi) All four line segments are contained in the same ruling of a *hyperbolic paraboloid* or a *hyperboloid of one sheet*; see Figure 1.1a and Figure 1.1b, respectively. In addition, Brönnimann et al. showed that the lines lie in at most four maximal *connected components*, see definition in Section 2.1.

A straightforward method to find all the lines that intersect four lines, given a set of n lines, examines each quadruplet of lines. The examination is simplified using the Plücker coordinate representation. The Plücker coordinates of a line L, defined by a sample point p on the line and a vector $\vec{u}$ that expresses the direction of the line, are the six-tuple $< \vec{u}, \vec{u} \times p >$. The *side product* of two lines L_a and L_b with Plücker coordinates $a = [a_1, \ldots, a_6]$ and $b = [b_1, \ldots, b_6]$, is defined as [TH99]: $a \odot b = (a_1b_4 + a_2b_5 + a_3b_6 + a_4b_1 + a_5b_2 + a_6b_3)$. The side product is zero whenever L_a and L_b intersect or are parallel and non zero otherwise. The method of finding intersecting lines using the Plücker coordinates representation was used by Hohmeyer and Teller [TH99] and also described by Redburn [Red03]. This method was later used by Everett et al. [ELLZ09] as a building block for the problem of finding line transversals. The use of Plücker coordinates simplifies the algebra but does not obviate the need to process each quadruplet of lines. The running time of this method is $O(n^4)$.

[2] Some conditions are omitted, e. g., no pair of the line segments are collinear.

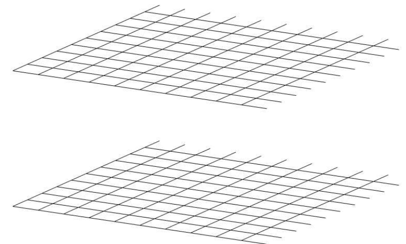


(a) Three concurrent line segments.



(b) Two coplanar line segments lying in a plane P, and two additional line segments intersecting P at the same point.

Figure 1.2: Configurations of line segments in which infinite number of lines are tangent to four line segments.



The combinatorial complexity of all the lines that intersect four line segments of a set of n line segments is $\Theta(n^4)$ (counting maximal connected components). The lower bound can be established by placing two grids of $n/2$ line segments each in two parallel planes and passing a line through every two intersection points, one from each grid. However, in many cases the number of output lines is considerably smaller. The size of the output tends to be even smaller, when the input consists of line segments (as opposed to lines), which is typically the case in practical problems, and it is expected to decrease with the decrease of the lengths of the input line segments; see Chapter 6.

Related Work

Much research has been devoted to finding all lines tangent to four geometric objects in $\mathbb{R}^3$. For a scene of n triangles in $\mathbb{R}^3$ the combinatorial complexity is $O(n^4)$ in the worst case, even when the triangles form a terrain [CS89]. For n disjoint convex polytopes of constant size each, De Berg et al. [dBEG98] showed a lower bound of $\Omega(n^3)$. The upper bound remains $O(n^4)$. For k convex polytopes of total complexity of n with $k \ll n$, where the convex polytopes may intersect, Brönnimann et al. [BDD$^+$07] proved the tight bound of $\Theta(n^2k^2)$. Glisse [GL10] showed a $\Theta(n^4)$ bound for the number of lines tangent to four balls of a set of n unit balls.

Implementations

Using Plücker coordinate representation, Hohmeyer and Teller [TH99] implemented an $O(n^4)$ algorithm, which finds all the lines tangent to four of n lines. J. Redburn implemented an

$O(n^4)$ algorithm, which finds all the lines tangent to four of n triangles in $\mathbb{R}^3$ [Red03]. Brönnimann et al. [BDD+07] suggested an $O(n^2k^2 \log n)$ time and $O(nk^2)$ space algorithm, for the LTP problem, where k is the number of polytopes. This algorithm was later implemented by Zhang et al. [ZEL+08] as part of the computation of the 3D visibility skeleton. They experimentally measured the running time of their algorithm and showed that it is proportional to $n^{3/2}k \log k$, while the algorithm's worst-case running time complexity is $O(n^2k^2 \log k)$.

Background and Motivation

Our interest in finding solutions to the LTP problem is motivated among the other by assembly-planning problems.

An *assembly* is a collection of pairwise interior disjoint polyhedra in some relative position in $\mathbb{R}^3$. Assembly partitioning is the application of a sequence of transforms to partition an assembly into its basic polyhedra. Each transform partitions a subset of the assembly, say $\mathcal{A}$, referred to as a sub-assembly, into ℓ subsets of $\mathcal{A}$, $\{A_1, A_2, ..., A_\ell\}$, such that $\bigcup_{i=1}^{\ell} A_i = \mathcal{A}$ and for $i \neq j$, $A_i \cap A_j = \emptyset$; ℓ is referred to as the *number of hands*. The assembly-partitioning problem is to find a sequence of partitioning transforms along with the corresponding sub-assemblies, or announce that such a sequence does not exist. In general, the problem reduces to finding a single transform. It turns out that if we confine ourselves to $\mathbb{R}^3$, treat each polyhedron as a rigid body, use only two hands, and consider each transform as a sequence of translations only, the problem can be solved, but the solution is based on the geometric operation at hand, that is, finding all lines tangent to four line segments of a given set. Finding the solution for the reduced assembly-partitioning problem, is the topic of ongoing research. In the following three paragraphs we highlight the motivation, showing how a solution to the problem at hand can help in finding a solution to the reduced assembly-partitioning prblem.

Finding a single transform in our context is formally stated as follows: Given an assembly $\mathcal{A} = \{P_1, \ldots, P_m\}$ in $\mathbb{R}^3$ find a proper subset $\mathcal{A}' \subsetneq \mathcal{A}$ and a sequence $\tau_1, \ldots, \tau_k$ of translations, where τ_k is a translation to infinity, such that $\mathcal{A}'$ can be moved to infinity as a rigid body applying $\tau_k \circ \cdots \circ \tau_1$ without intersecting the interior of a polyhedron from $\mathcal{A} \setminus \mathcal{A}'$.

When the transform consists of a single translation ($k = 1$), it can be represented as a direction in $\mathbb{R}^3$. Such approach was described by Halperin et al. [HLW00] and by Wilson and Latombe [WL94]. The motion-space arrangement (see definition in Section 2.1) in this case is a two-dimensional arrangement embedded on the unit sphere. Consider a cell C in this arrangement. For all $1 \leq i < j \leq m$, if P_i intersects P_j when P_i is translated along some direction $\vec{d} \in C$, then P_i intersects P_j when P_i is translated along any direction $\vec{d} \in C$. Every cell of the arrangement is associated with the *directional blocking graph* (DBG). The vertex set of this graph is $\mathcal{A}$, and there is an edge from P_i to P_j if P_i intersects P_j when translated along any of the corresponding directions. If the DBG for any region has at least two strongly connected components, the corresponding subsets of objects can be moved apart from the rest using a single translation to infinity. Note that considering all subsets of parts in isolation would lead to an exponential runtime; hence, resorting to the DBG is essential to achieve a polynomial time algorithm.

Due to the low number of degrees of freedom (DOF) for the case $k = 1$ the entire motion-space arrangement can be computed [FH08]. However, computing the complete motion-space arrangement in the general case where the number of DOF is $3k - 1$ is infeasible for any k larger than 1. Observe that if there exists any sequence of translations that move a sub-assembly to infinity, then there is also a sequence of translations such that at least one moving polyhedron and one stationary polyhedron touch each other along the way during each translation of the sequence. In other words, it is sufficient to consider only vertices of the arrangement. Instead of using a full dimensional configuration-space, the idea is to solve the problem in a three-dimensional configuration space that contains the Minkowski sums $\mathcal{M} = \{P_i \oplus P_j \,|\, 1 \leq i < j \leq m\}$. A sequence of partitioning translations in this space is a polyline unbounded on one end. Each line segment (or ray) that composes the polyline is contained in a line that is tangent to four edges of four different polyhedra from $\mathcal{M}$.

Finding all lines tangent to four polytopes can also be used to solve visibility problems. In visibility problems the scene is represented as a union of not necessarily disjoint polygonal or polyhedral objects. Some objects can be seen from a moving view-point, others are occluded. This moving view-point lies on a line segment S in $\mathbb{R}^3$. Just like as in the assembly-partitioning problem, the lines of sight emanate from S are tangent to three polytopes in the scene. That is, the critical points on S are all the points of intersection between lines tangent to S and three other line segments in the scene, each from a different polytope. Computing visibility information is crucial to many problems in computer graphics, vision, and robotics, such as computing umbra and penumbra cast by a light source [DDE+09].

Software

Generic Programming

The software described in this thesis is written in the C++ programming language and adheres to the generic-programming paradigm. Generic programming is a discipline that consists of the gradual lifting of concrete algorithms abstracting over details, while retaining the algorithm semantics and efficiency [MS88]. This capability allows reuse of the software components in a variety of situations. In C++ generic programming is characterized by extensive use of template functions and classes.

A generic implementation can be described by a *concept* and its *models*. A *concept* is a set of requirements that determine the properties of a type. A *model* of a concept is an actual type that satisfies the requirements of the concept. A *refinement* of a concept is an extension of the requirements of another concept. A *traits* is a class that provides a way to associate information, typically a type or several types and possibly operations on these types, with a compile-time entity. It is often a model of some specific concepts called *traits concept*. For example, let's look at the `class template std::iterator_traits<T>`.

```
template <class Iterator> struct iterator_traits{
   typedef typename Iterator::difference_type difference_type;
   typedef typename Iterator::value_type value_type;
   typedef typename Iterator::pointer pointer;
```

```
   typedef typename Iterator::reference reference;
   typedef typename Iterator::iterator_category iterator_category;
}
```

One of the iterator associated types is the value type. Pointers are also iterators, for example, the pointer `int*` is an iterator, its value type is `int`. A generic algorithm that accepts as an input, an iterator may need to declare a variable that its type is the iterator value type as in the following example:

```
template <class Iterator>
bool contain(Iterator begin, Iterator end, iterator_traits<Iterator>::value_type val)
{
   for (++begin; begin != end; ++begin){
      if (val == *begin)
         return true;
   }
   return false;
}
```

The class `iterator_traits` implements a mechanism that allows such declarations. At first glance it seems that the class `iterator_traits` is redundant and it is sufficient to require that each iterator will contain nested types. This cannot work, since it is impossible to declare `Iterator::value_type` when `Iterator` is of type `int*`. In order to overcome this problem, the standard template library (STL) provides specialized versions for pointers and pointers to const:

```
template <class T> struct iterator_traits<T*> {
   typedef ptrdiff_t difference_type;
   typedef T value_type;
   typedef T* pointer;
   typedef T& reference;
   typedef random_access_iterator_tag iterator_category;
}
template <class T> struct iterator_traits<const T*> {
   typedef ptrdiff_t difference_type;
   typedef T value_type;
   typedef const T* pointer;
   typedef const T& reference;
   typedef random_access_iterator_tag iterator_category;
}
```

Computational Geometric Software

The Computational Geometry Algorithms Library (CGAL) [2][3] is an open-source software library of efficient and reliable geometric algorithms. The code of the library is written in

[3]Throughout the thesis a number in brackets (e.g., [3]) refers to the link list on page 47, and an alphanumeric string in brackets (e.g., [FSH08]) is a standard bibliographic reference.

C++ and rigorously adheres to the generic-programming paradigm. It also follows the exact geometric-computation paradigm [Yap04, YD95] to achieve robustness and efficiency with exact results.

The implementation presented in this thesis is based on a package of CGAL called *2D Arrangements* [FWH11]. It supports the robust construction and maintenance of arrangements induced by curves embedded on certain two-dimensional parametric surfaces in three-dimensional space[4] [BFH+10], and robust operations on them. The implementation uses in particular arrangements induced by geodesic arcs embedded on the sphere [BFH+10] and arrangements induced by hyperbolic arcs in the plane. We plan to make our new component available as part of a future public release of CGAL. The ability to robustly construct such arrangements and carry out exact operations on them using (multi-precision) rational and algebraic arithmetic is a key property that enables our efficient and certified implementation.

Contribution of the Thesis

We present exact, complete, and robust implementation of efficient output-sensitive algorithms to solve the LTS and LTP problems.

The algorithm utilizes the idea of McKenna and O'Rouke [MO88] to represent the set of lines that intersect three lines in $\mathbb{R}^3$ as a rectangular hyperbola with a vertical and a horizontal asymptotes in $\mathbb{R}^2$. This idea was later on used by Olivier Devillers et al. [DGL08] to solve predicates for line transversals to lines and line segments. First, we show how to use this approach to solve the LTS problem. Then, we show how to enhance this approach to solve the LTP problem. The algorithm is implemented on top of the CGAL library [2], and is mainly based on the *2D Arrangements* package of the library [WFZH07, FWH11]. The implementation for line segments is complete and robust, as it handles all degenerate cases and guarantees exact results. We also report on the performance of our algorithm and implementation compared to others. Two number types of arbitrary precision are required for the implementation of the algorithm. The coefficients of the input objects must be represented by rational numbers, of unlimited precision. However, a rational number type cannot represent the coefficients of the output lines in an exact manner; thus, an algebraic number type of unlimited precision is necessary. In our algorithm and implementation of the LTS problem we do not assume general position. Namely, the algorithm and its implementation are robust and handle all cases. Examples of degenerate cases are: A line segment may degenerate to a point, several segments may intersect pairwise, be coplanar, parallel, concurrent, lie on the same supporting line, or even overlap. The enhancement to convex polytopes does not handle all degenerate cases; it is restricted to strictly disjoint (non-touching) polytopes. This restriction is due to implementation issues and can be lifted with some extra programming effort. The algorithm of the LTP problem computes only lines tangent to at least four polytopes, while in some cases three, two, or even one polytope determines tangent lines.

Theoretical bounds for the algorithms are $O((n^3+I)\log n)$ running time, and $O(n\log n + J)$ working storage, where n is the input size, I is the output size, and J is the maximum

[4]Arrangements on surfaces are supported as of CGAL version 3.4, albeit not documented.

number of intersection in a single arrangement; J is bounded by $O(n^2)$. I is bounded by $O(n^4)$ when the input is of n line segments, and by $O(n^2k^2)$ when the input is of k convex polytopes with n edges in total.

This is the first time an exact and efficient algorithm is implemented for the LTP problem. Other implementations are not as efficient (running time $O(n^4)$) and do not handle all degenerate cases [TH99, Red03].

2 Preliminaries

In this chapter we review some basic mathematical properties used in this thesis. Section 2.1 provides general definitions concerning lines in space. Section 2.2 describes the mapping of three pairwise skew lines in $\mathbb{R}^3$ to the (two-dimensional) plane. Software components and additional technical background are reviewed in Section 2.3.

2.1 Definitions

Definition 2.1 (2D Arrangement)**.** *A finite collection $\mathcal{C}$ of geometric objects (e. g., lines or hyperbola) contained in a two-dimensional parametric surface (e. g., the plane and the sphere) subdivides the ambient space into (i) cells of dimension* 0 *(*vertices*) embedded as points, (ii) cells of dimension* 1 *(*edges*) embedded as continuous curves, which are pairwise disjoint in their interiors, and (iii) cells of dimension* 2 *(*faces*). The* arrangement *$\mathcal{A}(\mathcal{C})$ is a data structure that maintains the incidence relations on the cells of the subdivision.*

Definition 2.2 (Plane Sweep)**.** *Given a finite collection $\mathcal{C}$ of x-monotone curves in the plane, the* plane-sweep *is an algorithmic framework that sweeps the plane using a vertical line and triggers an event for every endpoint and intersection point of the curves in $\mathcal{C}$.*

Definition 2.3 (Map Overlay)**.** *The* map overlay *of two subdivisions S_1 and S_2, embedded in a surface Σ, is a subdivision S embedded in Σ, such that there is a face f in S iff there are faces f_1 and f_2 in S_1 and S_2, respectively, such that f is a maximal connected subset of $f_1 \cap f_2$.*

We represent a line $L \subset \mathbb{R}^3$ by a point $p \in L$ and a direction $d \in \mathbb{R}^3 \setminus \{\mathcal{O}\}$ as $L(t) = p + t \cdot d$, where $\mathcal{O}$ denotes the origin and $t \in \mathbb{R}$. Clearly, this representation is not unique. A segment $S \subset L \subset \mathbb{R}^3$ is represented by restricting t to the interval $[a, b] \subset \mathbb{R}$. We refer to $S(a)$ and

$S(b)$ as the source and target points, respectively, and set $a = 0$ and $b = 1$. We denote the underlying line of a line segment S by $L(S)$. Two line segments, S_1 and S_2, in $\mathbb{R}^3$ are *intersecting* if there exist a point q, such that $q \in S_1$ and $q \in S_2$. Two line segments in $\mathbb{R}^3$ are *coplanar* if their supporting lines intersect or they are parallel. Two line segments *overlap* if they share a line segment. Three or more line segments are *concurrent* if they all intersect at a common intersection point. Two lines are *skew* if they are not coplanar; i. e., their supporting lines do not intersect and are not parallel.

Two lines tangent to the same four line segments are at the same *connected component* (of lines) iff one of the lines can be continuously moved into the other while remaining tangent to the same four line-segments. A *convex polytope* is the convex hull of a point set in $\mathbb{R}^3$. A plane is *tangent* to a polytope if it intersects the polytope at either a facet, an edge, or a vertex, and one of the closed half-spaces defined by the plane contains the polytope. A line is *tangent* to a polytope if it intersects the polytope and it is contained in a tangent plane.

A surface S is *ruled* if through every point p on S there exist a straight line that passes through p and lies on S. A surface S is *doubly ruled* if through every point p on S, there exist two distinct lines, L_1 and L_2, that pass through p and lie on S. It is well known that all the lines tangent to three pairwise disjoint skew lines in $\mathbb{R}^3$ are on one ruling of a doubly ruled surface [PW01]. This surface is either a *hyperbolic paraboloid* or a *hyperboloid of one sheet*; see Figure 1.1. All the lines on the other ruling of one of these ruled surfaces are tangent to the three lines. A fourth line may be on the same ruling. In that case the number of lines tangent to the four lines is infinite. Otherwise, it may cross the surface in 0, 1, or 2 points, such that, there are respectively 0, 1, or 2 lines tangent to the four lines.

2.2 Representation

Given two lines L_1 and L_2 we define a map $\Psi_{L_1L_2}$ as follows:

$$\Psi_{L_1L_2}(p_3) = \{(t_1, t_2) \in \mathbb{R}^2 \mid L_1(t_1),\ L_2(t_2),\ \text{and } p_3 \text{ are collinear}\}.$$

That is, $\Psi_{L_1L_2}$ maps a point in $\mathbb{R}^3$ to a set in $\mathbb{R}^2$. This set, which might be empty, corresponds to all lines that contain p_3 and intersect L_1 and L_2. Now consider the pair $(t_1, t_2) \in \mathbb{R}^2$. If $L_1(t_1) \neq L_2(t_2)$, then this pair uniquely defines a line, namely, the line that intersects L_1 and L_2 at $L_1(t_1)$ and $L_2(t_2)$, respectively. Thus, for skew lines L_1 and L_2 there is a canonical bijective map between $\mathbb{R}^2$ and all lines that intersect L_1 and L_2. It follows that for disjoint lines L_1 and L_2 and a third line L_3 the set $\Psi_{L_1L_2}(L_3)$ is sufficient to represent all lines that intersect L_1, L_2, and L_3, where $\Psi_{L_1L_2}(L_3) = \{\Psi_{L_1L_2}(q) \mid q \in L_3\}$.

The characterization of $\Psi_{S_1S_2}(S_3)$ serves as the theoretical foundation of the algorithm that solves the LTS problem. Since $\Psi_{S_1S_2}(x) = \Psi_{L(S_1)L(S_2)}(x) \cap [0, 1]^2$, it is sufficient to analyze $\Psi_{L_1L_2}(S_3)$ for a line segment S_3.

2.3 CGAL and the *2D Arrangements* Package

Implementers of computational geometry algorithms typically face two challenges: (i) Achieving efficiency and robustness at the same time, and (ii) successfully handling degenerate cases.

Geometric algorithms in theory are based on the machine model named "real RAM" [PS85]. This model assumes that arithmetic computation is performed with unlimited precision in real numbers and each operation costs constant time. In practice, these assumptions do not hold, as almost always constant time arithmetic operations and accurate arithmetic computation can not be supported at the same time for most common number types. Using floating-point arithmetic yields incorrect (intermediate) results due to inevitable rounding errors, which impairs the robustness of the application. Consider for example the *orientation* predicate; given three points, p_1, p_2, and p_3, in $\mathbb{R}^2$, their orientation is *left turn* if p_3 lies to the left of the oriented line l defined by p_1 and p_2, *right turn* if p_3 lies to the right of l, and *collinear* if p_3 lies on l. The evaluation of this predicate may be incorrect when using floating-point arithmetic. Incorrect results are obtained when the points are collinear or near collinear. Kettner et al. [KMP+08] show an example of the disastrous consequence this violation may have on the computation of the convex hull of a set of points, which relies on the orientation predicate. A naive attempt could realize this by carrying out each and every arithmetic operation using an expensive unlimited-precision number type. However, only the discrete decisions in an algorithm, namely the predicates, must be correct. This is a significant relaxation from the naive concept of numerical exactness, as it is possible to use fast inexact arithmetic (e. g., double-precision floating-point arithmetic [DP03]), while analyzing the correctness. CGAL in general, and the CGAL *2D Arrangements* package in particular, follow the *exact geometric-computation (EGC) paradigm* to achieve efficiency with exact results. EGC, as summarized by Yap [Yap04], simply amounts to ensuring that we never err in predicate evaluations. EGC represents a significant relaxation from the naive concept of numerical exactness. Here, computation is carried out using a number type that supports only inexact arithmetic (e. g., double-precision floating-point arithmetic), while analyzing the computation correctness. If the computation reaches a stage of uncertainty, the computation is redone using unlimited precision. In cases where such a state is never reached, expensive computation is avoided, while the result is still certified.

Additional difficulty lies in the successful handling of degenerate cases. A degenerate case may be provided as input or generated during the execution of the algorithm. Many geometric algorithms assume general position. General position is commonly assumed while describing an algorithm, or analyzing its running time. However, degenerate input is commonplace in reality. We have invested a significant amount of effort to successfully handle all possible degenerate cases that may arise in the input or in intermediate data.

CGAL is a software library that provides robust, efficient, and reliable generic implementations of algorithms and data structures in computational geometry, such as triangulations, Voronoi diagrams, Boolean operations on polygons and polyhedra, and arrangements of curves. The CGAL kernel consists of constant-size geometric objects, such as points, lines, and segments, and operations on objects of these types. Different coordinate systems are offered, i. e., Cartesian and homogeneous. The kernel provides certified geometric predicates,

such as, determining whether two geometric objects intersect, determining whether two objects are parallel, and computing the orientation of two geometric objects. The kernel also provides exact constructions of geometric objects, such as, the intersection of two geometric objects and the distance between two geometric objects. Certified predicate evaluation and exact construction are crucial for achieving robust implementation.

The implementation presented in this thesis heavily uses the *2D Arrangements* package, which supports arrangements embedded in the plane and arrangements embedded in the sphere. The *2D Arrangements* package supports the constructions and maintenance of two-dimensional arrangements embedded in certain orientable parametric surfaces in three dimensions [BFH$^+$10]. In addition to the ability to construct arrangements, the package supports various operations on arrangements, including traversing an arrangement, answering point-location queries on an arrangement, and overlaying two arrangements [FWH11]. The *2D Arrangements* package provides a convenient mechanism that uses observers [FWH11] to notify on arrangement-topology transformations an arrangement is subject to. The observer is an abstract class, which is attached to the arrangement, and receives notifications when the arrangement is modified. The notification functions are virtual, and can be overridden by the concrete observer classes that inherit from the base class. They are used to notify on local or global changes. All these operations are exploited by our implementation.

In the implementation we use arrangements induced by hyperbolas in the plane and arrangements induced by geodesic arcs on the sphere. The arrangement induced by the set of input curves $\mathcal{C}$ is stored as a doubly-connected edge list (DCEL) data structure [dBvKOS08, Chapter 2]. Each cell can be extended with additional information [FWH11, Chapter 6]; this functionality is intensively used by our implementation.

3

From $\mathbb{R}^3$ to Two-Dimensional Surfaces

In this chapter we describe how to map three line segments, S_1, S_2, and S_3 in $\mathbb{R}^3$ to a two-dimensional curve in the general case and to a point or a two-dimensional surface patch in degenerate cases. Section 3.1 describes the general position case, where the directions of S_1, S_2, and S_3 are linearly independent. Section 3.2 covers the case, where the directions are not linearly independent.

3.1 Directions Are Linearly Independent

In this section we discuss all cases in which the direction vectors of the underlying lines of the segments are linearly independent. In this setting we can always apply a rational affine transformation such that the three segments are given by $S_i(t_i) = p_i + t_i \cdot d_i,\ i \in \{1, 2, 3\}$, where $p_1 = (a, b, c)$, $p_2 = (d, e, f)$, $p_3 = \mathcal{O}$ and $d_i = e_i$ (where e_i denotes the unit vector along the ith axis). Thus, we continue with a refined case distinction that only depends on the coordinates of p_1 and p_2.

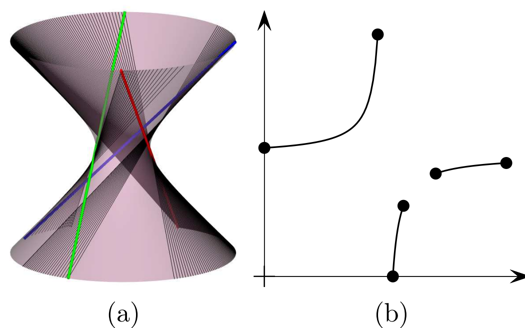


Figure 3.1: (a) Three surface patches the lines of which intersect three skew line segments, S_1, S_2, and S_3, in $\mathbb{R}^3$. These surface patches are contained in a hyperboloid of one sheet. (b) The point set $\Psi_{S_1S_2}(S_3)$.

$\mathbf{b \neq 0}$, $\mathbf{d \neq 0}$**, and** $\mathbf{c \neq f}$**:** All three lines are pairwise skew. Consider the points $L_1(t_1)$, $L_2(t_2)$, and $L_3(t_3)$. These points are collinear iff

$$|(L_1(t_1) - L_2(t_2)) \times (L_3(t_3) - L_2(t_2))| = 0\,. \quad (3.1)$$

These are three dependent equations in three unknowns. Eliminating t_3, we obtain the

following expression for t_2 in terms of t_1:

$$t_2(t_1) = \frac{e \cdot t_1 + (a \cdot e - d \cdot b)}{t_1 + a}\,. \tag{3.2}$$

It implies that $\Psi_{L_1L_2}(L_3)$ is a rectangular hyperbola with a vertical asymptote at $t_1 = -a$ and a horizontal asymptote at $t_2 = -e$. The point $(d-a, b-e)$ corresponds to the line that is parallel to L_3 and (by definition) intersects L_1 and L_2. Thus, this point is not in $\Psi_{L_1L_2}(L_3)$, as we consider affine space. Nonetheless, we are interested in $\Psi_{L_1L_2}(S_3)$, where $S_3 = \{L_3(t_3) \,|\, t_3 \in [0,1]\}$. Solving the system of Equation 3.1 for t_1 in terms of t_3 yields

$$t_1(t_3) = \frac{(d-a)t_3 + fa - dc}{t_3 - f}\,. \tag{3.3}$$

As t_3 is restricted to $[0,1]$, t_1 is restricted to $\mathcal{T} = \{t_1(t_3) \,|\, t_3 \in [0,1]\}$. $\Psi_{L_1L_2}(S_3)$ is not defined for values of $t_1 \notin \mathcal{T}$. Let $t' = \min(t_1(0), t_1(1))$ and $t'' = \max(t_1(0), t_1(1))$, where $t_1(0) = (dc - af)/f$ and $t_1(1) = (dc + a - fa - d)/(f-1)$. $t_1(t_3)$ is a hyperbola with a vertical asymptote at $t_3 = f$. If $f \in [0,1]$, then $\mathcal{T} = (-\infty, t'] \cup [t'', \infty)$. Otherwise, $\mathcal{T} = [t', t'']$. Recall that $\Psi_{L_1L_2}(S_3)$ is also not defined for the value $t_1 = -a$ due to the vertical asymptote of $t_2(t_1)$. It follows that $\Psi_{S_1S_2}(S_3)$ consists of at most three maximal connected components, where each component represents a patch of a ruled surface as depicted in Figure 3.1.

$\mathbf{b \neq 0,\ d \neq 0,}$ **and** $\mathbf{c = f}$**:** L_1 and L_2 intersect at $p = (d, b, c) = (d, b, f)$. L_3 is skew to both and intersects the $z = c$ plane (which is spanned by L_1 and L_2) at $q = (0, 0, c) = (0, 0, f)$. As in the previous case, $\Psi_{L_1L_2}(L_3)$ is a rectangular hyperbola. However, the point $(d-a, b-e) \in \Psi_{L_1L_2}(L_3)$ represents all lines containing p and intersecting L_3; see also Section 3.2.3. In case $q \notin S_3$, $\Psi_{L_1L_2}(S_3)$ degenerates to $\{(d-a, b-e)\}$.

$\mathbf{b = 0,\ d \neq 0,}$ **and** $\mathbf{c \neq f}$**:** L_1 intersects L_3 at $p = (0, 0, c)$. L_2 intersects the xz-plane at $q = (d, 0, f)$. Eliminating t_3 from Equation 3.1 we obtain $(e + t2)(a + t1) = 0$. Thus, $\Psi_{L_1L_2}(L_3)$ is a vertical line at $t1 = -a$ and a horizontal line at $t_2 = -e$. $t1 = -a$ represents all lines containing p and L_2. $t_2 = -e$ represents all lines in the xz-plane that contain q. Since we consider an affine space, the point $(d-a, -e) \notin \Psi_{L_1L_2}(L_3)$. The point set $\Psi_{L_1L_2}(S_3)$ (i) includes the vertical line only if $p \in S_3$, and (ii) includes the horizontal line only if $q \in S_2$. As t_3 is restricted to $[0,1]$, t_1 is restricted to $\mathcal{T} = \{t_1(t_3) \,|\, t_3 \in [0,1]\}$. The horizontal line is not defined for values of $t_1 \notin \mathcal{T}$. Let $t' = \min(t_1(0), t_1(1))$ and $t'' = \max(t_1(0), t_1(1))$, where $t_1(0) = (dc - af)/f$ and $t_1(1) = (dc + a - fa - d)/(f-1)$. $t_1(t_3)$ is a hyperbola with a vertical asymptote at $t_3 = f$. If $f \in [0,1]$, then $\mathcal{T} = (-\infty, t'] \cup [t'', \infty)$. Otherwise, $\mathcal{T} = [t', t'']$. For symmetry reasons this case essentially also covers the case $\mathbf{b \neq 0,\ d = 0}$, where the characters of the vertical and horizontal lines exchange.

$\mathbf{b = 0,\ d = 0,}$ **and** $\mathbf{c \neq f}$**:** L_1 and L_2 are pairwise skew and intersect L_3 at $p = (0, 0, c)$ and $q = (0, 0, f)$, respectively. $\Psi_{L_1L_2}(L_3)$ consists of a vertical line at $t_1 = -a$ and a horizontal line at $t_2 = -e$. $\Psi_{L_1L_2}(S_3)$ includes the vertical and horizontal lines if S_3 contains $p = (0, 0, c)$ and $q = (0, 0, f)$, respectively.

$\mathbf{b = 0,\ d \neq 0,}$ **and** $\mathbf{c = f}$**:** L_1 and L_2 intersect at $p = (d, 0, c) = (d, 0, f)$. L_1 and L_3 intersect at $q = (0, 0, c) = (0, 0, f)$. $\Psi_{L_1L_2}(L_3)$ consists of a vertical line at $t_1 = -a$ and a horizontal line at $t_2 = -e$. The latter is included in $\Psi_{L_1L_2}(S_3)$ if $q \in S_3$. $t_1 = -a$ corresponds to all lines

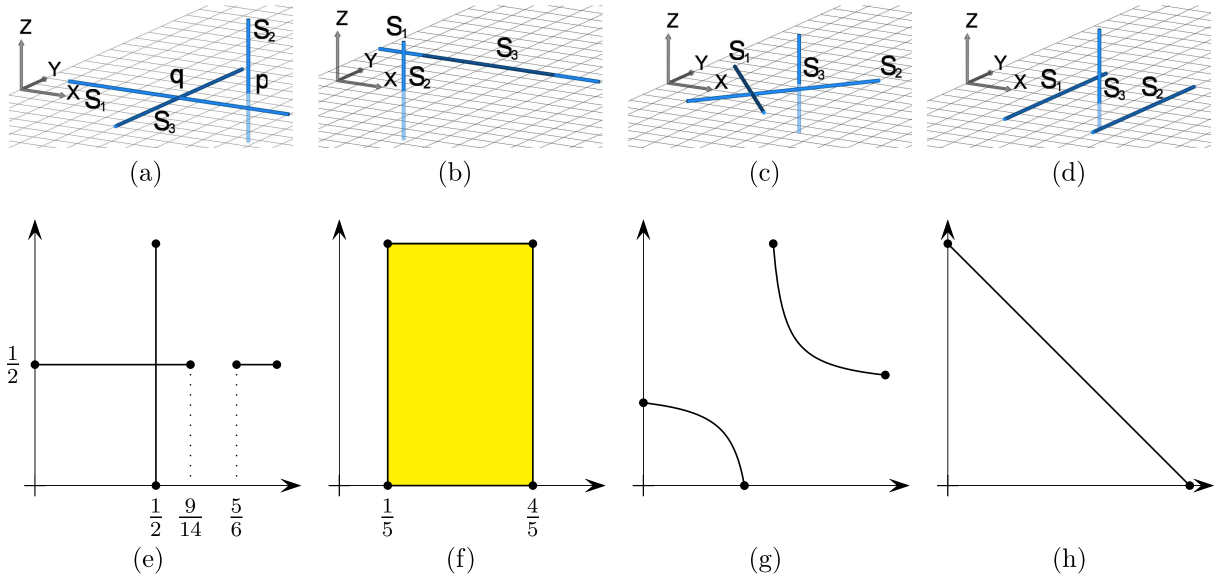


Figure 3.2: Mappings of three line segments, S_1, S_2, and S_3, in various configurations. The bottom figures depict the corresponding mapping $\Psi_{S_1S_2}(S_3)$. A point inside the yellow faces represents a line tangent to the three line segments. (a) S_1 and S_2 are skew, and S_1 and S_3 intersect at q. (e) $\Psi_{S_1S_2}(S_3)$ consists of two collinear horizontal line segments and one vertical line segment. (b) S_1 and S_2 are skew, and S_1 and S_3 overlap. (f) $\Psi_{S_1S_2}(S_3)$ is an axis parallel rectangle. (c) S_1 and S_2 are coplanar, and S_3 intersects the plane that contains S_1 and S_2 at a point. (g) $\Psi_{S_1S_2}(S_3)$ consists of two hyperbolic arcs. (d) S_1 and S_2 are parallel, and S_3 intersects the plane that contains S_1 and S_2 at a point. (h) $\Psi_{S_1S_2}(S_3)$ consists of a single line segment.

containing q and intersecting L_2. All points on $t_2 = -e$ correspond to L_1. For symmetry reasons this essentially also covers the case $\mathbf{b \neq 0,\ d = 0}$.

$\mathbf{b = 0,\ d = 0,}$ **and** $\mathbf{c = f}$**:** The three lines are concurrent at $p = (0, 0, c) = (0, 0, f)$. $\Psi_{L_1L_2}(L_3) = \{(-a, -e)\}$. $(-a, -e)$ represents all lines that contain p.

3.2 Directions Are Not Linearly Independent

3.2.1 Directions of L_1 and L_2 are Linearly Independent

We consider the case where L_1 and L_2 are linearly independent. Thus, we can assume that $d_1 = e_1,\ d_2 = e_2,\ d_3 = (u, v, 0),\ p_1 = (a, b, c),\ p_2 = (d, e, f)$ and $p_3 = \mathcal{O}$.

$\mathbf{c \neq 0,\ f \neq 0}$ **and** $\mathbf{c \neq f}$**:** The three lines are skew. $\Psi_{L_1L_2}(L_3)$ is the line $t_2 = (-fvt_1 - cue + ufb + cdv - fav))/(cu)$. $\Psi_{L_1L_2}(S_3)$ is a segment defined between $t_1 = -a + cd/f$ and $t_1 = u - a + c(d - u)/f$. Note that this also covers the case $v = 0$ (L_3 parallel to L_1) for which the line becomes horizontal. For $u = 0$ (L_3 parallel to L_2) $\Psi_{L_1L_2}(S_3)$ is the vertical line segment $t_1 = -a + cd/f$, define between $t_2 = -e + bf/c$ and $t_2 = v - e + f(b - v)/c$.

$\mathbf{c \neq 0,\ f \neq 0}$ **and** $\mathbf{c = f}$**:** L_1 and L_2 intersect at $p = (d, e, c) = (d, e, f)$. Since L_3 does not intersect the plane spanned by L_1 and L_2, $\Psi_{L_1L_2}(L_3)$ consists only of the point $(d - a, b - e)$ representing all lines containing p and L_3.

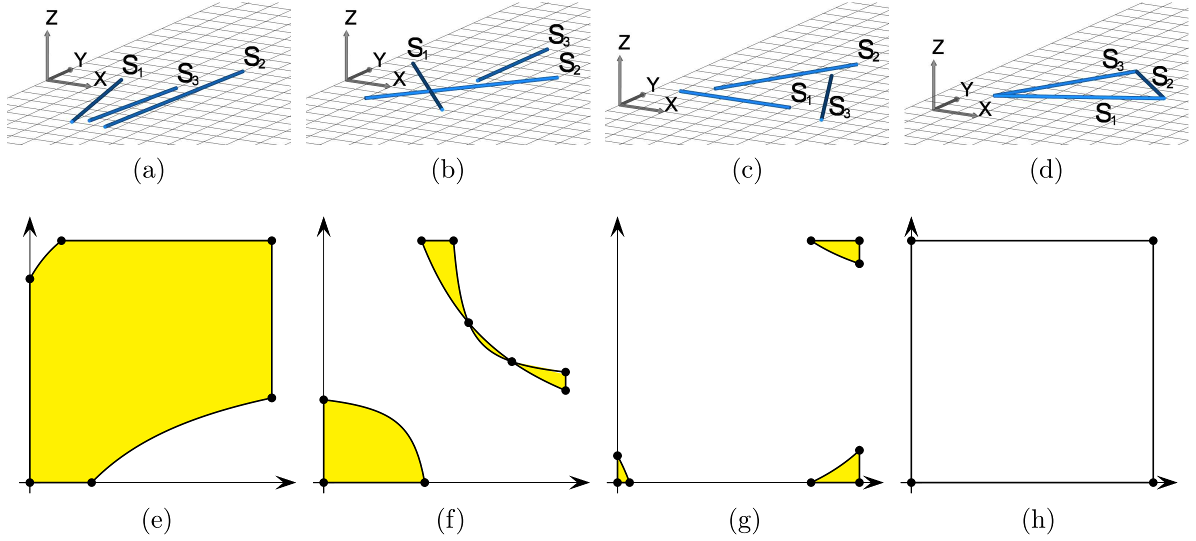


Figure 3.3: Mappings of three coplanar line segments, S_1, S_2, and S_3 in various configurations. The bottom figures depict the corresponding mapping $\Psi_{S_1 S_2}(S_3)$. A point inside the yellow face represents a line tangent to the three line segments. (a) S_1, S_2, and S_3 are pairwise disjoint. (b) S_1 and S_2 intersect, and S_3 is disjoint from both. (c) S_1, S_2, and S_3 are pairwise disjoint. (d) S_1, S_2, and S_3 form a triangle. (e) $\Psi_{S_1 S_2}(S_3)$ consists of a single connected component. (f) $\Psi_{S_1 S_2}(S_3)$ consists of four interior disjoint components. (g) $\Psi_{S_1 S_2}(S_3)$ consists of three disconnected components. (h) $\Psi_{S_1 S_2}(S_3)$ consists of the boundary of the unit square.

$\mathbf{c = 0,\ f \neq 0}$ **and** $\mathbf{c \neq f}$**:**

- $\mathbf{v \neq 0}$: L_1 intersects L_3 at $p = (bu/v, b, 0)$. $\Psi_{L_1 L_2}(S_3)$ is the vertical line $t_1 = -a + bu/v$ if $p \in S_3$ or empty otherwise.
- $\mathbf{v = 0 \wedge b \neq 0}$: L_1 is parallel to L_3 and since L_2 does not intersect the plane spanned by L_1 and L_2; it is obvious that $\Psi_{L_1 L_2}(L_3) = \emptyset$.
- $\mathbf{v = 0 \wedge b = 0}$: S_3 overlaps with L_1; thus $\Psi_{L_1 L_2}(S_3)$ is a two dimensional point set, namely, the vertical slab with t_1 between $-a$ and $u - a$.

This essentially also covers case $c \neq 0,\ f = 0$ for symmetry reasons.

$\mathbf{c = f = 0}$**:** L_3 is contained in the plane spanned by L_1 and L_2, thus $\Psi_{L_1 L_2}(L_3) = \mathbb{R}^2 \setminus \ell$, where ℓ is the line representing those lines that are parallel to L_3. However, $\Psi_{L_1 L_2}(S_3)$ is a bit more complex. $\Psi_{L_1 L_2}(L_3(t_3))$ is the rectangular hyperbola with additional parameter t_3

$$t_2(t_1, t_3) = \frac{-((e - t_3 v)t_1 + (bu - eu + dv - av)t_3 + ea - db)}{(t_1 + a - t_3 u)}.$$

This family has two fixed points that do not depend on t_3 as follows: $p' = (d - a, b - e)$, which represents the intersection of L_1 and L_2, and $p'' = (-a + bu/v, -e + dv/u)$, which corresponds to the line L_3. Since two such hyperbolas can only intersect in at most two points, we can conclude that the two-dimensional region $\Psi_{L_1 L_2}(L_3([0, 1] = S_3))$ is bounded by the hyperbolas $\Psi_{L_1 L_2}(L_3(0))$ and $\Psi_{L_1 L_2}(L_3(1))$; see also Figure 3.3. In the case $u = 0$ ($v = 0$) p'' is at infinity, since all hyperbolas in the family have the same vertical (horizontal) asymptote.

3.2.2 Directions of L_1 and L_2 Are Dependent

We consider the case where L_1 and L_2 are linearly dependent. Thus, we can assume that $d_1 = (1, 0, 0)$, $d_2 = (u, 0, 0) \neq \mathcal{O}$, $d_3 = (0, 1, 0)$, $p_1 = (a, b, c)$, $p_2 = (d, e, f)$ and $p_3 = \mathcal{O}$.

$\mathbf{c \neq 0}$, $\mathbf{f \neq 0}$**:** L_1 and L_2 are parallel and do not intersect L_3. L_3 intersects the plane spanned by L_1 and L_2 in $p = (0, (ce - fb)/(c - f), 0)$. $\Psi_{L_1L_2}(L_3)$ is the line $t_2 = (fa + ft_1 - cd)/(cu)$. $\Psi_{L_1L_2}(S_3) = \emptyset$ iff $p \notin S_3$.

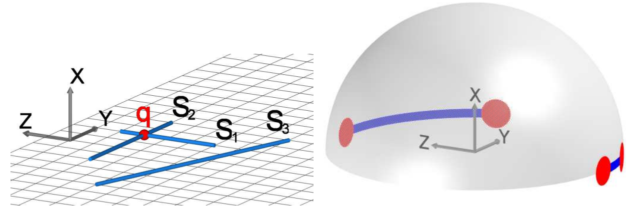
$\mathbf{c = 0}$, $\mathbf{f \neq 0}$**:** L_1 and L_2 are parallel. L_1 intersects L_3 at $p = (0, b, 0)$. $\Psi_{L_1L_2}(L_3)$ is the vertical line $t_1 = -a$. $\Psi_{L_1L_2}(S_3) = \emptyset$ iff $p \notin S_3$.

$\mathbf{c \neq 0}$, $\mathbf{f = 0}$**:** L_1 and L_2 are parallel. L_2 intersects L_3 at $p = (0, e, 0)$. $\Psi_{L_1L_2}(L_3)$ is the horizontal line $t_2 = -d/u$. $\Psi_{L_1L_2}(S_3) = \emptyset$ iff $p \notin S_3$.

$\mathbf{c = 0}$, $\mathbf{f = 0}$**:** L_1 and L_2 are parallel and intersect L_3 at $p' = (0, b, 0)$ and $p'' = (0, e, 0)$. Thus $\Psi_{L_1L_2}(L_3) = \mathbb{R}^2 \setminus \ell$, where ℓ is the line representing those lines that are parallel to L_3. This case is similar to the one in Subsection 3.2.1. $\Psi_{L_1L_2}(L_3(t_3))$ is a line $u(b - t_3)t_2 = (-at_3 - t_1t_3 + dt_3 - db + et_1 + ea)$. The family has a fixed point $p = (-a, -d/u)$, which corresponds to the line L_3. $\Psi_{L_1L_2}(L_3([0, 1]))$ is a wedge that is bounded by the lines $\Psi_{L_1L_2}(L_3(0))$ and $\Psi_{L_1L_2}(L_3(1))$.

3.2.3 S_1 and S_2 Intersect

Assume L_1 and L_2 intersect, and let $q = L_1(\tilde{t_1}) = L_2(\tilde{t_2})$ be the intersection point. The point $(\tilde{t_1}, \tilde{t_2})$ represents all lines that contain q. We represent these lines by points on a semi open upper hemisphere centered at q. We define the additional map $\Xi_q : \mathbb{R}^3 \setminus \{q\} \to \mathbb{H}^2$ and $\Xi_q(p) \longmapsto d = s(p - q)/|p - q|$, with $s \in \{\pm 1\}$, such that $d \in \mathbb{H}^2 = \{p \,|\, p \in \mathbb{S}^2$ and p is lexicographically larger than $\mathcal{O}\}$.



In the generic case a segment S maps to one or two geodesic arcs on $\mathbb{H}^2$. If S_3 is a point, or $L(S_3)$ contains q and S_3 does not, $\Xi_q(S)$ consists of a single point. If $q \in S_3$, we define $\Xi_q(S_3) = \mathbb{H}^2$. The left image of the figure depicts three line segments, S_1, S_2, and S_3, such that S_1 and S_2 intersect at q (and S_3 does not). The right image depicts the mapping $\Xi_q(S_3)$, where $\Xi_q(S_3) = \{\Xi_q(p) \,|\, p \in S_3\}$. It consists of two geodesic arcs on $\mathbb{H}^2$.

3.2.4 S_1 and S_2 Are Collinear

The case where S_1 and S_2 are collinear completes the list of possible cases. If S_1 and S_2 do not overlap, the only line that can possibly intersect S_1 and S_2 is the line containing S_1 and S_2. Otherwise, the number of degrees of freedom of all the lines that intersect S_1 and S_2 is three. The handling does not involve a mapping to a two-dimensional surface, as explained in the next chapter.

Corollary 3.1. $\Psi_{S_1 S_2}(S_3) \subset \mathbb{R}^2$ *is either a point, a one-dimensional set consisting of line segments or arcs of rectangular hyperbolas with horizontal and vertical asymptotes, or a two-dimensional set bounded by linear segments or arcs of such hyperbolas.*

We are now ready to describe our algorithm for solving the LTS problem in its full generality.

4

Algorithmic and Implementation Details

In this chapter we give a detailed description of the algorithm that solves the LTS problem. In section 4.1 we describe the algorithm and how it handles all of the degenerate cases. In Section 4.2 we describe some aspects of the implementation related to Cgal.

4.1 Algorithm Overview

The input is a set $\mathcal{S} = \{S_1, \ldots, S_n\}$ of n line segments in $\mathbb{R}^3$. In general an input line segment imposes an intersection constraint. In our implementation we assume that every input line segment imposes exactly a single constraint. It is possible to alter the implementation to follow the assumption that a sub-segment that is the intersection of multiple overlapping line segments imposes a single constraint, and a point that is either the intersection of multiple line segments, or simply a degenerate line segment, imposes two constraints. The output is a set of at most $O(n^4)$ (one-dimensional) lines or (two-dimensional) ruled surface patches in $\mathbb{R}^3$, such that each line abides by exactly four intersection constraints imposed by the line segments in $\mathcal{S}$, and all lines of each ruled surface patch abide by exactly four such intersection constraints. The line segments that impose the constraints of an output element are referred to as the *generating line segments* of that element. The generating line segments of every output surface patch or line are provided as part of the output. An element of the output is thus a pair of (i) a line or a surface patch and (ii) a quadruple of generating line segments.

Next, we describe the complete algorithm, which handles all degenerate cases, e. g., line segments that degenerate to points, concurrent line segments, and collinear, including overlapping, line segments. We transform the original three-dimensional LTS problem into a collection of two-dimensional problems and use two-dimensional arrangements to solve them, exploiting the plane-sweep algorithmic framework, which is output sensitive.

We set the line segments that degenerate to points apart from the rest. Let $\mathcal{S}'$ denote the

subset of such degenerate line segments, and $\mathcal{S}''$ denote the full-dimensional line segments, $\mathcal{S} = \mathcal{S}' \cup \mathcal{S}''$. The entire algorithm consists of three phases. During the first phase we go over all points in $\mathcal{S}'$. For each point $S'_i \in \mathcal{S}'$ we find every line that contains S'_i, possibly contains other points in $\mathcal{S}'$, and possibly intersects line segments in $\mathcal{S}''$, such that the total number of imposing constraints is at least four. During the second phase, we go over pairs of the full-dimensional line segments in $\mathcal{S}''$ that are not collinear. For each pair, (S_i, S_j), $S_i, S_j \in \mathcal{S}''$, we find lines that intersect S_i and S_j and other line segments in $\mathcal{S}''$ that have not been found yet in previous iterations, such that the total number of imposing constraints is at least four. These constraints, however, might be imposed by collinear input line segments. The processing of groups of four or more collinear line segments is deferred to the third phase. Such groups can easily be detected by lexicographically sorting the line segments by their normalized Plücker coordinates. W. l. o. g. assume that $\mathcal{S}$ consists of $\mathcal{S}''$ sorted accordingly preceded by $\mathcal{S}'$. Also, assume that $\mathcal{S}'$ consists of m points; see Algorithm 1 for pseudo code.

Algorithm 1 Compute lines that intersect line segments in $\mathcal{S} = \{S_1, \ldots, S_n\}$.

Phase I

1.1	**for** $i = 1, \ldots, m$,
1.2.1	Construct the arrangement $\mathcal{A}^s_{S_i}$ induced by $\{\Xi_{S_i}(S_k) \mid k = i+1, \ldots, n\}$.
1.2.2	Extract lines that intersect S_i from $\mathcal{A}^s_{S_i}$.

Phase II

2.1	**for** $i = m+1, \ldots, n-3$,
2.2	**for** $j = n, \ldots, i+3$,
2.3	**if** S_i and S_j are collinear, **break**.
2.5.1	Construct the arrangement $\mathcal{A}_{S_i S_j}$ induced by $\{\Psi_{S_i S_j}(S_k) \mid k = i+1, \ldots, j-1\}$.
2.5.2	Extract lines that intersect S_i and S_j from $\mathcal{A}_{S_i S_j}$.
2.6	**if** S_i and S_j intersect,
2.7.1	Construct the arrangement $\mathcal{A}^s_{S_i \cap S_j}$ induced by $\{\Xi_{S_i \cap S_j}(S_k) \mid k = i+1, \ldots, j-1\}$.
2.7.2	Extract lines that intersect S_i and S_j from $\mathcal{A}^s_{S_i \cap S_j}$.

Phase III

3.1	$i \leftarrow m+1$.
3.2	**while** $i \leq n-3$,
3.3	$j \leftarrow i+1$.
3.4	**while** $j \leq n$,
3.5	**if** S_i and S_j are not collinear,
3.6	**if** $i+3 < j$,
3.7	Process the set of collinear line segments $S_i, \ldots, S_{j-1}$.
3.8	$i \leftarrow j$.
3.9	**break**.
3.10	$j \leftarrow j+1$.
3.11	**if** $i+3 < j$,
3.12	Process the set of collinear line segments $S_i, \ldots, S_{j-1}$.
3.13	$i \leftarrow j$.

We defer the description of Phase I to Section 4.1.2, and proceed with the description of

Phase II.

4.1.1 The Processing of Arrangements in the Plane

In Step 2.5.1 of Algorithm 1 we construct the arrangement $\mathcal{A}_{S_iS_j}$ induced by the set $\mathcal{C}_{ij} = \{\Psi_{S_iS_j}(S_k) \,|\, k = i+1,\ldots,j-1\}$. Notice that S_i and S_j are not collinear. We distinguish between two disjoint subsets $\mathcal{C}'_{ij}$ and $\mathcal{C}''_{ij}$ of $\mathcal{C}_{ij}$, such that $\mathcal{C}'_{ij}$ consists of one-dimensional hyperbolic arcs restricted to the unit square (see Chapter 3) and $\mathcal{C}''_{ij}$ consists of a two-dimensional regions bounded by hyperbolic arcs and the unit square. (Recall that $\Psi_{S_iS_j}(S_k)$ consists of two-dimensional regions if either S_i or S_j overlap with S_k, or S_i, S_j, and S_k are coplanar; see Chapter 3.)

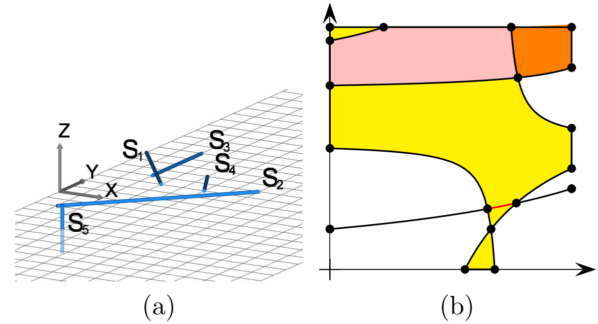


Figure 4.1: (a) Five line segments, $S_1,\ldots,S_5$, such that the first four are coplanar, and S_5 pierces the plane containing the first four. (b) The arrangement $A_{S_1S_2}$. The pink face represents a ruled surface patch the lines of which intersect the four coplanar line segments. The red curve represents a ruled surface patch the lines of which intersect S_1, S_2, S_3, and S_5.

We process the line segments $S_{i+1},\ldots,S_{j-1}$ one at a time to produce the inducing point sets $\mathcal{C}'_{ij}$ and $\mathcal{C}''_{ij}$. Next, using a plane-sweep algorithm, we construct the arrangement $\mathcal{A}'_{S_iS_j}$ induced by $\mathcal{C}'_{ij}$. The processing of $\mathcal{C}''_{ij}$ is different, because $\mathcal{C}''_{ij}$ consists of two-dimensional regions. For each maximal two-dimensional point set $\mathcal{R}_k \in \mathcal{C}''_{ij}$, $\mathcal{R}_k \subseteq \Psi_{S_iS_j}(S_k)$, we construct the arrangement $\mathcal{A}^k_{S_iS_j}$ induced by $\mathcal{R}_k$. Then, we construct $\mathcal{A}''_{S_iS_j}$ by overlaying all the arrangements in $\{\mathcal{A}^k_{S_iS_j} \,|\, \mathcal{R}_k \in \mathcal{C}''_{ij}\}$. The *2D Arrangements* package of CGAL supports an overlay operation, which computes the overlay of two given arrangements. In practice, we apply this operation several times to compute $\mathcal{A}''_{S_iS_j}$. Finally, we overlay the arrangements $\mathcal{A}'_{S_iS_j}$ and $\mathcal{A}''_{S_iS_j}$ to produce the final arrangement $\mathcal{A}_{S_iS_j}$; see Figure 4.1.

We store with each vertex and edge of the arrangement $\mathcal{A}'_{S_iS_j}$ the sorted sequence of input line segments that are mapped through $\Psi_{S_iS_j}$ to the points and curves that induce that cell. The segments are sorted by their indexes. Similarly, we store with each face of the arrangement $\mathcal{A}''_{S_iS_j}$ the sorted sequence of the line segments mapped through $\Psi_{S_iS_j}$ to the regions that induce that face. Each cell, i.e., vertex, edge, or face, of $\mathcal{A}_{S_iS_j}$ stores the sequence of the line segments obtained during the overlay from the corresponding cell in $\mathcal{A}'_{S_iS_j}$ and $\mathcal{A}''_{S_iS_j}$. We store only the minimal necessary set of line segments in every sequence. A member of a sequence of a vertex is a line segment that maps to a (zero-dimensional) point $p \in \mathcal{C}'_{ij}$. A member of a sequence of an edge is a line segment that maps to a (one-dimensional) curve $C \in \mathcal{C}'_{ij}$. A member of a sequence of a face is a line segment that maps to a (two-dimensional) region $R \in \mathcal{C}''_{ij}$. For example, consider a curve $C \in \mathcal{C}'_{ij}$, such that $C \subseteq \Psi_{S_iS_j}(S)$. The sequences of all edges induced by C contain S. However, the sequences of line segments of all vertices incident to these edges do not contain S, as this information is immediately accessible from the incident edges. Similarly, the sequences of all faces induced by a region $R \subseteq \Psi_{S_iS_j}(S)$ contain the line segment S, but the sequences of all vertices and edges incident to these faces do not. When a new face, f, is formed while

overlaying two arrangements, say $\mathcal{A}_1$ and $\mathcal{A}_2$, we merge the sequence of line segments of the two faces, f_1 and f_2 of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively, that induce f, and store the resulting sequence with f.

The generating line segments of every output element are immediately available from the sequences of line segments stored with vertices and edges. However, the role of these sequences extends beyond reporting. It turns out that some intersection points do not represent lines that intersect four line segments. An example of such a case occurs when either S_i or S_j intersects a third line segment, S_k. In such a case $\Psi_{S_iS_j}(S_k)$ consists of horizontal and vertical line segments; see Section 3.2.1. The intersection point of the vertical and horizontal line segments does not represent a line that intersects four line segments and, thus, must be ignored. This case is detected by examining the sorted sequences of line segments.

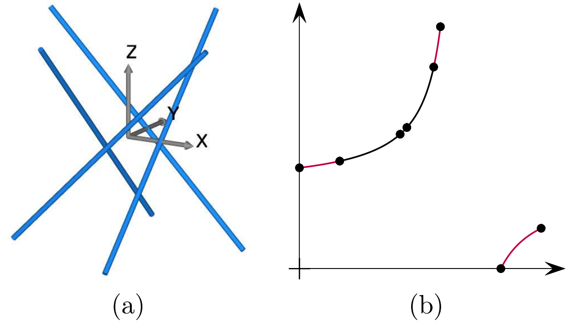


Figure 4.2: (a) Four line segments, S_1, S_2, S_3, S_4, supported by four lines of one ruling of a *hyperbolic paraboloid*, respectively; see also Figure 1.1a. (b) The arrangement $A_{S_1S_2}$. The edge drawn in purple is induced by two overlapping curves, one in $\Psi_{S_1S_2}(S_3)$ and the other in $\Psi_{S_1S_2}(S_4)$.

In Step 2.5.2 of Algorithm 1 we extract the information and provide it to the user in a usable format. We refer to an arrangement cell that represents a valid output element as an eligible cell. The eligibility of a given cell is immediately established from the sequence of line segments stored with that cell. We provide the user with the ability to iterate over eligible cells of different dimensions separately. This way, for example, a user can choose to obtain only the vertices that represent valid output lines. By default we consider a surface patch of the output represented by an edge or a face open. For example, consider an edge e that satisfies the output criteria, and let C denote its geometric embedding. The curve C is provided to the user as part of the iteration over the one-dimensional output elements. The two endpoints of C are provided to the user as part of the iteration over the zero-dimensional output elements. The user can override this setting, and choose to consider surface patches closed. In this case the iteration over the zero-dimensional output elements results with only eligible vertices that are not incident to eligible edges.

4.1.2 The Processing of Arrangements on the Sphere

Consider the case where two input line segments intersect at a point, say p. In this case we must output every line that contains p and abides by two additional intersection constraints. In Step 2.7.1 of Algorithm 1 we construct an arrangement on the sphere centered at p, the point of intersection between S_i and S_j. The arrangement is induced by the point set $\mathcal{C}^s_{ij} = \{\Xi_{S_i \cap S_j}(S_k) \,|\, k = i+1, \ldots, j-1\}$. We process the line segments $S_{i+1}, \ldots, S_{j-1}$ one at a time to produce the inducing set $\mathcal{C}^s_{ij}$. When the underlying line of a line segment S_k contains the sphere center, $\Xi_{S_i \cap S_j}(S_k)$ consists of a single point. For each k, $i < k < j$, $\Xi_{S_i \cap S_j}(S_k)$ consists of either an isolated point or at most two geodesic arcs on the sphere; see Section 3.2.3. The pairwise intersections of the points and arcs in $\mathcal{C}^s_{ij}$ represent lines that

are tangent to at least four input line segments. Next, using a plane-sweep algorithm on the sphere, we construct the arrangement $\mathcal{A}^s_{S_i \cap S_j}$ induced by $\mathcal{C}^s_{ij}$. When $\Xi_{S_i \cap S_j}(S_k)$ consists of a single point it induces a single vertex in the arrangement.

We maintain a set, $\mathcal{Q}_{ij}$, of all the line segments $S_k, i < k < j$ that contain p. If S_k contains the sphere center p, we insert S_k into $\mathcal{Q}_{ij}$ and proceed. Assume that $\mathcal{Q}_{ij}$ contains N line segments. If $2 \leq N$ then we account that the point p abides by at least four intersection constraints.

Similarly, in Step 1.2.1 of Algorithm 1, we construct the arrangement $\mathcal{A}^s_{S_i}$ on the sphere centered at S_i to account for every line that contains the point S_i and abides by at least three additional constraints. The arrangement is induced by the sets $\mathcal{C}'^s_i$ and $\mathcal{C}''^s_i$. We process the points $S_{i+1}, \ldots, S_m$ one at a time to produce the inducing set $\mathcal{C}'^s_i = \{\Xi_{S_i}(S_k) \,|\, k = i+1, \ldots, m\}$, which consists of points. We process the (full-dimensional) line segments $S_{m+1}, \ldots, S_n$ one at a time to produce the inducing set $\mathcal{C}''^s_i = \{\Xi_{S_i}(S_k) \,|\, k = m+1, \ldots, n\}$. As in the case above, we maintain a set, $\mathcal{Q}_i$, of all the line segments from $S_{m+1}, \ldots, S_n$ that contain the point S_i. If S_k contains the point S_i, we insert S_k into $\mathcal{Q}_i$ and proceed. When the underlying line of a line segment S_k contains the sphere center, $\Xi_{S_i}(S_k)$ consists of a single point. Next, using a sweep of the sphere algorithm, we construct the arrangement $\mathcal{A}^s_{S_i}$ induced by $\mathcal{C}^s_i = \mathcal{C}'^s_i \cup \mathcal{C}''^s_i$.

In both cases we extend the vertex and edge records. In the former case we store with each vertex and edge of the arrangement $\mathcal{A}^s_{S_i \cap S_j}$ the sorted sequence of line segments that are mapped through $\Xi_{S_i \cap S_j}$ to the points and geodesic arcs that induce that cell. In the latter case we store with each vertex and edge of the arrangement $\mathcal{A}^s_{S_i}$ the sorted sequence of input (zero and full-dimensional) line segments that are mapped through Ξ_{S_i} to the points and geodesic arcs that induce that cell.

As with the planar arrangements, we store only the minimal necessary set of line segments in every sequence. A member of a sequence of a vertex is a line segment that maps to a (zero-dimensional) point $p \in \mathcal{C}^s_{S_i \cap S_j}$. A member of a sequence of an edge is a line segment that maps to a (one-dimensional) geodesic arc $C \in \mathcal{C}^s_{S_i \cap S_j}$.

We extract the information from the arrangements on the sphere and provide it to the user in a usable format. All the settings that apply to the processing of the arrangements in the plane apply to the processing of the arrangements on the sphere as well; see Section 1. As with the aforementioned processing of the arrangements in the plane, we provide the user with the ability to iterate over eligible vertices and over eligible edges separately.

4.1.3 The Processing of Collinear Line Segments

Let $\mathcal{L} = \{S_1, ..., S_\ell\}$ be a set of $4 \leq \ell$ collinear line segments and let L be their underlying line. Assume $\mathcal{L}$ is processed in Step 3.7 or Step 3.12 of Algorithm 1. We generate $\binom{\ell}{4}$ output elements. Every output element consists of the line L and a different quadruple of line segments from $\mathcal{L}$.

We need to account for all lines that intersect sub-segments that are the intersections of at least four input line segments. We construct a one-dimensional arrangement $\mathcal{A}^\ell$, embedded in the line L, induced by the line segments in $\mathcal{L}$. We store with each vertex and edge

of $\mathcal{A}^\ell$ the inducing line segments of that cell. As with the aforementioned processing of two-dimensional arrangements, we provide the user with the ability to iterate over eligible vertices and over eligible edges separately.

4.1.4 Complexity Analysis

Setting apart the subset $\mathcal{S}'$ of line segments that degenerate to points is naturally done in linear time. Sorting the subset $\mathcal{S}''$ of full-dimensional line segments according to their normalized Plücker coordinates is done in $O(n \log n)$ time. Constructing the arrangement $\mathcal{A}^s_{S_i}$ for a given point $S_i \in \mathcal{S}'$ is done in $O((n+k_i)\log n)$ time using a plane-sweep algorithm, where k_i in the number of intersections of the inducing geodesic arcs. The total time it takes to construct all arrangements $\{\mathcal{A}^s_{S_i} \,|\, i = 1, \ldots, m\}$ is $O((mn + I_1)\log n)$, where I_1 is the total number of respective intersections. Extracting the output lines from all arrangements $\{\mathcal{A}^s_{S_i} \,|\, i = 1, \ldots, m\}$ takes $O((mn + I_1))$ time in total. Given two line segments $S_i, S_j \in \mathcal{S}''$, constructing the arrangement $\mathcal{A}'_{S_iS_j}$ is done in $O((n + k'_{ij})\log n)$ time using a plane-sweep algorithm, where k'_{ij} in the number of intersections of the inducing hyperbolic arcs. Constructing the arrangement $\mathcal{A}''_{S_iS_j}$ can also be done in $O((n + k''_{ij})\log n)$, where k''_{ij} in the number of intersections of the inducing hyperbolic arcs. Computing the overlay of $\mathcal{A}'_{S_iS_j}$ and $\mathcal{A}''_{S_iS_j}$ is done in $O((n + k_{ij})\log n)$ time, where k_{ij} in the number of intersections of the curves of the two arrangements. If S_i and S_j intersect, we also construct the arrangement $\mathcal{A}^S_{S_iS_j}$ in $O((n + k^s_{ij})\log n)$ time, where k^s_{ij} in the number of intersections of the geodesic arcs. Thus, the total time it takes to construct and process all arrangements in Phase II is $O((n^3 + I_2)\log n)$, where I_2 is the total number of respective intersections. The asymptotic resource-consumption of Phase III is negligible compared to those of the other phases. In summary, the process can be performed in $O((n^3 + I)\log n)$ running time. Where n is the input size and I is the output size. I is bounded by $O(n^4)$. As only one arrangement must be retained at a time, the required storage space is $O(n \log n + J)$, where J is the maximum number of intersections in a single arrangement. J is bounded by $O(n^2)$.

4.2 Implementation with Cgal

This section describes the packages of Cgal used in the implementation and the interface of the software for the computation of all lines tangent to four line segments taken from a set of n line segments. We assume in this section some familiarity of the reader with the C++ programming language [Str04] and the generic programming paradigm [Aus99].

Cgal follows the *generic programming paradigm* [Aus99], that is, algorithms are formulated and implemented such that they abstract away from the actual types, constructions, and predicates. Using the C++ programming language this is realized by means of class and function templates. Cgal's arrangement class template [FWH11] is parametrized by a *traits* class that handles the specific family of curves required by the application.

For the arrangement of geodesic arcs on the sphere we use the existing traits class [BFH+10]. As this only requires a linear kernel, it uses Cgal's efficient Lazy Kernel [BBP01]. However, in order to compute the planar arrangements of rectangular hyperbolic arcs with horizontal

and vertical asymptotes, CGAL offered only a general traits class for rational functions, which was introduced in [SHRH11]. The class uses the general univariate algebraic kernel [BHK11] of CGAL, which does not offer lazy constructions.

The aforementioned traits class is capable of representing rectangular hyperbolic arcs with horizontal and vertical asymptotes. However, since it was developed for general rational functions, the code is written assuming arbitrary degree in the numerator and denominator polynomials of the rational function. In our case the degree of both is just one. It follows that the degree of the polynomial, the roots of which represent the x-coordinates of intersection points of two hyperbolas, is at most 2. That is, the solution to these polynomials, and thus the coordinates of the intersection points, are numbers of algebraic degree 2. The *square-root extension* type of CGAL represents such a number as $a + b\sqrt{c}$, where $a, b, c \in \mathbb{Q}$. This explicit representation makes it possible to use the number type in conjunction with the lazy mechanism.

In order to benefit from the *square-root extension* type, we implemented a univariate algebraic kernel that is, similar to the one introduced in [dCCLT09], restricted to polynomials of degree 2. Enabling the use of CGAL's *square-root extensions* and in particular the use of lazy mechanism speeds up the computation significantly. In addition, we enhanced the implementation of the existing rational function traits, such that it is capable of using any algebraic kernel that complies with the requirements listed in [BHL+11]. This traits class, instantiated with the new algebraic kernel, uses lazy constructions and is thus able to handle hyperbolic arcs with horizontal and vertical asymptotes in a more efficient manner than the one presented in [SHRH11], as shown by experiments in Chapter 6 below.

4.3 Application Interface

Given a set $\mathcal{S} = \{S_1, S_2, ..., S_n\}$ of n line segments, *Lines Through Segments* is a new package that provides the means to find all the lines tangent to at least four line segments of $\mathcal{S}$. The class template `Lines_through_segments_3<Traits>` is a functor the parenthesis operator of which accepts as input the set $\mathcal{S}$ and returns a list of connected components of lines, each tangent to four line segments of $\mathcal{S}$. When instantiated, the template parameter `Traits` must be substituted with a model of the concept *LinesThroughSegmentsTraits_3*. The new concept defines the following types of geometric primitives.

- A model of the concept *GeodesicTraits_2*, which is a refinement of the concept *ArrangementTraits_2*, used to construct and maintain arrangements of geodesic arcs embedded on the sphere.
- A model of the concept *HyperbolicTraits_2*, which is a refinement of the concept *ArrangementTraits_2*, used to construct and maintain arrangements of hyperbolic arcs in the plane.

Every instance of the traits class template `Lines_through_segments_traits_3< Rational_kernel, Algebraic_kernel>` is a model of the concept *LinesThroughSegmentsTraits_3*; currently, it is the only model supplied with the package. The template parameters `Rational_kernel` and

`Algebraic_kernel` must be substituted by two exact geometric kernels when the class is instantiated. The first is a rational kernel, which is used for the input line segments, and the second is an algebraic kernel, which is used for the output objects.

4.3.1 Output Elements

Recall that we provide the user with the ability to iterate over eligible cells of different dimensions separately. We provide alternative option to the user to directly obtain the output elements; we describe the details for this alternative in this section.

Each element of the output is of type `Transversal`. The `Transversal` is defined as a BOOST variant [1] of the following types:

- `Line_3` — A rational line in three-dimensional Euclidean space.
- `Mapped_2` — A geometric object on a two-dimensional arrangement that represents a connected components of lines tangent to four input line segments. `Mapped_2` is implemented as a BOOST variant of the following types:
  - `Mapped_point_2` — The class contains an algebraic point on a planar arrangement $\mathcal{A}_{S_iS_j}$ and the two rational line segments S_i and S_j.
  - `Mapped_x_monotone_curve_2` — The class contains a hyperbolic arc on a planar arrangement $\mathcal{A}_{S_iS_j}$ and the two rational line segments S_i and S_j. Each point on the hyperbolic arc represents a tangent line to four input line segments.
  - `Mapped_general_polygon_2` — The class contains a face as a `General_polygon_2` on a planar arrangement $\mathcal{A}_{S_iS_j}$ and the two rational line segments S_i and S_j. Each point on the face represents a tangent line to four input line segments.
- `Through_3` — A geometric object in three-dimensional Euclidean space that all the lines that pass through this object are tangent to four input line segments. `Through_3` is implemented as BOOST variant of the following types:
  - `Through_point_3` — An intersection point of four or more concurrent line segments.
  - `Through_point_3_segment_3` — A pair of a point p and a line segments S. All the lines that pass through p and S pass through four input line segments.
  - `Through_segment_3` — A common rational line segment to four or more overlapping input line segments.

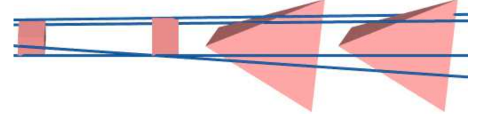
In some cases the four originating line segments that the transversal is tangent to are required. A compile-time flag is used to control the inclusion of the four originating line segments. If the flag is set, each element of the output is a pair of `Transversal` and an array of four input line segments.

Additional useful information for debugging is the arrangements that were constructed during the computation. The class template `Lines_through_segments_3_with_arrangements`

inherits from the class `Lines_through_segments_3` and stores these arrangements in a container. In this configuration the `Mapped_2` object is enhanced with a pointer to the arrangement that the mapped object lies on.

5

Lines Tangent to Four Strictly Disjoint Polytopes



We present a robust implementation of an efficient output-sensitive algorithm that solves the LTP problem. In particular, given a set $\mathcal{P} = \{P_1, ..., P_k\}$, of pairwise disjoint convex polytopes in $\mathbb{R}^3$, we find all lines tangent to four or more polytopes. Let L be an output line tangent to the polytopes $P_{i_1}, \ldots, P_{i_m}$, $4 \leq m$. Our implementation solves a variant of this problem where L may pierce $P_j, j \notin \{i_1, \ldots, i_m\}$. It is not hard to extend the implementation and filter out lines that pierce any input polyhedron. The figure to the right depicts two tetrahedra and two cubes and four lines, each tangent to the four of them.

5.1 Algorithm Overview

The algorithm that solves the LTP problem resembles Algorithm 1. We go over unordered pairs of polytopes. For each pair of polytopes (P_i, P_j), $i < j$, we go over all pairs of edges (e', e'') such that $e' \in E(P_i)$ and $e'' \in E(P_j)$, where $E(P)$ denotes the set of edges of P. For each pair of edges (e', e''), we go over the polytopes $P_{j+1}, \ldots, P_k$ and we compute the sets $\mathcal{C}_{e'e''P_\ell} = \{\Psi_{e'e''}(e) \,|\, e \in E(P_\ell)\}$, $\ell = j+1, \ldots, k$. Points in the set $\mathcal{C}_{e'e''P_\ell}$ represent lines that are tangent to e' and e''. These lines either intersect P_ℓ in its interior or are tangent to P_ℓ. Note that a single curve $c \in \mathcal{C}_{e'e''P_\ell}$ may comprise sub-curves of points of both types. Let $\mathcal{H}_{e'e''P_\ell}$ denote the set of all points that represent lines tangent to e', e'', and P_ℓ. We compute $\mathcal{H}_{e'e''P_\ell}$ from $\mathcal{C}_{e'e''P_\ell}$ for $\ell = j+1, \ldots, k$. Next, we construct the arrangement $\mathcal{A}_{e'e''P_\ell}$ induced by $\mathcal{H}_{e'e''P_\ell}$, $\ell = j+1, \ldots, k$, and eliminate cells associated with lines that intersect the interior of P_i or of P_j. Finally, we construct the arrangement $\mathcal{A}_{e'e''}$ by overlaying all the

arrangements $\mathcal{A}_{e'e''P_\ell}$, $\ell = j+1,\ldots,k$ and extract from the arrangement the lines tangent to P_i at e', P_j at e'', and at least two additional polytopes; see Algorithm 2 for pseudo code.

Algorithm 2 Find all lines tangent to four strictly disjoint convex polytopes in $\mathcal{P}$.

```
1   for i = 1,...,k−3,
2     for j = i+1,...,k−2,
3       foreach e' ∈ E(P_i), e'' ∈ E(P_j),
4         for ℓ = j+1,...,k,
5           Construct the set C_{e'e''P_ℓ}.
6           Construct the set H_{e'e''P_ℓ} from the set C_{e'e''P_ℓ}.
7           Construct the arrangement A_{e'e''P_ℓ} induced by H_{e'e''P_ℓ}.
8           Eliminate cells of A_{e'e''P_ℓ} associated with lines that intersect the interior
            of P_i or of P_j.
9         Construct the arrangement A_{e'e''} from the arrangements A_{e'e''P_ℓ}, ℓ =
          j+1,...,k.
10        Extract tangent lines from A_{e'e''}.
```

5.2 Constructing the Arrangement

In line 7 of Algorithm 2 we construct the arrangement induced by the sets $\mathcal{H}_{e'e''P_\ell}$, where $\mathcal{H}_{e'e''P_\ell}$ represents lines tangent to e', e'', and P_ℓ. In line 6 of Algorithm 2 we construct the set $\mathcal{H}_{e'e''P_\ell}$ from the set $\mathcal{C}_{e'e''P_\ell} = \{\Psi_{e'e''}(e) \,|\, e \in E(P_\ell)\}$. In this section we show that $\mathcal{H}_{e'e''P_\ell}$ is subset of $\mathcal{C}_{e'e''P_\ell}$ and describe how to compute it efficiently. Consider the arrangement induced by the point set $\mathcal{C}_{e'e''P_\ell}$. Each pair of adjacent edges (two edges are adjacent if they share a common vertex) in the arrangement are the images of two adjacent edges in P_ℓ; see Figure 5.1. In the following we omit the subscript letter ℓ in all notations for clarity.

Lemma 5.1. *Given a polytope P and two skew lines L_1 and L_2, the points in the set $\mathcal{C}_{L_1L_2P} = \{\Psi_{L_1L_2}(e) \,|\, e \in E(P)\}$ comprise at most three connected components. They represent lines divided into at most three connected components of lines respectively.*

Proof. Let Q_1 denote the plane that contains L_1 and does not intersect L_2. Since L_1 and L_2 are skew, such a plane exists. Similarly, let Q_2 denote the plane that contains L_2 and does not intersect L_1. As Q_1 and Q_2 are parallel, they divide $\mathbb{R}^3$ into three parts. None of the lines contained in Q_1 or in Q_2 are tangent to both

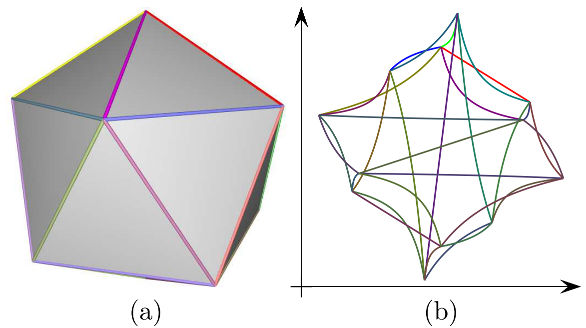


Figure 5.1: (a) An icosahedron I. (b) The arrangement induced by the point set $\{\Psi_{S_1S_2}(e) \,|\, e \in E(I)\}$, where S_1 and S_2 are two skew segments (omitted in the figure). The color of each edge of the arrangement is the same as the color of its generating edge.

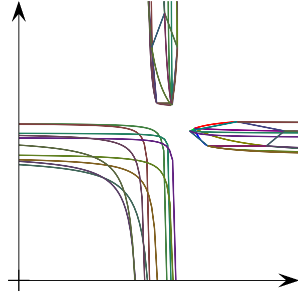


Figure 5.2: An arrangement induced by the point set $\{\Psi_{S_1S_2}(e) \,|\, e \in E(I)\}$, where S_1 and S_2 are two line segments carefully chosen, so that the points in the set $\{\Psi_{L(S_1)L(S_2)}(e) \,|\, e \in E(I)\}$ represent lines divided into three connected components of lines.

L_1 and L_2 and all other lines are. As P is convex, Q_1 and Q_2 may divide P into at most three parts. In particular, the intersection $P \cap \{Q_1 \cup Q_2\}$ consists of points that cannot lie on lines simultaneously tangent to L_1 and L_2. Moreover, a line tangent to L_1, L_2, and P at r_1 cannot be moved to a line tangent to L_1, L_2, and P at r_2, while holding the tangency constraints, if r_1 and r_2 are not in the same part. We conclude that, the set of lines tangent to L_1 and L_2, and intersect P, is divided into at most three connected components of lines; see Figure 5.2. □

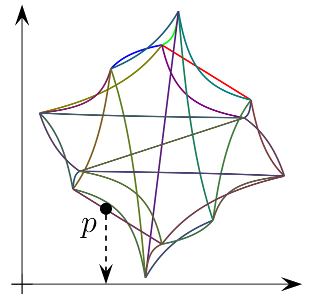
Lemma 5.2. *Let P be a convex polytope and let $\mathcal{C}_{L_1L_2P} = \{\Psi_{L_1L_2}(e) \,|\, e \in E(P)\}$ be the set of all planar points that represent lines that intersect P, and two skew lines L_1 and L_2. Let $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ be (possibly empty) maximal connected component, $\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 = \mathcal{C}_{L_1L_2P}$. For $i = 1, 2, 3$, the intersection of every vertical line with $\mathcal{C}_i$ is either a segment, a ray, or empty.*

Proof. Let V denote a vertical line in the plane. V represents the group $\mathcal{L}$ of all the lines that intersect L_1 at a point, say q, and the line L_2. Since P is convex, the intersection of P and the plane that contains q and L_2 is either a convex polygon or empty. If the line parallel to L_2 that contains q also intersects P, $\mathcal{L}$ is either empty or contains two connected components of lines. Thus, $\mathcal{L}$ contains zero, one, or two connected components of lines, which implies that the intersection of V with $\mathcal{C}_i$ is either a segment, a ray, or empty. □

Corollary 5.3. *The upper and lower boundaries of $\mathcal{C}_i$, $i = 1, 2, 3$, are x-monotone.*

Lemma 5.4. *Let L be a line tangent to two skew lines L_1 and L_2, and let p be a planar point that represents L. Let P be a convex polytope. Let $\mathcal{C}_{L_1L_2P} = \{\Psi_{L_1L_2}(e) \,|\, e \in E(P)\}$ be the set of all planar points that represent lines that intersect L_1, L_2, and P. Let $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ be (possibly empty) maximal connected component, $\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 = \mathcal{C}_{L_1L_2P}$. Finally, consider the point set $\mathcal{H}_{L_1L_2P}$, and let $\mathcal{H}_i \subset \mathcal{H}$ be the union of the lower and the upper envelopes, if exist, of $\mathcal{C}_i$, for $i = 1, 2, 3$. L is also tangent to P iff p lies on $\mathcal{H}_{i'}$ for some $1 \leq i' \leq 3$.*

Proof. All the points in the plane represent lines tangent to L_1 and L_2. A line tangent to P must intersect the closed segment associated with an edge of P. Thus, all the points in every face of the arrangement induced by $\mathcal{C}_{L_1L_2P}$ have the same invariant—all the lines they represent either intersect P or they do not.



($\Leftarrow$), W.l.o.g, let p be a point on the lower envelope of one of $\mathcal{C}_{i'}$ for some $1 \leq i' \leq 3$. Let V denote the vertical line in the plane that contains p. From Lemma 5.2 we get that V intersects $\mathcal{C}_{i'}$ at either

a segment or a ray. Since P is bounded, the face below p represents lines that do not intersect P (while the points above p represent lines that do intersect P). If V intersects additional point set say, $\mathcal{C}'_j$, $j' \neq i'$, then since $\mathcal{C}_{i'}$ and $\mathcal{C}_{j'}$ represent lines in different connected components, we come to the same conclusion; that is, the face below p represents lines that do not intersect P. Thus, L is tangent to P.

($\Rightarrow$), Let L be a line tangent to P, L_1, and L_2. Let q be the intersection point of L with L_1. Let Q be the plane defined by q and L_2. Q intersects P at a convex polygon. All the lines on Q that intersect both q and P are mapped to a line segment or to two rays on a vertical line, L is mapped to an endpoint p on the line segment or on one of the rays. Thus, p is on the boundary of some C_i. From Corollary 5.3 we know that the boundary is x-monotone, hence, $p \in \mathcal{H}_{i'}$ for some $1 \leq i' \leq 3$. □

Given two line segments, S_1 and S_2, and a polytope, P, let $\mathcal{H}_{S_1S_2P}$ be the set of all points in the plane that represent lines tangent to S_1, S_2, and P. Let $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}_3$ be (possibly empty) maximal connected component, $\mathcal{H}_1 \cup \mathcal{H}_2 \cup \mathcal{H}_3 = \mathcal{H}_{L(S_1)L(S_2)P}$. Consider the point set $\mathcal{C} = \{\Psi_{L(S_1)L(S_2)}(e) \,|\, e \in E(P)\}$, and let $\mathcal{C}_i \subset \mathcal{C}$ be the point set that contains $\mathcal{H}_i$, for $i = 1, 2, 3$. First, from Lemma 5.4 we get that the union of the envelopes of $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ yields $\mathcal{H}_{L(S_1)L(S_2)P}$, which represents lines tangent to $L(S_1)$, $L(S_2)$, and P. Secondly, the desired point set $\mathcal{H}_{S_1S_2P}$ is simply the result of clipping $\mathcal{H}_{L(S_1)L(S_2)P}$ to the unit square. Since, $\mathcal{H}_{L(S_1)L(S_2)P}$ consists of arcs of a rectangular hyperbola with horizontal and vertical asymptotes, clipping does not yield additional connected components.

First, we compute $\mathcal{C} = \{\Psi_{L(S_1)L(S_2)}(e) \,|\, e \in E(P)\}$. Next, we set apart the vertical lines in $\mathcal{C}$ to yield $\mathcal{C}'$. Such lines are generated in each case where an edge in $E(P)$ intersects $L(S_1)$; see Section 3.2.1. Next, we identify the connected component of every curve in $\mathcal{C}'$ as follows: We calculate the two parallel planes Q_1 and Q_2. For every curve $C \in \mathcal{C}'$ (that was mapped by an edge $e \in E(P)$) we pick a representative point $p \in C$, calculate the intersection point q of the line represented by p and the edge e, and locate p in one of the volumes induced by Q_1 and Q_2. This yields the identity of the connected component of the lines represented by C. Naturally, if neither Q_1 nor Q_2 intersect P, there is only one connected component, and there is no need to perform the point location at all. Moreover, if a curve C represents lines of a certain connected component, then all curves connected to C represent lines of the same connected component. We detect connectivity at endpoints to expedite the connected component identification. Next, we compute the upper and lower envelope of every connected component to obtain the boundary. Finally, we add the vertical lines back and clip the resulting curves to the unit square. It is possible to reverse the order of operations; that is, first clip the curves in $\mathcal{C}$ to the unit square to yield $\mathcal{C}^S$. Next, identify the connected component of the curves in $\mathcal{C}^S$. Next, compute the boundary of the connected components. The last step becomes more complicated, since faces bounded by the unit square may represent lines that intersect P. Thus, on the one hand it is necessary to identify the connected component of lines represented by these faces. On the other hand, clipping may substantially decrease the number of curves.

We collect the set $\mathcal{H}_{S_1S_2P}$ and insert it into the arrangement $\mathcal{A}_{S_1S_2P}$ using a plane-sweep algorithm.

5.3 Removing Cells From the Arrangement

The geometric embeddings of the edges of the arrangement $\mathcal{A}_{S_1S_2P}$ represent all lines tangent to S_1, S_2, and P. Some of these lines may intersect the interior of P_i or of P_j. Their representation must be removed from the arrangement. We construct two additional arrangements, $\mathcal{A}^i_{S_1S_2}$ and $\mathcal{A}^j_{S_1S_2}$. Each has (at most three) marked faces, such that all points inside these faces represent lines tangent to S_1 and S_2, and intersect P_i and P_j, respectively. We slightly modify the algorithm described in the previous section used to construct each of these two arrangements as follows: (i) To construct $\mathcal{A}^i_{S_1S_2}$ and $\mathcal{A}^j_{S_1S_2}$ we start with the point sets $\mathcal{C}^i_{L(S_1)L(S_2)} = \{\Psi_{L(S_1)L(S_2)}(e) \,|\, e \in E(P_i) \setminus \{S_1\}\}$ and $\mathcal{C}^j_{L(S_1)L(S_2)} = \{\Psi_{L(S_1)L(S_2)}(e) \,|\, e \in E(P_j) \setminus \{S_2\}\}$, respectively. (ii) We find the connected components $\mathcal{C}^i_1$, $\mathcal{C}^i_2$, and $\mathcal{C}^i_3$, $\mathcal{C}^i_1 \cup \mathcal{C}^i_2 \cup \mathcal{C}^i_3 = \mathcal{C}^i_{L(S_1)L(S_2)P}$ and $\mathcal{C}^j_1$, $\mathcal{C}^j_2$, and $\mathcal{C}^j_3$, $\mathcal{C}^j_1 \cup \mathcal{C}^j_2 \cup \mathcal{C}^j_3 = \mathcal{C}^j_{L(S_1)L(S_2)P}$. (iii) We compute the lower and upper envelopes of $\mathcal{C}^i_1$, $\mathcal{C}^i_2$, and $\mathcal{C}^i_3$ to obtain $\mathcal{H}^i_{L(S_1)L(S_2)}$. (iv) We compute the lower and upper envelopes of $\mathcal{C}^j_1$, $\mathcal{C}^j_2$, and $\mathcal{C}^j_3$ and to obtain $\mathcal{H}^j_{L(S_1)L(S_2)}$. (v) We construct the arrangements $\mathcal{A}^i_{S_1S_2}$ and $\mathcal{A}^j_{S_1S_2}$ induced by the point sets $\mathcal{H}^i_{L(S_1)L(S_2)}$ and $\mathcal{H}^j_{L(S_1)L(S_2)}$, respectively. (vi) We mark the faces of $\mathcal{A}^i_{S_1S_2}$ and $\mathcal{A}^j_{S_1S_2}$ that represent lines that intersect P_i and P_j, respectively. Then, we compute the union of the marked faces $\mathcal{A}^i_{S_1S_2}$ and $\mathcal{A}^j_{S_1S_2}$ via their overlay $\mathcal{A}'_{S_1S_2}$. Finally, in line 8, we compute the overlay of $\mathcal{A}_{S_1S_2P}$ and $\mathcal{A}'_{S_1S_2}$ and remove all edges in $\mathcal{A}_{S_1S_2P}$ that are overlaid with marked cells in $\mathcal{A}'_{S_1S_2}$.

Notice that each one of the two endpoints of S_1 is also the endpoint of at least two other edges of P_i. It implies that the set $\mathcal{C}^i_{S_1S_2}$ consists of vertical and horizontal lines. This case is adequately handled; see previous section. The same holds for $\mathcal{C}^j_{S_1S_2}$. This observation can be used to optimize line 8 of the algorithm, which is not implemented though.

We overlay the arrangements $A_{S_1S_2P_\ell}$, $\ell = j+1, \ldots, k$, and obtain the arrangement $A_{S_1S_2}$. We need to account for all lines that are tangent to S_1, S_2, and two additional polytopes. We store with each edge of $A_{S_1S_2}$ the inducing polytope of that edge. We only report on vertices that are the result of the intersection of two edges of distinct polytopes. Some of the vertices are the result of the intersection of two edges of the same polytope, there are $O(n)$ such vertices. Since we have only $O(n^2)$ arrangements, handling these vertices do not effect the asymptote complexity of the algorithm.

5.3.1 Degeneracies

In this section we go over degenerate cases. The code that handles these degeneracies has not been implemented yet, and is suggested as future work. Let S denote an edge of P_ℓ.

$L(S_1)$ **contains** S**.** In case $L(S_1)$ contains S, the point set $\{\Psi_{L(S_1)L(S_2)}(p) \,|\, p \in S\}$ is a bounded face between two vertical lines. This face does not intersect the unit square and hence it can be ignored.

$L(S_2)$ **contains** S**.** In case $L(S_2)$ contains S, the point set $\{\Psi_{L(S_1)L(S_2)}(p) \,|\, p \in S\}$ is a bounded face between two horizontal lines. This face does not intersect the unit square and

hence it can be ignored.

S_1 and S_2 are coplanar. When S_1 and S_2 are coplanar we can reduce the problem to the plane $P_{S_1S_2}$ that contains both S_1 and S_2. Each polytope intersect $P_{S_1S_2}$ at either an empty intersection, a vertex, an edge, a facet, or a convex polygon. When the intersection is either a facet, an edge or a vertex, $P_{S_1S_2}$ is tangent to the polytope. If the intersection is a vertex, a segment of a hyperbola is added to the arrangement; in case of an edge, a face is added to the arrangement. If the intersection is a facet, several faces are added to the arrangement, one for each edge. The union faces of all of these faces are added to the arrangement. The union can be found output-sensitively using envelopes. Since each point on these faces represents a line tangent to three polytopes, they are marked and a counter of the face is set to one. If two such faces overlap then the counter of the overlapping face is set to two. If the intersection is a convex polygon (that is not a face), no special treatment is needed. We compute $\mathcal{C} = \{\Psi_{L(S_1)L(S_2)}(e) \mid e \in E(P)\}$ and apply the procedure we described above for the general case.

5.4 Complexity Analysis

Computing $\mathcal{C} = \{\Psi_{L(S_i)L(S_j)}(e) \mid e \in E(P)\}$ is done in $O(n)$ time. Identifying the connected components take $O(n)$ time as well. Computing the upper and lower envelope of every connected components takes $O(\lambda_2(n) \log n)$ ($O(\lambda_2(n)) = 2n - 1$), since the curves in $\mathcal{C}'$ are all arcs of rectangular hyperbolas with vertical and horizontal asymptotes or horizontal lines and the maximal number of intersections between two such curves is 2. Collecting the set $\mathcal{H}_{S_iS_j}$ and inserting it into the arrangement $\mathcal{A}_{S_iS_j}$ using a plane-sweep algorithm takes $O(n \log n + I_1)$, where I_1 is the total number of respective intersections. Thus, the total time it takes to construct and process all arrangements is $O((n^3 + I_2) \log n)$, where I_2 is the total number of respective intersections. In summary, the process can be performed in $O((n^3 + I) \log n)$ running time. Where n is the input size and I is the output size. I is bounded by $O(n^4)$. As only one arrangement must be retained at a time, the required storage space is $O(n \log n + J)$, where J is the maximum number of intersections in a single arrangement. J is bounded by $O(n^2)$.

5.5 Implementation with CGAL.

We use the `Polyhedron_3` class of CGAL to represent an input polytope. The output is template of the same format as the format of the output of the `Lines_through_segments_3<Traits>` functor; see Section 4.3. However, since we do not handle all degenerate cases, the output will always be of type `Mapped_point_2`.

`Lines_through_polytopes_3` is a new class template that provides the means to find all lines tangent to four polytopes. The class `Lines_through_polytopes_3<Traits>` is a functor that accepts as input a set $\mathcal{P}$ of k convex polytopes and returns a list of connected components, each tangent to four polytopes of $\mathcal{P}$. When instantiated, the template parameter

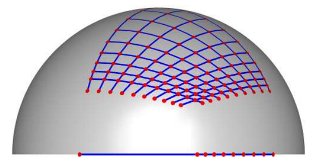
`Traits` must be substituted with model of the concept *LinesThroughSegments_3*, which is described in Section 4.2. The new concept lists the set of requirements that must be fulfilled by an instance of the traits template-parameter of the class `Lines_through_polytopes_3`.

The class uses the *Envelopes of Surfaces in 2D* package of CGAL, in order to compute the lower and upper envelopes of each component.

6
Experiments

In this chapter we report on several experiments that show the strength of the algorithm and its implementation. We have conducted experiments on three types of data sets. The first produces the worst-case combinatorial output and has many degeneracies. The second consists of transformed versions of the first and has many near-degeneracies. The third comprises random input. We report on the time consumption of our implementation, and compare it to those of other implementations. All experiments were performed on a Pentium PC clocked at 2.40 GHz.

6.1 Grid



The Grid data set comprises 40 line segments arranged in two grids of 20 lines segments each lying in two planes, P_1 and P_2, parallel to the yz-plane; see Chapter 1. Each grid consists of ten vertical and ten horizontal line segments. The output consists of several planar patches each lying in P_1 or P_2 and exactly $10,000$ lines, such that each contains one intersection point in one plane and one in the other plane. Table 6.1 lists all output elements. 703 arrangements in the plane and 200 arrangements on the sphere were constructed during the process. All single output lines are represented by vertices of arrangements on the sphere. Such an arrangement is depicted in the figure above. The origin of the sphere is the intersection point of two line segments S_1 and S_{40} in the same plane. The arrangement is induced by the point set $\{\Xi_{S_1 \cap S_{40}}(S_i) \,|\, i = 2, \ldots, 39\}$. The figure on the next page depicts an arrangement in the plane constructed during the process. The arrangement is induced by the point set $\{\Psi_{S_1 S_{40}}(S_i) \,|\, i = 2, \ldots, 39\}$. Each face of the arrangement represents a ruled surface patch, such that each line lying in the surface intersects at least 6 and up to 20 line segments. Different colors represent different number of originating line segments.

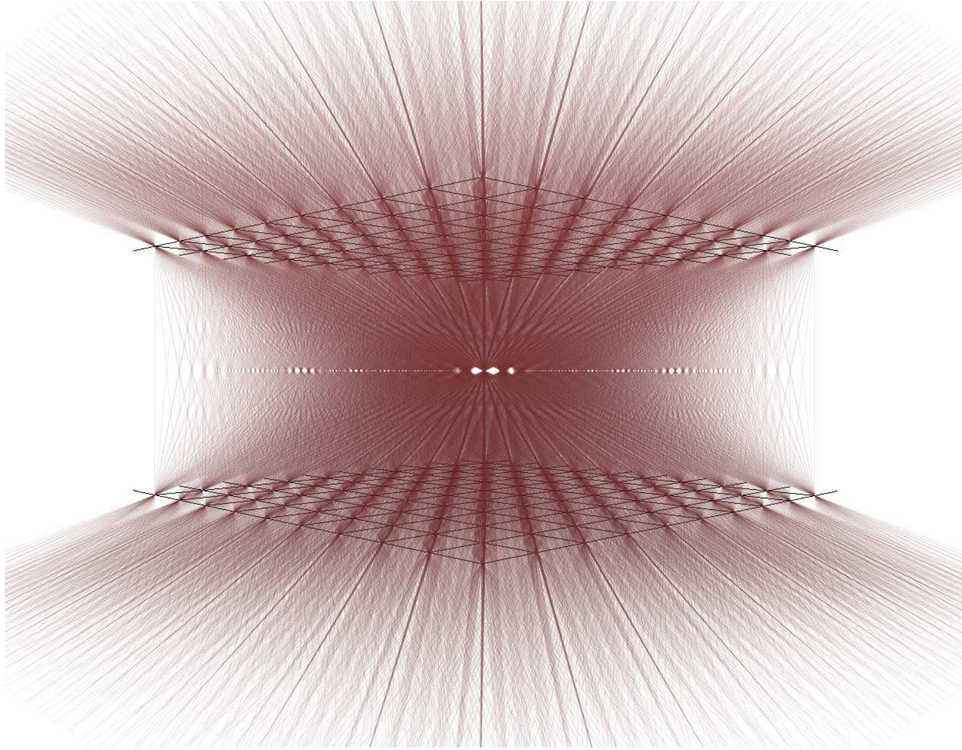
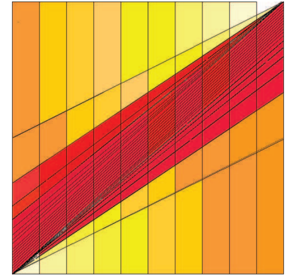


Figure 6.1: Two grids parallel to the $z = 0$ plane.

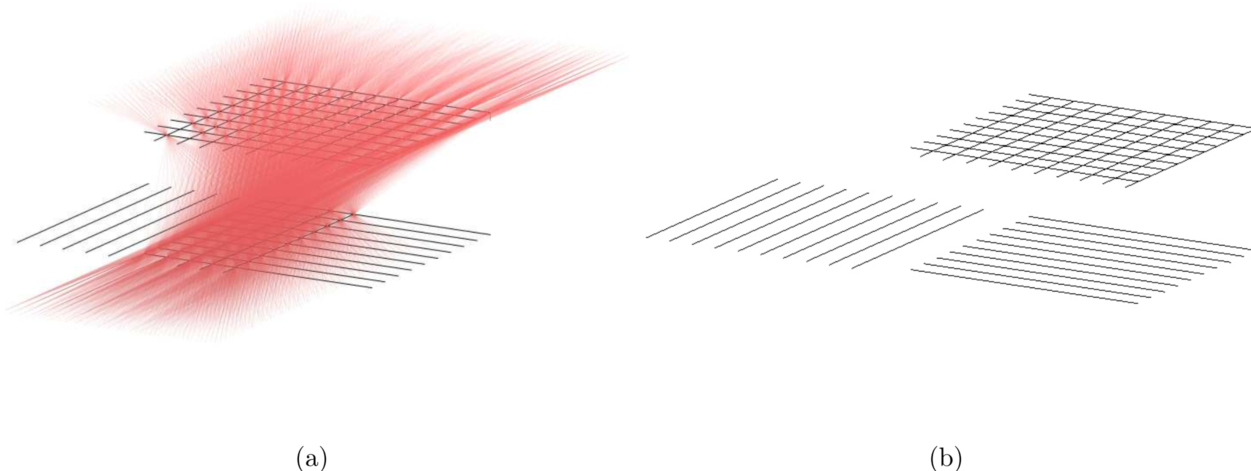
Table 6.1: Grid Output

Number of lines	Number of curves	Number of arcs	Number of general polygons	Time (in seconds)
10,000	36	1,224	17,060	20.74

6.1.1 Transformed Grid



We conducted three additional experiments using a transformed version of the Grid data set. First, we slightly perturbed the input line segments, such that every two line segments became skew and the directions of every three line segments became linearly independent. We refer to this input as **Perturbed Grid**. Secondly, we translated the (perturbed) horizontal line segments of one grid along the plane that contains this grid (referred to as **Perturbed Grid 1**), increasing the distance between the (perturbed) vertical and horizontal line segments of that grid. This drastically reduced the number of output lines. Thirdly, we translated the (perturbed) horizontal line segments of that grid even more along the plane (referred to as **Perturbed Grid 2**), further reducing the number of output lines; see Figure 6.2. Table 6.2 shows the number of output lines and the time it took to perform the computation using our implementation, referred to as **LTS**. The monotonic relation between the output size and time consumption of our implementation is prominent. The table also shows the time



(a) (b)

Figure 6.2: Translation of the horizontal lines at the lower grid.

it took to perform the computation using two instances of a program developed by J. Redburn [Red03], which represents lines by their Plücker coordinates and exhaustively examines every quadruple of input line segments. One instance, relies on a number type with unlimited precision, while the other resorts to double-precision floating-point numbers. As expected, when limited precision numbers were used, the output was only an approximation. Notice that the influence of the output size on the time consumption of Redburn's implementation is negligible.[1]

Table 6.2: Perturbed Grid. Time is measured in seconds.

Input	Unlimited Precision			Double Precision	
	Time		Lines	Time	Lines
	LTS	Redburn		Redburn	
Perturbed Grid	23.72	140.17	12,139	0.70	12,009
Translated Grid 1	11.83	132.80	5,923	0.69	5,927
Translated Grid 2	6.90	128.80	1,350	0.70	1,253

6.2 Random Input

The Random data set consists of 50 line segments drawn uniformly at random. In particular, the endpoints are selected uniformly at random within a sphere. We experimented with three different radii, namely, **Short**, **Medium**, and **Long** listed in increasing lengths. We verified that the line segments are in general position; that is, the directions of every three are linearly independent and they are pairwise skew. Table 6.3 shows the number of

[1] Redburn's implementation does not handle well degenerate input. Thus, we were unable to experiment with the original Grid data set using this implementation.

output lines and the time it took to perform the computation using (i) our implementation referred to as **LTS**, (ii) our implementation enhanced with the lazy mechanism referred to as **LLTS** (see Section 4.2), and (iii) the instance of Redburn's implementation that relies on unlimited precision. Once again, one can clearly observe how the time consumption of our implementation decreases with the decrease of the output size, which in turn decreases with the decrease in the line-segment lengths. Adversely, the time consumption of Redburn's implementation hardly changes.

Table 6.3: Random Input, 50 segments.

Input	**Time**			**Lines**
	LTS	**LLTS**	**Redburn**	
Short	3.04	1.06	300.4	0
Medium	6.80	2.82	314.0	20,742
Long	12.36	5.15	327.0	64,151

7

Conclusions and Future Work

We presented an efficient output-sensitive algorithm and an implementation with CGAL which finds all the lines tangent to four geometric objects taken from a set of n geometric objects in three-dimensional Euclidean space. In our algorithm and implementation of the LTS problem we do not assume general position. Namely, the algorithm and its implementation are robust and handle all cases. In order to improve the implementation of the LTP problem, the following future work can be done:

1. Support the degenerate case of two coplanar edges of two distinct polytopes.
2. Enhance the algorithm to support possibly intersecting polytopes.
3. Enhance the algorithm to find all the lines tangent to four non-convex polyhedra.

The approach of using two-dimensional arrangements in order to output-sensitively find all the lines tangent to four geometric objects can be enhanced to other smooth objects such as spheres. All the lines tangent to three spheres in three-dimensional Euclidean space have one degree of freedom. Moreover, Macdonald et al. showed that the number of lines tangent to four unit spheres in three-dimensional space whose centers are not colinear have at most twelve common tangent lines [MPT01]. In degenerate cases where the centers are colinear an infinite number of lines may be tangent to the four spheres. Future work is to develop and implement an algorithm, which finds all the lines tangent to four elements taken from a set of n spheres.

Bibliography

[Aus99] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1999. 24

[BBP01] Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109:25–47, 2001. 24

[BDD+07] Hervé Brönnimann, Olivier Devillers, Vida Dujmovic, Hazel Everett, Marc Glisse, Xavier Goaoc, Sylvain Lazard, Hyeon-Suk Na, and Sue Whitesides. Lines and free line segments tangent to arbitrary three-dimensional convex polyhedra. *SIAM Journal on Computing*, 37(2):522–551, 2007. 3, 4

[BEL+05] Hervé Brönnimann, Hazel Everett, Sylvain Lazard, Frank Sottile, and Sue Whitesides. Transversals to line segments in three-dimensional space. *Discrete and Computational Geometry*, 34:381–390, 2005. 1

[BFH+10] Eric Berberich, Efi Fogel, Dan Halperin, Michael Kerber, and Ophir Setter. Arrangements on parametric surfaces II: Concretizations and applications. *Mathematics in Computer Science*, 4:67–91, 2010. 7, 12, 24

[BHK11] Eric Berberich, Michael Hemmer, and Michael Kerber. A generic algebraic kernel for non-linear geometric applications. In *Proceedings of the 27th annual symposium on computational geometry*, pages 179–186, New York, NY, USA, 2011. Association for Computing Machinery (ACM) Press. 25

[BHL+11] Eric Berberich, Michael Hemmer, Sylvain Lazard, Luis Penaranda, and Monique Teillaud. Algebraic kernel. In *CGAL User and Reference Manual*. CGAL Editorial Board, 3.9 edition, 2011. `http://www.cgal.org/Manual/3.9/doc_html/cgal_manual/packages.html#Pkg:AlgebraicKerneld`. 25

[CS89] Richard Cole and Micha Sharir. Visibility problems for polyhedral terrains. *Journal of Symbolic Computation*, 7(1):11 – 30, 1989. 3

[dBEG98] Mark de Berg, Hazel Everett, and Leonidas J. Guibas. The union of moving polygonal pseudodiscs–combinatorial bounds and applications. *Computational Geometry*, 11(2):69 – 81, 1998. 3

[dBvKOS08] Mark de Berg, Mark van Kreveld, Mark H. Overmars, and Otfried Schwarzkopf. *Computational geometry: algorithms and applications*. Springer, 3nd edition, 2008. 12

[dCCLT09] Pedro M.M. de Castro, Frédéric Cazals, Sébastien Loriot, and Monique Teillaud. Design of the CGAL 3D spherical kernel and application to arrangements of circles on a sphere. *Computational Geometry: Theory and Applications*, 42(6–7):536–550, 2009. 25

[DDE+09] J. Demouth, O. Devillers, H. Everett, M. Glisse, S. Lazard, and R. Seidel. On the complexity of umbra and penumbra. *Computational Geometry — Theory and Applications*, 42:758–771, 2009. 1, 5

[DGL08] Olivier Devillers, Marc Glisse, and Sylvain Lazard. Predicates for line transversals to lines and line segments in three-dimensional space. In *Proceedings of the twenty-fourth annual symposium on computational geometry*, pages 174–181, New York, NY, USA, 2008. ACM. 7

[DP03] Olivier Devillers and Sylvain Pion. Efficient exact geometric predicates for Delaunay triangulations. In *Proceedings of the 5th Workshop Algorithm Engineering and Experiments.*, pages 37–44, 2003. 11

[ELLZ09] Hazel Everett, Sylvain Lazard, William Lenhart, and Linqiao Zhang. On the degree of standard geometric predicates for line transversals in 3D. *Computational Geometry — Theory and Applications*, 42(5):484 – 494, 2009. 2

[FH08] Efi Fogel and Dan Halperin. Polyhedral assembly partitioning with infinite translations or the importance of being exact. In *Proceedings of the 8th International Workshop on Algorithmic Foundations of Robotics*, pages 417–432, 2008. 5

[FSH08] Efi Fogel, Ophir Setter, and Dan Halperin. Exact implementation of arrangements of geodesic arcs on the sphere with applications. In *Abstracts of the 24th European Workshop on Computational Geometry*, pages 83–86, 2008. 6

[FWH11] Efi Fogel, Ron Wein, and Dan Halperin. *CGAL Arrangements and Their Applications, A Step-by-Step Guide*. 2011. 7, 12, 24

[GL10] Marc Glisse and Sylvain Lazard. On the complexity of sets of free lines and line segments among balls in three dimensions. In *Proceedings of the 26th annual symposium on computational geometry*, pages 48–57, New York, NY, USA, 2010. ACM. 3

[HLW00] D. Halperin, J.-C. Latombe, and R. H. Wilson. A general framework for assembly planning: The motion space approach. *Algorithmica*, 26:577–601, 2000. 4

[KMP+08] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Yap. Classroom examples of robustness problems in geometric computations. *Computational Geometry — Theory and Applications*, 40:61–78, 2008. 11

[MO88] M. McKenna and J. O'Rourke. Arrangements of lines in 3-space: a data structure with applications. In *Proceedings of the fourth annual symposium on computational geometry*, pages 371–380, New York, NY, USA, 1988. ACM. 7

[MPT01] I. G. Macdonald, J. Pach, and T. Theobald. Common tangents to four unit balls in. *Discrete and Computational Geometry*, 26:1–17, 2001. 41

[MS88] David A. Musser and Alexander A. Stepanov. Generic programming. In *Proceedings of the International Conference on Symbolic and Algebraic Computation*, volume 358 of *LNCS*, pages 13–25. Springer-Verlag, 1988. 5

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, USA, 1985. 11

[PW01] H. Pottmann and J. Wallner. *Computational line geometry.* Mathematics and visualization. Springer, 2001. 10

[Red03] J. Redburn. *Robust computation of the non-obstructed line segments tangent to four amongst n triangles.* PhD thesis, Williams College, Massachusetts, 2003. 2, 4, 8, 39

[SHRH11] Oren Salzman, Michael Hemmer, Barak Raveh, and Dan Halperin. Motion planning via manifold samples. In *Algorithms - ESA 2011*, pages 493–505. 2011. 25

[Str04] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Boston, MA, USA, 3rd edition, 2004. 24

[TH99] Seth Teller and Michael Hohmeyer. Determining the lines through four lines. *Journal of graphics, GPU, and game tools*, 4(3):11–22, 1999. 2, 3, 8

[WFZH07] Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin. Advanced programming techniques applied to CGAL's arrangement package. *Computational Geometry: Theory and Applications*, 38(1–2):37–63, 2007. Special issue on CGAL. 7

[WL94] Randall H. Wilson and Jean-Claude Latombe. Geometric reasoning about mechanical assembly. *Artificial Intelligence*, 71(2):371–396, 1994. 4

[Yap04] Chee-Keng Yap. Robust geomtric computation. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapman & Hall/CRC, Boca Raton, FL, 2nd edition, 2004. 7, 11

[YD95] Chee-Keng Yap and Thomas Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 1 of *LNCS*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995. 7

[ZEL+08] Linqiao Zhang, Hazel Everett, Sylvain Lazard, Christophe Weibel, and Sue Whitesides. On the size of the 3d visibility skeleton: Experimental results. In Dan Halperin and Kurt Mehlhorn, editors, *Algorithms - ESA 2008*, volume 5193 of *LNCS*, pages 805–816. Springer Berlin / Heidelberg, 2008. 4

Links

[1] Boost — portable C++ libraries.
http://www.boost.org. 26

[2] Cgal — computational geometry algorithms library.
http://www.cgal.org. 6, 7

[3] Gmp — GNU multiple precision arithmetic library.
http://gmplib.org. 6