Robust and Efficient Software for Problems in 2.5-Dimensional Non-Linear Geometry

Algorithms and Implementations

Dissertation

zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

vorgelegt von

Eric Berberich

Saarbrücken
2008

Tag des Kolloquiums:
22. Dezember 2008

Dekan der Naturwissenschaftlich-Technischen Fakultät I:
Prof. Dr. Joachim Weickert

Berichterstatter:
Prof. Dr. Kurt Mehlhorn
Prof. Dr. Stefan Schirra
Prof. Dr. Dan Halperin

Mitglieder des Prüfungsausschusses:
Prof. Dr. Reinhard Wilhelm (Vorsitzender)
Universität des Saarlandes, Saarbrücken, Deutschland
Prof. Dr. Kurt Mehlhorn
Max-Planck-Institut für Informatik, Saarbrücken, Deutschland
Prof. Dr. Stefan Schirra
Otto-von-Guericke-Universität, Magdeburg, Deutschland
Prof. Dr. Dan Halperin
Tel-Aviv University, Tel-Aviv, Israel
Dr. Michael Sagraloff
Max-Planck-Institut für Informatik, Saarbrücken, Deutschland

Abstract

We discuss how to compute and implement three geometric problems dealing with non-linear three-dimensional surfaces. As a main tool we rely on planar subdivisions induced by algebraic curves, developed in CGAL (Computational Geometry Algorithm Library).

First, we achieve lower envelopes of quadrics using CGAL's `Envelope_3` package. Second, we extend CGAL's `Arrangement_2` package to support two-dimensional arrangements on a parametric reference surface. Two main examples are discussed: Arrangements induced by algebraic surfaces on an elliptic quadric and on a ring Dupin cyclide. Third, we decompose a set of quadrics or a set of algebraic surfaces into cells using projection. Our goal is to achieve topological information for the surfaces, while preserving their geometric properties. We maintain a special two-dimensional arrangement; the lifting to the third dimension benefits from the recently presented bitstream Descartes method. The obtained cell decomposition supports a set of other geometric applications on surfaces.

Our implementations follow the geometric programming paradigm. That is, we split combinatorial tasks from geometric operations by generic programming techniques. It is also ensured that each geometric predicate returns the mathematically correct result, even if it internally exploits approximative methods to speed up the computation.

The thesis is written in English.

Zusammenfassung

Wir besprechen die Berechnung und Implementierung dreier Probleme aus der algorithmischen Geometrie, deren Eingabe aus gekrümmten Oberflächen besteht. Als Werkzeug benutzen wir in CGAL (Computational Geometry Algorithm Library) entwickelte Zerlegungen der Ebene durch algebraische Kurven.

Zunächst berechnen wir die untere Einhüllende einer Menge von Quadriken. Danach erweitern wir CGALs `Arrangement_2` Paket, so dass zweidimensionale Zerlegungen auf parameterisierbaren Oberflächen berechnet werden können, und führen zwei konkrete Beispiele aus: Zerlegungen induziert durch algebraische Oberflächen auf einer Quadrik und auf einem ringförmigen Zykliden nach Dupin. Zum Abschluss unterteilen wir eine Menge von Quadriken bzw. algebraischen Oberflächen in disjunkte Untermannigfaltigkeiten mit Hilfe einer Projektion. Die Hebung erfolgt mit einem kürzlich vorgestellten approximativen Verfahren zur Nullstellenisolation (bitstream Descartes). Ingesamt erhalten wir geometrische Eigenschaften der Eingabe und erfahren mehr über deren topologische Zusammensetzung. Die kombinatorische Ausgabe hilft bei der Berechnung anderer geometrischer Probleme auf den Oberflächen.

Unsere Implementierungen trennen kombinatorische Aufgaben von geometrischen durch Anwenden von generischen Programmiertechniken. Wir stellen außerdem sicher, dass Prädikate stets das mathematisch korrekte Ergebnis ausgeben, auch wenn sie intern mit approximativen Methoden rechnen.

Die Arbeit ist in englischer Sprache verfasst.

Acknowledgments

First of all I want to thank Kurt Mehlhorn for giving me the great opportunity to work in his group with the freedom to work on the research I wanted to zoom in. The bracing atmosphere at the "Max-Planck-Institut für Informatik" is unique and catching. It is a pleasure to work with so many interesting people. In particular, I appreciate all my office-mates for a warm and relaxed atmosphere, and all my collaborators in the EXACUS-project, especially Arno Eigenwillig, Michael Hemmer, Michael Kerber, Michael Sagraloff, and Pavel Emeliyanenko. As a team we achieve wonderful results and software. My thanks naturally continues towards the developers of CGAL and the members of its editorial board for electing me to be among experienced "dinosaurs".

I thank the reviewers of my work for very useful comments, good advice, and their gradings — be it for submission at conferences or this thesis. I also appreciate all comments that I received on preliminary versions of this thesis.

During my doctorate studies I had the wonderful chance to visit twice the lab of Dan Halperin at Tel-Aviv University in Israel. I thank him and his team (especially Efi Fogel, Ophir Setter, and Ron Wein) for giving me a warm and friendly welcome and for the very fruitful collaboration that we started. I believe and know that this cooperation is still growing.

A special thank goes to Ingrid Finkler-Paul, Christina Fries, and Petra Mayer who always helped me perfectly with the small and big organizatorial tasks, such as travels. Another helping hand is Joachim Reichel, who also provided me with his LaTeX-template for this thesis.

This work has been supported in part by the IST Programme of the EU as Shared-cost RTD (FET Open) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes). I enjoyed meeting many interesting people and also working with them.

My deep and final "thank you" goes to my family who gives me advice and supported me in all circumstances. My special graditude is for my girlfriend Anne-Kathrin who never stops believing in me.

Contents

List of Figures

List of Tables

List of Algorithms

The world is not linear.

1 Introduction

Geometry is one of the oldest sciences on earth. Several thousand years ago, people had already discovered principles about lengths, angles, areas, and volumes. Progress in understanding geometry was mainly driven by practical needs required in some crafts: surveying the earth to create maps (e. g., to demarcate ownership), astronomy, and, of course, constructions of buildings and other infrastructure. More generally, geometry[1] is a subfield of mathematics that deals with the shapes of objects, their sizes and relative positions, and with the properties of space. Euclid presented fundamental axioms of geometry in his books. The field is also strongly coupled with numbers that represent geometric entities, such as lengths and areas, and also coordinate systems that were introduced by Descartes. Descartes also observed the connection between geometric objects and their algebraic descriptions. Actually, the field of algebraic geometry, which defines objects by polynomial equations, is a large and important subfield on its own. In particular, its low-dimensional variant for real spaces are critical for many recent applications in the world of surveying, motion planning, and construction. In these latter areas, the focus is on curved objects, that is, objects defined by non-linear equations, such as circles, spheres, cones, tori, and many others. Even the earth itself is an ellipsoid.[2] Dealing with such curved objects is often not solely for artistic purpose. In contrast, curved objects are essential for specific design goals. For instance, car and plane manufacturers try to reduce the air drag coefficient that saves fuel, or loudspeakers have a curved chassis to avoid undesired acoustic reflections. Fields like computer-aided geometric design (CAGD), robotics, or molecular biology can model their problems with algebraic equations, which provide accurate techniques. Indeed, accuracy is a central goal in geometry as a slight displacement of an object may result in a completely different combinatorial relation among the geometric shapes. While the movement of a geometric object may still be continuous, that is, it can be performed without *jumps*, the alignment of the object with respect to another one might be

[1] Based on Greek words *geo* for earth and *metria* for measurement.

[2] Actually, the earth is a *geoid* whose shape is dependent on the local gravity. However, the ellipsoidal shape is the state-of-the-art technique to model the surface, for example, for geographic information systems.

non-continuous. An example is the relative position of a point and an infinite line in the plane; see Figure 1.1. While moving the point, it can be uniquely determined whether

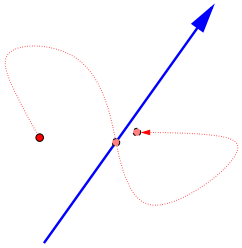
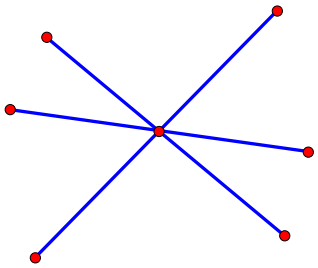


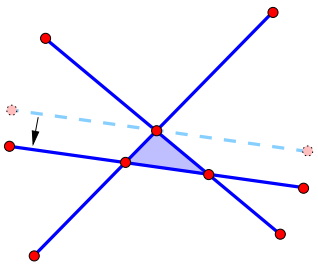
Figure 1.1. Non-continuous function for red point and blue (oriented) line: It is either to the left of the line, on it, or to its right.

the point meets the line or on which of the two sides of the line it resides. The point's continuous movement is mapped to a three-valued status. Another example is illustrated in Figure 1.2. We see segments intersecting at a common point. Notice that the picture *also* visualizes their combinatorial relation: It induces a graph whose nodes are endpoints of segments and their intersections. An edge is added between two nodes if they are connected by a piece of an input segment. If we slightly move one of the segments, as, for instance, in Figure 1.2 (b), the graph changes dramatically:

Figure 1.2. Geometry induces combinatorics



(a) Three segments intersect in a single point. The induced graph has 7 nodes and 6 edges.



(b) A slight change in the geometry can have much impact on the structure of the graph: It now has 9 nodes and 9 edges.

The number of nodes increases, and a non-empty bounded area surrounded and defined by segments (shaded) appears. Note that the existence of this area can also be modelled as a non-continuous function in terms of the position of the segments. *Thus, we emphasize that in our definition of geometry, dealing with geometric objects also involves analyzing their combinatorial structure.* The structure is determined by evaluations of a number of non-continuous functions — that we also call *predicates*. This, indeed, opens an algorithmic way to tackle geometric problems.

While dealing with non-continuous functions poses no problem in theory, the field of *geometric computing* strives for an *actual* and *robust* algorithmic handling of geometric problems *on a computer*. Doing so efficiently is also one of its important objectives. Examples are to compute convex hulls and Voronoi diagrams, to reconstruct surfaces from a

point cloud, or to compute the partitioning of a space induced by geometric objects as in the recent example. Usually, each such task can be solved by a combinatorial algorithm whose execution path is determined by geometric constructions and, as mentioned, evaluations of non-continuous functions. A central goal is to *guarantee* the *termination* and the *correctness of the output*. This goal can be achieved if two principles are fulfilled: First, the algorithmic design is guaranteed to deal with all possible cases. That is, it also handles so-called *degeneracies*. Second, the evaluations of non-continuous functions have to compute the correct values. If these goals cannot be fulfilled, a geometric computation can quickly crash, loop forever, or simply produce incorrect results. It typically requires extraordinary effort to meet the second requirement on a computer. The reason is that, as mentioned, numbers play a key role in geometry, but standard hardware that carries out arithmetic represents only finite sets of (solely the rational) numbers.

It is no secret that both problems have been successfully tackled, even in software, when computing geometric problems with *linear* objects, such as line segments. Starting in the 1990s, researchers have been providing more and more robust implementations for various geometric tasks. Main examples of libraries collecting such software are LEDA, the Library of Efficient Data structures and Algorithms, and CGAL, the Computational Geometry Algorithms Library. It is exciting that their implementations are highly efficient and even competitive with non-robust software.[3] However, there are also curved objects, especially the mentioned ones that are defined algebraically. In three-dimensional space, they are formed by the vanishing sets of uni-, bi-, and trivariate polynomials. Such became more popular recently in several domains: Computer graphics, computer aided geometric design, motion planning, and robotics. One way to approach such issues is to approximate each object with a corresponding set of linear objects, for example, connected line segments for curves, or triangular meshes for surfaces. But approximation implies drawbacks. First, it is hard to ensure that the function evaluations on approximations reflect the exact version and thus subsequent computations actually output the correct answers. Second, the number of linear objects required to reach this stage might be very large, if possible at all. This may lead to an inefficient approach. On the other hand, it might be advantageous to directly deal with curved objects, that is, non-linear algebraic ones — although this objective is highly ambitious. Exploiting generic and symbolic computer algebra systems seems to be the alternative. Cylindrical algebraic decomposition (cad) is perhaps the most famous example. Unfortunately, such systems usually have extremely long running times. However, in recent years, computational geometers have developed robust *and* efficient software for curved objects, too. This work focusses on geometric and topological properties. The key to success is to abstract combinatorial tasks from simple predicates, and to replace their costly symbolic evaluations with approximative but certified computations as much as possible. But up to now, most work of that kind has been restricted to curves embedded in the plane.

1.1. Our contributions

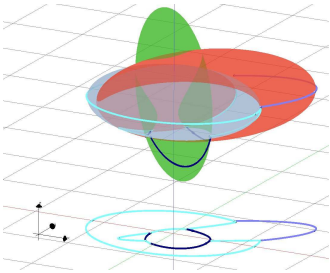
The main contents of this thesis are exactly cut from the same cloth. We aim for robust and efficient software for geometric problems, but in "2.5 dimensions". The fraction indicates,

[3] If the non-robust version computes "by accident" the correct result—in other cases, a comparison is not meaningful.

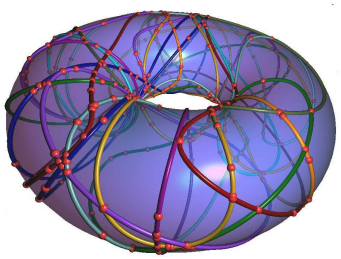
that the input is usually a set of surfaces in three-dimensional space, but either the output is two-dimensional, or we reduce the problem to a two-dimensional one in order to compute the desired three-dimensional output. In particular, we deal with the following challenges. Each is a geometric problem whose input consists of surfaces in $\mathbb{R}^3$:

(a) Construct lower envelopes of quadrics
(b) Construct and maintain arrangements on two-dimensional parametric surfaces
(c) Stratify algebraic surfaces using planar arrangements

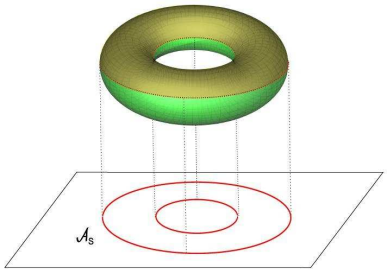
Figure 1.3. Examples of our contributions



(a) Lower envelope of three quadrics



(b) Arrangement on a torus induced by ten quadrics



(c) Decompose set of surfaces into finitely many lifted cells (and compute adjacencies)

taken from presentation of [BKS08]

Each of the main chapters is dedicated to one challenge. It turns out that the construction of two-dimensional arrangements is a fundamental and essential tool for each. For this purpose, we rely on CGAL's matured `Arrangement_2` package developed by Dan Halperin's group at Tel-Aviv University with main contributions by Ron Wein and Efi Fogel. The problems we discuss mainly utilize this package, while (b) describes its new generalization that we completed in collaboration with colleagues at Tel-Aviv University. For each challenge, we show its relation to two-dimensional arrangements, and we also identify which problem specific adaptations are required.

Computing lower envelopes of surfaces also exploits CGAL's generic `Envelope_3` package. In a remarkable amount of engineering work, Michal Meyerovitch from Tel-Aviv University extended planar arrangements to provide this enhancement. In order to support a certain family of surfaces, the implementation expects a certain set of geometric types, predicates, and constructions. In collaboration with Michal, we provide a proper and runtime-efficient set for the case of quadrics.

For the other two challenges we develop new combinatorial frameworks that decouple generic issues from surface-specific tasks. We also instantiate them with concrete implementations. That is, we learn how to compute arrangements on an elliptic quadric or a ring Dupin cyclide (a generalization of a torus), both induced by algebraic surfaces intersecting the reference surface. The latter is joint work with Michael Kerber from the Max-Planck-Institut für Informatik. For the stratification of surfaces, we provide the required geometric operations for quadrics and for algebraic surfaces of any degree. These results are based on joint work with Michael Kerber and Michael Sagraloff.

Besides two-dimensional arrangements, our three main objectives fundamentally rely on a *two-dimensional algebraic curve kernel* that provides exact analysis of planar algebraic curves and pairs of such curves. An important instance of such a kernel has mainly been developed and is maintained by Michael Kerber. As our various implementations mostly perform combinatorics on such planar analyses, it is no surprise that the final performance measures we observe strongly depend on the efficiency of the supporting two-dimensional algebraic kernel.

We remark that challenge (b) and (c) constitute major building blocks towards three-dimensional arrangements of algebraic surfaces, or at least possibilities to support them. While we decompose the input into zero-, one-, and two-dimensional cells, we are not able yet to combine them into a coherent data structure that combinatorially represents the induced partitioning of the space, including (maximal) connected three-dimensional cells. The obtained accurate topological and geometric information of algebraic objects is crucial for other interesting utilizations, such as computing substructures, good visualizations, and for meaningful approximations by simpler objects (as triangles or splines).

Finally, it should be mentioned, that our achieved results match goals recorded in 2004 as part of a strategy report of the EXACUS project [Exa04]. This project started in 2001 at the Max-Planck-Institut für Informatik in Saarbrücken, aiming for robust, efficient, and complete software for non-linear curves and surfaces. Main parts of our software are now contained in CGAL, as EXACUS is absorbed in CGAL.

1.2. Related work

Implementing robust and efficient algorithms for non-linear problems in computational geometry has received a lot of attention in recent years, especially for algebraically defined objects. A fundamental problem is the (real) root isolation of a univariate polynomial that is often a key substep for more sophisticated algorithms. Several techniques exist, each having advantages and disadvantages. Real root solving using continued fractions has been considered in [TE08]. A method relying on Descartes' rule of signs with optimal memory consumption and using multiprecision interval arithmetic is presented in [RZ03]. Its adaptation into a CGAL-like interface is shown in [LPT08]. Completely implemented in CGAL are two real root solvers, both based on Descartes's rule of sign [HL07]. While one deals with an exact representation of the coefficients, the other interfaces them as possibly infinite bitstreams. This allows one to isolate real roots of polynomials whose coefficients are algebraic or even transcendental [EKK+05]. A comparison of different approaches is conducted in [EHK+08].

In two dimensions, the prominent example is the exact computation of arrangement induced by curved objects. A key contribution in terms of software is CGAL's `Arrangement_2` package developed by Dan Halperin's group at Tel-Aviv University [WFZH07a]. Besides basic linear objects contained in the package and CGAL's kernels, there exists support for various families of curves (and arcs of them): Pion and Teillaud give a circular kernel that enables the computation of arrangements of circles and line segments [PT07]. The same goal is aimed at by work of Wein and Zukerman [WZ06]. Wein also presented how to compute arrangements of conics [Wei02]. In cooperation with Hanniel he developed an exact implementation that allows to compute arrangements of Bézier curves [HW07]. There is also a joint initiative to develop an "open curved kernel" [EKP+04].

In parallel, the members of the EXACUS project also derived robust and efficient algorithms (as software) to compute arrangements of non-linear curves and the corresponding arcs of these curves. Berberich et al. developed the CONIX library. It allows one to consider (arcs of) conic curves and polygons bordered by such curves [BEH+02]. Besides proper support for CGAL's `Arrangement_2` package, it has been shown how to extend LEDA's sweep line algorithm, which originally dealt only with line segments, to curved input. Later, Eigenwillig et al. extended the set with cubic curves in the CUBIX library [EKSW06]. Recently, Kerber et al. have been able to robustly implement the analyses of algebraic curves of any degree [EKW07], [EK08a], which allows to compute arrangements of them using a framework that interfaces the analyses into geometric predicates and constructions [BE08].

The `Arrangement_2` package itself is augmented with various interesting extensions, such as observers that get notified about structural changes of an arrangement, the possibility to overlay two arrangements, or various point location strategies; see [WFZH07b]. In addition, it is possible to compute lower envelopes of them [Wei07a] or to perform regularized boolean set operations [FWZH07]. More details on planar arrangements appear in §2.4. They are also utilized in CGAL's `Envelope_3` package by Meyerovitch that allows to compute (lower) envelopes of surfaces defined in three-dimensional space; see [Mey06a], [Mey06b], and [MWZ07]. Her implementation applies a randomized divide-and-conquer-strategy and makes use of (dis)continuity information of the surfaces and their intersections. Note that the problem is two-and-a-half-dimensional: The input consists of objects in $\mathbb{R}^3$, while the output decomposes a two-dimensional space.

When increasing the dimension from two to three, we are also aware of related results. First, we mention ESOLID, a boundary evaluation system by Keyser et al. [KCF+04]. It can deal with low-degree curved solids (such as quadrics). However, it requires that solids be in general position. Namely, it is not able to handle all degeneracies. Additionally, there exist three main specialized approaches for quadrics. The first sweeps a plane through the scene of quadrics, maintaining a pseudo-trapezoidal decomposition on the plane. This approach is due to Mourrain et al. [MTT05]; however, an implementation is missing. The second technique uses a parameterization of the intersection curves by Dupont et al. [DLLP08a, DLLP08b, DLLP08c], which is based on Levin's result [Lev79]. It has been used to successfully construct the adjacency graph of quadrics [DHPS07]. The third approach by Berberich et al. [BHK+05] computes for a given reference quadric two planar arrangements, one for its lower part and one for its upper part. In combination, these arrangements encode the arrangement that is induced by other quadrics intersecting the reference surface. While these approaches do not fully compute a three-dimensional arrangement, CGAL provides the `Nef_3` package which is a complete, robust, and efficient implementation for three-dimensional Nef-Polyhedra [HK07b], [HKM07]. A d-dimensional Nef-polyhedron is a point set $P \subset \mathbb{R}^d$ generated from a finite number of open halfspaces by set *complement* and set *intersection* operations [Nef78]. Union and (symmetric) difference can be reduced to intersection and complement. The topological operations *boundary*, *interior*, *exterior*, *closure*, and *regularization* can also be modelled with Nef-polyhedra. The package is restricted to linear features.

This compiled list of results comprises general results obtained in the computational geometry's subarea of robust realizations for (non-linear) geometric problems. Further results specific to one of our three problems are postponed to the individual related chapters. There, we first introduce the problems themselves in more details.

1.3. Outline

But first, in Chapter 2, we give a comprehensive review of concepts and tools required throughout this thesis. This includes an introduction to algebraic foundations, a guide for implementing robust geometric algorithms, the presentation of available arithmetic and algebraic tools, and finally a detailed discussion of planar arrangements, as they are the connecting entity of the main chapters.

Chapter 3 starts with a short introduction to (lower) envelopes and also remembers how to robustly compute intersection curves induced on a quadric. In its main section we discuss how to obtain required geometric types and predicates in order to compute lower envelopes of quadrics. The chapter closes with experimental results and some variants.

Thereafter, in Chapter 4, we present an extension to CGAL's `Arrangement_2` package that allows to construct and maintain arrangements on two-dimensional orientable parametric surfaces. The chapter is organized as follows. We first introduce the setting followed by a discussion of existing work. We then show in individual steps how we augment algorithms and implementations for planar arrangements to finally support arrangements on parametric surfaces. To do so, we abstract surface- and curve-specific geometric and topological tasks from generic functionality. As a first step we show how to obtain a unique order of events on the surface, even if some points have multiple pre-images in the parameter space of the surface. As second step, we show how a new collection of simple surface-specific functions can be used to consistently construct the DCEL (double-connected-edge-list) that represents the induced arrangement. At the end of the chapter we describe two example surfaces in detail. We consider arrangements on elliptic quadrics induced by other quadrics, and arrangements on ring Dupin cyclides (containing the torus as special case) that are induced by the intersection with algebraic surfaces of arbitrary degree. We show that the geometric operations can be established by mostly combinatorial recombinations of operations actually designed for algebraic plane curves. We conclude the chapter with an outlook for future directions.

In Chapter 5 we show how to stratify a set of algebraic surfaces. We first abstractly identify required tasks, and introduce a decomposition of the given surfaces into cells. We then show that algebraic surfaces serve these needs. Our actual implementation is split into two parts. The combinatorics are handled by a framework that defines a set of tasks demanded by surfaces. We are able to implement these tasks for algebraic surfaces of arbitrary degree, and a specialized version for quadrics that exploits their low degree. We finally show utilizations that can be implemented in terms of the achieved output. Results of experiments are reported before we conclude the chapter with directions for future progress and research.

2

Algebraic Foundations Geometric Programming Arrangements

The main parts of this thesis cover the area of curved geometry, that is, it deals with objects beyond segments, triangles, planes, even beyond spheres. The role of this chapter is to equip the reader with basic terminology and fundamental information on the objects, basic tools and data structures we deal with in later chapters, namely with and towards arrangements of algebraic objects in two and three dimensions.

The geometric objects we want to handle are defined algebraically. They form a class of non-linear input, while their particularities pop up interesting cases to consider. §2.1 introduces very basic algebraic notation and main tools, like polynomials, sequences of them, their roots, and how to isolate real roots. In §2.1.4 we turn towards algebraic curves, while §2.1.5 covers algebraic surfaces. Both are defined by multivariate polynomials. A general ansatz for dealing with arbitrary polynomials in any dimension (actually for quantifier elimination) is the cylindrical algebraic decomposition that we present in §2.1.6. We close the theoretical introduction by some terms of topology in §2.1.7.

Implementing geometric algorithms is a highly non-trivial task, especially if the input consists of curved objects. As we are not only interested in theoretical algorithm design, but also aim for a state-of-the-art implementation of our algorithms, §2.2 surveys occurring difficulties, introduces the geometric programming paradigm, and presents the geometric libraries CGAL and EXACUS.

The development of geometric software from scratch is not necessary. A large number of tools are available. §2.3 showcases the kit we use. It consists of number types, filter techniques and algebraic kernels. Such kernels exist for the one- and the two-dimensional case. We also give details on the interface of a special real root isolator.

We close the chapter with an introduction to a basic but very fundamental structure in computational geometry in its own section, namely the arrangement. Arrangements can be defined in any dimension, however in §2.4 we focus on the cases where $d = 3$ and

for $d = 2$. Throughout the thesis, two-dimensional arrangements form the main building block. Thus, we shortly repeat how to construct and maintain planar arrangements, followed by details of two-dimensional planar arrangements in CGAL. We finally present in §2.4.4 a generic class that queries a so-called two-dimensional algebraic kernel with analysis (see §2.3.3) in order to provide basic geometric types and operations required for CGAL's `Arrangement_2` package. Depending on the algebraic kernel this triple enables a user to compute arrangements of algebraic curves.

2.1. Algebraic foundations

A lot of geometric objects, even the very simple ones, are usually (piecewise) defined by (semi-)algebraic sets. In particular, all objects we are dealing with in the main chapters are algebraically defined. Thus, we sketch central algebraic concepts and considerations which should already be known to an experienced reader. Most of this content is basic and previously appears in standard textbooks like [vdW71], [Lan02], [Bos06], [CLO97], [CLO05], or the comprehensive overview in [MPS$^+$]. This also implies that the tools we introduce are well-known and proven, such that we are less comprehensive than any of the given references. We refer to them for very basic concepts, generalizations of the results that we state, and the proofs. In contrast, we try to formulate the tools as algorithmically as possible, as our ultimate goal is also to provide a working implementation. It is above all Chapter 5 for which we unreel some of the theory. The other chapters rely on combinatorial information of algebraic curves by properly querying analyses provided by algebraic kernels.

2.1.1. Polynomials

The key expressions in our compiled list of algebraic concepts are polynomials.

Definition 2.1 (Polynomial). Let $\mathbb{D}$ be a factorial domain. An expression of the form

$$f = \sum_{i=0}^{n} a_i t^i \in \mathbb{D}[t]$$

is a *polynomial over* $\mathbb{D}$ with *coefficients* $a_n \neq 0, a_{n-1}, \ldots, a_0 \in \mathbb{D}$. We may regard *variable* t as a formal symbol of indeterminate meaning. $\mathbb{D}[t]$ denotes the ring of polynomials with coefficients in $\mathbb{D}$.

Properties of polynomials We start with very technical terms for a given polynomial f. The *degree* of f, denoted by $\deg(f)$ is the greatest non-vanishing power of t, which is n as we have $a_n \neq 0$. If $f \equiv 0$, $\deg(f) = -\infty$. Another expression for the i-th coefficient a_i is $\mathrm{coef}_i(f)$. We call a_n the *leading coefficient* of f and denote it $\mathrm{lcf}(f) = \mathrm{coef}_n(f)$. With $f_{(k)} := \sum_{i=0}^{k} a_i t^i$ we denote the *k-th reductum* of f.

With $\mathbb{K}$ we denote a field that contains $\mathbb{D}$. We usually refer to $\mathbb{K} = \mathbb{Q}$ or $\mathbb{K} = \mathbb{C}$ which is already algebraically closed. We use the fraction field $\mathbb{K} = \mathbb{Q}(u_1, \ldots, u_k)$ if the problems depends on parameters $u_1, \ldots, u_k$. Remember that $\mathbb{D} = \mathbb{Z}$ (or $\mathbb{D}$ being a field) is factorial, that is, $0 \neq r \in \mathbb{D}$ can be decomposed (up to order) into $r = u \cdot r_1 \cdot \ldots \cdot r_\ell$ with u being a unit, $r_i \in \mathbb{D}$, and all irreducible in $\mathbb{D}$. Following Gauss' theorem ([Bos06, §2.7]), it holds $\mathbb{D}[t]$ is also factorial, which has several implications.

First, for $a_i, a_j \in \mathbb{K}$, the $\gcd(a_i, a_j)$ exists and is well-defined, and so for $f, g \in \mathbb{K}[t]$. The *content* of f is the gcd of the coefficients, that is, $\mathrm{cont}(f) = \gcd(a_1, \ldots, a_n)$. We refer to a *primitive* polynomial if $\mathrm{cont}(f) = 1$, and to the *primitive part* of f for $\mathrm{pp}(f) := \frac{f}{cont(f)}$.

A polynomial $g \in \mathbb{K}[t]$ is a *factor of* f if there exists a polynomial $h \in \mathbb{K}[t]$ with $f = g \cdot h$. Contrary, two polynomials $f, g \in \mathbb{K}[t]$ are called *coprime* if $\gcd(f, g)$ is a constant. We can also define the *factorization* of $f \in \mathbb{K}[t]$ $(\deg(f) > 0)$ by $f = u \cdot \Pi_{i=1}^{n} f_i$ with $u = \mathrm{lcf}(f)$ and f_i being monic irreducible elements of $\mathbb{K}[t]$ with positive degree. We call f *square-free* if all f_i are distinct. For a square-free f it holds that $\gcd(f, f')$ is a constant. On the contrary, the square-free part $f^\star$ of f can be obtained by $f^\star = \frac{f}{gcd(f,f')}$. Alternatively, one can also compute a finer granulation of f into square-free factors $\hat{f}_j$. We group the f_i by their number of occurrences which results in a *square-free factorization* $f = u \cdot \Pi_{j=1}^{k} \hat{f}_j^j$, that is, $\hat{f}_j \in \mathbb{K}[t]$ contains all f_i that appear j times in the factorization of f. It is obvious that $k \leq n$. *Yun's square-free factorization algorithm* cleverly combines iterated gcds to compute such $\hat{f}_j$; see [GCL92, Algorithm 8.2] and [Yun76] for details. For our purposes the weaker concept of the square-free factorization fulfills the needs.

Roots of polynomials

Definition 2.2 (Root). Let $f(t) \in \mathbb{D}[t]$ be a polynomial. We call an element α with $f(\alpha) = 0$ a *root* of f.

Usually, the roots of f are not necessarily elements of $\mathbb{D}$. We mostly refer to the real roots of a polynomial. Switching to the algebraic closure $\mathbb{C}$ of $\mathbb{D}$ allows to write f, with $\deg(f) = n$, as a product of linear factors

$$f(t) = u \cdot \Pi_{i=1}^{n} (t - \alpha_i)$$

with u being the leading coefficient of f and α_i being the not necessarily distinct roots of f over $\mathbb{C}$, whose number is n.

Definition 2.3 (Multiplicity). Let $f(t) \in \mathbb{D}[t]$ be a polynomial with root $\alpha \in \mathbb{C}$. The number of linear factors $(t - \alpha)$ in $f(t)$ defines the *multiplicity* m of α as root of f. Such a root is called *simple* if $m = 1$, and *multiple* if $m > 1$. We also refer to the m-fold root α of f.

It can be shown, that a square-free polynomial over $\mathbb{D}$ only contains simple roots, as otherwise, some factor appears twice and thus, each root of such a component must be a multiple of the polynomial.

Multivariate polynomials A polynomial ring $\mathbb{D}[t]$ can serve as a domain again. This strategy yields to *multivariate polynomials* whose ring is given by $\mathbb{D}[t_1] \ldots [t_d] = \mathbb{D}[t_1, \ldots, t_d]$. The order of adjunction can be chosen freely. Two views on a multivariate polynomial f are common.

Hierarchical: f is univariate in a chosen *outermost* variable, say t_d, that is, $f \in D[t_d]$ with $D = \mathbb{D}[t_1, \ldots, t_{d-1}]$.

Flat: f is expressed as a sum of *monomials* $a_{i_1,\ldots,i_d} t_1^{i_1}, \ldots, t_d^{i_d}$.

The *total degree* $\deg_{\text{total}}(f)$ of f is the highest sum of exponents $i_1 + \ldots + i_d$ among all monomials in the flat view. The value $\deg_{t_i}$ is equal to $\deg(f)$ assuming f being univariate in t_i. A multivariate polynomial $f \in \mathbb{D}[t_1, \ldots, t_d]$ is *t_i-regular* if it contains a monomial of the form $c \cdot t_i^{\deg_{\text{total}}(f)}$ with $0 \neq c \in \mathbb{D}$, which is equivalent to $\deg_{\text{total}}(f) = \deg_{t_i}(f)$.

In the hierarchical view, we can decompose a multivariate $f \in \mathbb{D}[t_1, \ldots, t_d]$ into $f = \text{cont}_{t_d}(f) \cdot \text{pp}_{t_d}(f)$, where $\text{cont}_{t_d}(f) \in \mathbb{D}[t_1, \ldots, t_{d-1}]$ and $\text{pp}_{t_d}(f) \in (\mathbb{D}[t_1, \ldots, t_{d-1}])[t_d]$. We call a multivariate polynomial f *square-free* if $\text{cont}_{t_d}(f)$ and $\text{pp}_{t_d}(f)$, seen as univariate polynomials in t_d, are square-free. Mind a possible recursion for $\text{cont}_{t_d}(f)$. f being square-free is equivalent to $\gcd(f, f') = c$ and also equivalent to: There is no non-trivial $g \in \mathbb{D}[t_1, \ldots, t_d]$ with $g^2 | f$. Computing a square-free factorization of f reduces to compute one for $\text{cont}_{t_d}(f)$, one for $\text{pp}_{t_d}(f)$, and to multiply factors of same multiplicity. The later step is often omitted as $\text{cont}_{t_d}(f)$ has interesting properties with respect to the vanishing set of f that we introduce in Definition 2.4 and used in §2.1.4 (page 38 ff) and §2.1.5 (page 42 ff).

For numbers $\overline{\alpha} = (\alpha_1, \ldots, \alpha_d) \in \mathbb{K}^d$ we can evaluate f either in full which results in a scalar $s \in \mathbb{K}$, or with a subvector of $\overline{\alpha}$, which gives another polynomial over a ring dependent on the domain of the α_i. Actually, arbitrary evaluation is not expected often, but the following set of homomorphisms is of interest. For a fixed k: $\mathbb{D}[t_1, \ldots, t_d] \to \mathbb{K}^{d-k}$. Let $\overline{\alpha}^{(k)}$ be a sequence (vector) of k numbers $(\alpha_1, \ldots, \alpha_k)$ from a field $\mathbb{K}$ and $f_{\overline{\alpha}^{(k)}} := f(\alpha_1, \ldots, \alpha_k, t_{k+1}, \ldots, t_d) \in \mathbb{K}[t_{k+1}, \ldots, t_d]$, that is, evaluating the d-dimensional polynomial f with $k \leq d$ numbers α_i results in a $(d-k)$-dimensional polynomial over $\mathbb{K}$. We often have $k = d - 1$, which eventually leads to a univariate polynomial $\in \mathbb{K}[t_d]$.

Definition 2.4 (Vanishing set). Let $f(t_1, \ldots, t_d) \in \mathbb{D}[t_1, \ldots, t_d]$ be a polynomial and $\mathbb{K}$ be a field. We call $V_{\mathbb{K}}(f) := \{\overline{\alpha}^{(d)} \in \mathbb{K}^d \mid f_{\overline{\alpha}^{(d)}} = f(\alpha_1, \ldots, \alpha_d) = 0\}$ the *vanishing set* of f over $\mathbb{K}^d$.

The following proposition is essential for us and also easy to verify.

Proposition 2.5. *Let $f \in \mathbb{K}[t_1, \ldots, t_d]$ with $f = f_1 \cdot f_2$ and $f_1, f_2 \in \mathbb{K}[t_1, \ldots, t_d]$. Then $V_{\mathbb{K}}(f) = V_{\mathbb{K}}(f_1) \cup V_{\mathbb{K}}(f_2)$. Direct implications are $V_{\mathbb{K}}(f) = V_{\mathbb{K}}(c \cdot f)$, with $0 \neq c \in \mathbb{K}$, $V_{\mathbb{K}}(f) = V_{\mathbb{K}}(f_1^k f_2^\ell)$, with $k, l \in \mathbb{N}$, and if $f_1 | f$, then $V_{\mathbb{K}}(f_1) \subset V_{\mathbb{K}}(f)$ (similar for f_2).*

Our geometric applications mainly strive for objects defined by the vanishing sets of (simple) integral polynomials in dimensions 1, 2 and 3 over $\mathbb{R}$. However, as $\mathbb{D} = \mathbb{Z}$ the gcd and the square-free factorization are only definable *up to constant factor*. That is, it is possible to compute for $f, g \in \mathbb{Z}[t]$ a polynomial $g = c \cdot \gcd(f, g)$, with $c \in \mathbb{Z}$ (and similar for the other decompositions). The good news is, that, as stated in Proposition 2.5, such a constant factor does not change the vanishing sets of the resulting polynomials in which we are mainly interested in subsequent parts; see §2.1.2 for real roots, §2.1.4 for algebraic curves, and §2.1.5 for algebraic surfaces.

Polynomial sequences We next turn to more sophisticated algebraic tools, namely subresultant and Sturm-Habicht sequences. They are well-studied in algebraic geometry, such that we omit to unreel the full theoretical considerations, and refer to textbooks discussing them in detail. We narrow their introduction to mention their existence and give results relevant for our further considerations.

Definition 2.6 (Sylvester matrix, subresultant and sequences). Given $f = \sum_{i=0}^{n} a_i t^i \in$

$\mathbb{D}[t]$ and $g = \sum_{i=0}^{m} b_i t^i \in \mathbb{D}[t]$ with $n = \deg(f) \geq \deg(g) = m > 0$.

- For $k \leq m$, the *k-th Sylvester submatrix* has dimension $(m+n-2k) \times (m+n-k)$, build with $(m-k)$ rows of coefficients of f and $(n-k)$ rows of coefficients of g. It has the following form:

$$\mathrm{Syl}_k(f,g) = \begin{pmatrix} a_n & \cdots & \cdots & a_0 & & \\ & \ddots & & & \ddots & \\ & & a_n & \cdots & \cdots & a_0 \\ b_m & \cdots & \cdots & b_0 & & \\ & \ddots & & & \ddots & \\ & & b_m & \cdots & \cdots & b_0 \end{pmatrix}$$

  These matrices occur when asking for (non)-zero polynomials u, v with $\deg(u) < m-k$ and $\deg(v) < n-k$ and fulfilling $uf + vg = 0$. It corresponds to the linear system of equations $(u,v)\mathrm{Syl}_k(f,g) = 0$, where u and v are identified with their coefficient vector.

- For $0 \leq k \leq n$, the *k-th subresultant of f and g* is defined as

$$\mathrm{Sres}_k(f,g) := \begin{cases} \sum_{i=0}^{k} M_i^k(f,g) t^i & k \leq m-1 \\ g & k = m \\ 0 & m+1 \leq k < 1 \\ f & k = n \end{cases}$$

  where $M_i^k(f,g)$ is the determinant of the matrix build with the first $n+m-2k-1$ and the $(n+m-k-i)$th column of $\mathrm{Syl}_k(f,g)$.[4]

- The *k-th principal subresultant coefficient*, $0 \leq k \leq n$, is given by

$$\mathrm{sres}_k(f,g) := \begin{cases} \mathrm{coef}_k(\mathrm{Sres}_k(f,g)) & 0 \leq k < n \\ 1 & k = n \end{cases}$$

- The *k-th coprincipal subresultant coefficient*, $1 \leq k \leq n$, is given by $\mathrm{cores}_k(f,g) := \mathrm{coef}_{k-1}(\mathrm{Sres}_k(f,g))$.

- The *subresultant sequence of f and g* is given by $\mathrm{Sres}_n(f,g), \ldots, \mathrm{Sres}_0(f,g)$. Similar sequences exists of sres and cores.

- It holds $\mathrm{Sres}_0(f,g) = \mathrm{sres}_0(f,g) =: \mathrm{Res}(f,g)$, where $\mathrm{Res}(f,g)$ states the *resultant of f and g*. If $g = f'$, then we call $\mathrm{Res}(f,f')$ the *discriminant* of f.

- We also write $\mathrm{Res}_{t_d}(f,g)$, which especially makes sense, if $\mathbb{D}$ itself is a polynomial ring, that is, we consider f and g as univariate polynomials in some t_d, whose coefficients can be themselves polynomials in other variables. Similar for $\mathrm{Sres}_{t_d,k}$, $\mathrm{sres}_{t_d,k}$, and $\mathrm{cores}_{t_d,k}$.

[4] This definition of the subresultant is different from the standard literature (e. g., [BPR06]). It is presented in [Ker06].

We next state without proofs results relevant for our work, where f and g are polynomials as in Definition 2.6.

Proposition 2.7. *The resultant* $\mathrm{Res}(f,g) \in \mathbb{K}$ *is zero if and only if* f *and* g *have a non-constant common factor, that is, for* $h = \gcd(f,g)$*, it holds* $\deg(h) > 0$.

If $\mathbb{K} = \mathbb{C}$, $\mathrm{Res}(f,g) = 0$ *holds if and only if* f *and* g *have a common complex root.*

For details on this proposition we refer to [Ber04, Proposition 2.1.14]. Observe, that in any case it holds $\mathrm{Res}(f,g) \in \mathbb{K}$. Thus, the complexity of the problem has been reduced with respect to dimensionality. On the other hand, $\mathrm{Res}(f,g)$ is an expression of complexity $O(m \cdot n)$. In particular, if $\mathrm{Res}(f,g)$ is a polynomial again: Consider, for example, $\mathrm{Res}_{t_2}(f,g) \in \mathbb{D}[t_1]$. Then, it holds $\deg_{t_1}(\mathrm{Res}_{t_2}(f,g)) = n \cdot m$ (with $n = \deg_{t_2}(f)$ and $m = \deg_{t_2}(g)$).

For the case that $f, g \in \mathbb{K}[t_1, \ldots, t_d]$, *elimination theory* paves a way to compute a zero-dimensional solution for $f = g = 0$ by reduction of dimension. We first compute a partial solution $\alpha_1, \ldots, \alpha_{d-1}$ which is being extended in a second step by all possible full solutions $\alpha_1, \ldots, \alpha_d$. It is obvious that the method should be applied recursively. The claim is, that the solutions to $\mathrm{Res}_{t_d}(f,g)$ constitute a set of partial solutions that can be extended. However, this is broken if for such a solution $\alpha_1, \ldots, \alpha_{d-1}$, we have that $\mathrm{lcf}_{t_d}(f)(\alpha_1, \ldots, \alpha_{d-1}) = 0$ and $\mathrm{lcf}_{t_d}(g)(\alpha_1, \ldots, \alpha_{d-1}) = 0$. In this case, $\mathrm{Res}_{t_d}(f,g)(\alpha_1, \ldots, \alpha_{d-1})$ vanishes ignoring the fact whether $\alpha_1, \ldots, \alpha_{d-1}$ is a partial solution or not. The reason is that the first column of the Sylvester matrix completely vanishes. However, $\mathrm{Res}_{t_d}(f,g)(\alpha_1, \ldots, \alpha_{d-1}) = 0$ is a necessary condition for $\alpha_1, \ldots, \alpha_{d-1}$ being an extendible partial solution. The problem becomes handy if f or g is t_d-regular.

Proposition 2.8. *Let* $\mathbb{K}$ *be a field, and let* $f, g \in \mathbb{K}[t_1, \ldots, t_d]$ *be non-zero polynomials. Furthermore, let* f *be* t_d*-regular. Then, for all* $(\alpha_1, \ldots, \alpha_{d-1}) \in \overline{\mathbb{K}}^{d-1}$ *the two conditions*

1. $\mathrm{Res}_{t_d}(f,g)((\alpha_1, \ldots, \alpha_{d-1})) = 0$
2. *There is* $\alpha_d \in \overline{\mathbb{K}}$ *such that* $f(\alpha_1, \ldots, \alpha_d)) = g((\alpha_1, \ldots, \alpha_d)) = 0$

are equivalent; see also [Ber04, Proposition 2.1.13].

Cylindrical algebraic decomposition (see §2.1.6 on page 44 f) mainly uses terms introduced in Definition 2.6 and Propositions 2.7 and 2.8 to project an algebraic problem to an instance of lower dimensionality. In §2.1.4 and §2.3.3 the technique is used to analyze algebraic curves, and in Chapter 5 we also rely on dimension reduction to analyze algebraic surfaces.

There is a relation of the greatest common divisor and the subresultant sequence.

Proposition 2.9 ([BPR06, Prop. 10.14, Cor. 10.15]).

- $\deg(\gcd(f,g)) = \min\{k \in \{0, \ldots, n\} \mid \mathrm{sres}_k(f,g) \neq 0\}$
- $\mathrm{Sres}_k(f,g) \sim \gcd(f,g)$
  ($h_1 \sim h_2$ denotes that either $h_1 = c \cdot h_2$ or $c \cdot h_1 = h_2$, for h_1 and h_2 polynomials over some $\mathbb{D}$ and $c \in \mathbb{D}$.)

One can even show, that the subresultant sequences contains (up to associates) all polynomials occurring during the Euclidean algorithm to compute the gcd, but with less complexity of the coefficients [BPR06, § 8.2]. Proposition 2.9 implies the following two algorithms on polynomials.

Algorithm 2.1. Computing greatest common divisor with subresultants

INPUT: $f, g \in \mathbb{K}[t]$ as in Definition 2.6
OUTPUT: $\gcd(f,g) \in \mathbb{K}[t]$

- $k \leftarrow 0$
- While $(\text{sign}(\text{sres}_k(f,g)) = 0)$ Do $k \leftarrow k+1$
- Return $\text{Sres}_k(f,g)$

Algorithm 2.2. Computing square-free part of a polynomial using subresultants

INPUT: $f \in \mathbb{K}[t]$ as in Definition 2.6
OUTPUT: $f^\star \in \mathbb{K}[t]$ that contains each distinct factor of f once.

- Compute $h = \gcd(f, f')$ with Algorithm 2.1
- Return f/h

The subresultant is robust with respect to ring homomorphisms $\varphi : \mathbb{D} \to \mathbb{D}'$ that are degree-preserving for f and g. Then, $\forall i : \varphi(\text{Syl}_i(f,g)) = \text{Syl}_i(\varphi(f), \varphi(g))$, where $\varphi(A)$ means to apply φ to each entry of A (see, e. g., Algorithm 2.4). As the determinant is just a sum of products, we have $\varphi(\det(A)) = \det(\varphi(A))$, which proves the following theorem (see also [Yap00, §4.4, Lemma 4.9]).

Theorem 2.10 (Specialization property). *Given a homomorphism of domains $\varphi : \mathbb{D} \to \mathbb{D}'$, with $\text{lcf}(f), \text{lcf}(g) \notin \ker(\varphi)$. Then, for $0 \le i \le n$, $\varphi(\text{Sres}_i(f,g)) = \text{Sres}_i(\varphi(f), \varphi(g))$.*

There is a main application which explains the name of the theorem. Think of $\mathbb{D} = D[t]$ for some basic domain D, that is, $\mathbb{D}$ has the parameter t. There is a simple homomorphism to D that specializes t to some value α. Then, instead of $\text{Sres}_i(f|_{t=\alpha}, g|_{t=\alpha})$, it is possible to access $\text{Sres}_i(f,g)|_{t=\alpha}$. Actually, the number of parameters is free, and a homomorphism can specialize all of them, or just a subset.

In Algorithm 2.2 we set $g = f'$. It is easy to see, that applying this idea in general to the given sequences, allows to obtain interesting information on the multiple factors of a single polynomial f. The Sturm-Habicht sequence, that we introduce next, is another sequence that derives even more information for such a f. Actually, the sequence can also be defined for arbitrary g, from which we abstain, as we are aiming to only introduce the tools relevant for subsequent chapters.

Definition 2.11 (Sturm-Habicht sequence [GVRLR]). Given $f = \sum_{i=0}^{n} a_i t^i \in \mathbb{K}[t]$ with $n = \deg(f)$, and $\delta_k := (-1)^{k(k+1)/2}$. For $k \in \{0, \ldots, n\}$, the *k-th Sturm-Habicht polynomial* of f is defined as

$$\begin{aligned} \text{StHa}_n(f) &:= f \\ \text{StHa}_{n-1}(f) &:= f' \\ \text{StHa}_k(f) &:= \delta_{n-k-1}\text{Sres}_k(f,f'), \quad k = 0, \ldots, n-2 \end{aligned}$$

We define $\text{stha}_k(f)$, the *k-th principal Sturm-Habicht coefficient* of f, as the coefficient of t^k in $\text{StHa}_k(f)$.

In [BPR06] the Sturm-Habicht sequence is introduced as *signed subresultant sequence*, which reflects that the Sturm-Habicht sequence basically coincides with the subresultant

sequence, but whose members are possibly multiplied by -1. This slight difference has no implication on the specialization property. That is, a Sturm-Habicht (coefficient) sequence still behaves well under specialization. On the other side, the (possible) multiplication by -1 makes a difference, as Sturm-Habicht sequences allow to compute the number m of distinct real roots of f in a given interval $[c, d]$ without actually actuating a real root isolator (to be presented in §2.1.2). In fact, that section describes an isolator that decisively relies on this information. The theoretical result that allows the compute m is stated with a full proof in [GVN02], while the version in [EKW07] is restricted to $I =]-\infty, \infty[$. Instead of the theorem, we give an algorithm.

Algorithm 2.3. Computing the number of distinct real roots using Sturm-Habicht sequence

INPUT: $f \in \mathbb{R}[t]$, with $\deg(f) = n > 0$

OUTPUT: The number m of distinct real roots of f

1. Compute the sequence $S = s_0, \ldots, s_n$ with $s_i := \operatorname{sign}(\operatorname{stha}_i(f))$. Observe, that $\operatorname{stha}_i(f) \in \mathbb{R}$.
2. $m \leftarrow 0$
3. For each subsequence $S' = (a, (0)_{0\ldots k}, b)$ of S with $a \neq 0, b \neq 0$ and $k \geq 0$ Do
   - If k even, then $m \leftarrow m + (-1)^{k/2}\operatorname{sign}(ab)$
4. Return m

Besides the number of distinct real roots, we are also interested in multiple roots. In that direction, Proposition 2.9 states a fundamental result used in Algorithm 2.1 to compute an important information, namely the degree k of $\gcd(f, f')$. By the definitions of StHa_i and stha_i, it is easy to see, that Algorithm 2.1 still computes the correct k, if sres_i is replaced by stha_i and Sres_i by StHa_i.

Remark. If both m and k are desired, it is recommended to first compute m with Algorithm 2.3 and then to reuse the sequence $S = s_0, \ldots, s_n$ in Algorithm 2.1 which gives k as side-effect: Namely, when searching for the minimal k with $\operatorname{stha}_k = 0$. A clever combination of the two algorithms allows to obtain k with no additional costs on top of the expenses of Algorithm 2.3.

Computing $gcd(f, f')$ in the second part of modified Algorithm 2.1 still needs StHa_k for the given k, and computing $f^\star$ needs the subsequent division in Algorithm 2.2. However, the cofactors of the Sturm-Habicht polynomials already contain $f^\star$ [BPR06, Prop. 8.38].

Proposition 2.12. *For $j < n$, there exist polynomials u_j, v_j with $\deg(u_j) \leq n - j - 2$, $\deg(v_j) \leq n - j - 1$ such that $\operatorname{StHa}_j(f) = u_j f + v_j f'$.*

All cofactors u_j and v_j can be written as determinants of Sylvester-like matrices. The square-free part $f^\star$ of f is given by one of the v_j's [BPR06, Prop. 10.14, Cor. 10.15].

Lemma 2.13. *If $k = \deg(\gcd(f, f')) > 0$, then $f^\star = v_{k-1}$.*

Algorithm 2.4. Computing square-free part of a polynomial using Sturm-Habicht sequence

INPUT: $f \in \mathbb{K}[t]$ as in Definition 2.6
OUTPUT: $f^\star \in \mathbb{K}[t]$ that contains each distinct factor of f once.

- $k \leftarrow 0$
- While $(\mathrm{stha}_k(f,g) = 0)$ Do $k \leftarrow k+1$
- Return v_{k-1} as stated in Lemma 2.13

An algorithm to compute a Sturm-Habicht sequence with cofactors is [BPR06, Alg. 8.22]. In addition, it is more efficient to prefer a polynomial remainder sequence [Loo82a] than computing the Sturm-Habicht sequence via determinantal expressions.

Subresultant and Sturm-Habicht sequences in combination with their specialization property are key tools when analyzing algebraic objects of higher degree. We present further basics on this in §2.1.2, while §2.1.4 introduces algebraic curves and §2.1.4 algebraic surfaces. Chapter 5 presents how to analyze algebraic surfaces in the spirit as previously done for algebraic curves [EKW07],[EK08a]. Both cases still require some exact computations, that is, launching algorithms that we presented in this section.

2.1.2. Algebraic numbers and real root isolation

Definition 2.14 ((Real) algebraic number). Let $\mathbb{K}$ be a field, and $f(t) \in \mathbb{K}[t]$. We call an element α with $f(\alpha) = 0$, an *algebraic number over* $\mathbb{K}$. It is called *real algebraic number* if $\alpha \in \mathbb{R}$. If f is irreducible over $\mathbb{K}$ (i. e., f cannot be expressed in the form $f = f_1 f_2$, with $f_1 \neq 1$ and $f_2 \neq 1$), then we call f the *minimal polynomial* of α. The other roots $\overline{\alpha} \neq \alpha$ of the minimal polynomial are the *conjugates* of α. The *degree* of α is defined by the degree of the minimal polynomial. If f is reducible, there always exists a minimal polynomial that is a factor of f, and defines the degree.

In our geometric applications, we focus on the case $\mathbb{K} = \mathbb{R}$. For proofs, we sometimes also have to refer to the complex roots of a polynomial. An important property is, that the roots of polynomial f with algebraic numbers as coefficients are also algebraic numbers. In the remainder of this part we shortly discuss how to represent (real) algebraic numbers, how to compare two of them, and how to isolate the real roots of a univariate polynomial.

Representation, comparison, evaluation

An algebraic number can be expressed in form of an algebraic expression E formed by a directed acyclic graph whose leaves are integers, and whose inner nodes define operations on their children. Allowed operations are $+$, $-$, $\cdot$, $/$, $\sqrt[k]{}$, and $\diamond$. The expression $\diamond(j, E_d, \ldots, E_0)$ identifies the $1 \leq j \leq d$ root of the polynomial $\sum_{i=0}^{d} \mathrm{val}(E_i) t^i$, where $\mathrm{val}(E)$ is the real value given by the expression E.[5] Each node knows an interval approximation of the exact value defined by its subgraph, which can be refined by recursively approximating the values with higher precision. An operation is applied by creating a new root node, connecting it to the graph, and by computing a first approximation. The comparison of two such numbers is reduced to the computation of the sign of a difference. If the approximation interval of the difference does not contain zero, the answer is simple. Otherwise, a *separation bound*

[5] Note that $\diamond$ actually subsumes all other operations.

is computed, that is, a value $\overline{E}$ with the property that $\mathrm{val}(E) \neq 0 \Rightarrow |\mathrm{val}(E)| \geq \overline{E}$. This means, that an expression E is either zero, or has a minimal absolute value. Thus, the correct sign is achieved by refining the approximation until the absolute values of both ends are smaller (or greater) than $\overline{E}$, which allows to decide the sign. The theory on separation bounds is wide-spreaded. We refer to [LY01] and [BFM+01] for further reading and to §3.3 (page 97 ff) where we utilize corresponding number types. For the next representation we need a term.

Definition 2.15 (Isolating interval). Let f be a univariate polynomial with a root $\alpha \in \mathbb{R}$. A closed interval $[a, b] \subset \mathbb{R}$ containing α, but no other root of f, is called an *isolating interval for α with respect to f*. Containing means that either $a = \alpha = b$ or $a < \alpha < b$.

This section contains a brief overview on algorithms that isolate all real roots of a polynomial, while for now, we state without proofs, that for each real root, there exists such an isolating interval, which even can be refined to arbitrary small length (if not already degenerate) in a sequence of nested intervals. Such an interval is a key ingredient to represent a real algebraic number α over $\mathbb{K}$; see Definition 2.16. Usually, we have $\mathbb{K} = \mathbb{Z}$. Observe that such a number is also algebraic over $\mathbb{Q}$. So, we restrict to the integral case for the following definition.

Definition 2.16 (Integral interval representation). Let α be a real algebraic number that is a root of $f \in \mathbb{Z}[t]$ having an isolating interval $I = [a, b]$. We call $\alpha \widehat{=} (f; I)$ an *(integral) interval representation* of α. The representation is *simple*, if α is a simple root of f.

Note that the representation uniquely identifies the root, though neither the polynomial nor the interval is unique. Arithmetic on this representation is not directly supported, but also not desired. Its main purpose is to represent, to refine, and to compare real algebraic numbers. Definition 2.17 gives a more generic representation for a certain set of algebraic numbers over $\mathbb{K} = \mathbb{R}$. Some of our intended applications require them.

For the interval boundaries, one usually chooses $a, b \in \mathbb{Q}$, as $\mathbb{Q}$ is dense in $\mathbb{R}$. However, every set that is dense in $\mathbb{R}$ is possible. For a simple representation of $\alpha \widehat{=} (f; [a, b])$, we have $f(a)f(b) < 0$. This directly implies a bisection method to refine I: Namely, I is replaced by $I_\ell = [a, \frac{a+b}{2}]$ or $I_r = [\frac{a+b}{2}, b]$, depending on the sign of $f(\frac{a+b}{2})$. This strategy allows to refine an isolating interval with linear convergence. An alternative with quadratic behavior is due to Abbot [Abb06]. Algorithm 2.5 gives a high-level description of a method to compute the order of two such representations.

Algorithm 2.5. Compare two simple interval representations

INPUT: $\alpha_1 \widehat{=} (f_1; I_1)$; $\alpha_2 \widehat{=} (f_2; I_2)$, both simple
OUTPUT: Order of α_1 and α_2

- If I_1 and I_2 are disjoint, we return the order and are done.
- Compute $I = I_1 \cap I_2 = [a, b]$
- Check if I is isolating for α_1 and α_2 by determine the signs of $f_1(a), f_1(b)$ and $f_2(a), f_2(b)$. If not, we refine I_1 and I_2 until they are disjoint, which gives the order.
- Otherwise, compute $g = \gcd(f_1, f_2)$ and check whether $g(a)$ and $g(b)$ have different signs. If so, I is isolating for a common root of f_1 and f_2, which gives $\alpha_1 = \alpha_2$.
- If not, refine I_1 and I_2 until they are disjoint, which gives the order.

There are subtleties in the refinement and comparison that must be considered, for example, an occurring zero sign (in some $f_i(a)$ or $f_i(b)$). However, we omitted them for simplicity. A similar algorithm is used to compute the sign of a polynomial at a given interval representation.

Algorithm 2.6. Computing sign of a polynomial at simple interval representation

INPUT: $g \in \mathbb{Z}[t]$, square-free; $\alpha \widehat{=} (f; I)$, simple with $f \in \mathbb{Z}[t]$
OUTPUT: $\operatorname{sign}(g(\alpha))$

- Compute $h = \gcd(f, g)$ and check whether $h(a)$ and $h(b)$ have different signs. If so, return 0.
- Compute $J = [c, d] = h(I)$ with interval arithmetic (see §2.3.1 on page 53).
- If $\operatorname{sign}(c) = \operatorname{sign}(d)$, return $\operatorname{sign}(c)$.
- Otherwise, refine I to I' and restart with the computation of a new J.

There are further direct representations of real algebraic numbers, like *Thom's encoding* [BPR06]. However, we do not go into the details. Usually, we make use of the isolating interval representation.

Remember that the roots of a univariate polynomial with algebraic coefficients are algebraic again. One way to obtain such a polynomial is to evaluate a d-variate polynomial of rational coefficients with $d-1$ algebraic numbers. For example, let α_1, α_2 be real algebraic numbers, $f \in \mathbb{Q}[t_1, t_2]$, $g \in \mathbb{Q}[t_1, t_2, t_3]$, then $f_{\alpha_1}(t_2) := f(\alpha_1, t_2) \in \mathbb{R}[t_2]$ and $g_{\alpha_1,\alpha_2}(t_3) := g(\alpha_1, \alpha_2, t_3) \in \mathbb{R}[t_3]$ are such polynomials. We introduce a more generic representation for such real algebraic numbers over $\mathbb{R}$.

Definition 2.17 (Algebraic interval representation). Let $d > 1$ be some dimension, $f \in \mathbb{Z}[t_1, \ldots, t_d]$, primitive, and $\overline{\alpha}^{(d-1)} = (\alpha_1, \ldots, \alpha_{d-1}) \in \mathbb{R}^{d-1}, \beta \in \mathbb{R}$, where α_1 is in interval representation, while α_i with $i > 1$ is recursively defined with $\overline{\alpha}^{(i-1)}$. Remember that $f_{\overline{\alpha}^{(d-1)}}(t_d) := f(\alpha_1, \ldots, \alpha_{d-1}, t_d) \in \mathbb{R}[t_d]$.

If $f_{\overline{\alpha}^{(d-1)}}(\beta) = 0$ and $I = [a, b] \subset \mathbb{R}$ is isolating for β, we call $\beta \widehat{=} (f; \overline{\alpha}^{(d-1)}; I)$ an *algebraic interval representation (of dimension d) of* β. Again, β is *simple* if it is a simple root of $f_{\overline{\alpha}^{(d-1)}}$.
Additional remarks:

- α_i with $1 < i < d$ is an algebraic interval representation of dimension i at $\alpha^{(i-1)}$, namely $\alpha_i \widehat{=} (f_i; \overline{\alpha}^{(i-1)}; I_i)$
- We call the collection of numbers $\overline{\alpha}^{(d-1)}$ a *base point (of dimension* $(d-1)$*)* and refer to the polynomial of shape $f_{\overline{\alpha}^{(d-1)}}(t_d)$ as a *lifting polynomial at the base point* $\overline{\alpha}^{(d-1)}$.

We should mention that there are methods to convert an algebraic interval representation into an integral interval representation [Loo82a]. Although this allows to directly apply Algorithms 2.5 and 2.6, we abstain for reasons of efficiency to deploy this strategy.

Instead, we pursue an indirect approach in order to compare two algebraic interval representations or to compute the sign of a polynomial at an algebraic interval representation. In fact, it turns out that the sign determination is key when refining isolating intervals. Recall that iterated refinements of the isolating intervals suffice to decide the order of two non-equal numbers. Thus, before explaining how to decide equality for two algebraic interval representations, we first consider how to refine the interval I of a given $\beta \widehat{=} (f; \overline{\alpha}^{(d-1)}; I)$. Below, we present a methods to isolate the real roots of a square-free

polynomial based on Descartes' rule of sign. It provide as by-product a possibility to refine such intervals. The following is more direct, but also holds only if $f_{\overline{\alpha}^{(d-1)}}$ is square-free: Deploying the bisection approach in order to refine I reduces to compute three signs, namely $\text{sign}(f_{\overline{\alpha}^{(d-1)}}(r))$, where $r \in \{a, \frac{a+b}{2}, b\}$. By defining $g_r \in \mathbb{Z}[t_1, \ldots, t_{d-1}]$ as the integralized version of $f(t_1, \ldots, t_{d-1}, r)$ (mind that integralizing keeps roots and signs), the remaining problem is to compute $\text{sign}(g_r(\alpha_1, \ldots, \alpha_{d-1})$. The following algorithm is a recursive version of Algorithm 2.6 exploiting the fact, that α_i depends on $\overline{\alpha}^{(i-1)}$.

Algorithm 2.7. Computing the sign of a polynomial at algebraic interval representations

INPUT: $g \in \mathbb{Z}[t_1, \ldots, t_d]$; $\overline{\alpha}^{(d)} = (\alpha_1, \ldots, \alpha_d)$, forming a sequence of algebraic interval representations where $\alpha_1 \widehat{=}(f_1; I_1)$ and $\alpha_i \widehat{=}(f_i; \overline{\alpha}^{(i-1)}; I_i)$ for $1 < i \leq d$. Observe that $f_i \in \mathbb{Z}[t_1, \ldots, t_i]$, and $I_i = [a_i, b_i] \subset \mathbb{R}$

OUTPUT: $\text{sign}(g(\alpha_1, \ldots, \alpha_d))$

- Let $f := f_d$. Compute $h = \gcd(f_{\overline{\alpha}^{(d-1)}}, g_{\overline{\alpha}^{(d-1)}})$.
- Compute (recursively) the signs of $h(a_d)$ and $h(b_d)$. If they have different signs, return 0.
- Compute $J = [c, d] = h(I_d)$ with interval arithmetic using all $I_j, 1 \leq j \leq d$.
- If $\text{sign}(c) = \text{sign}(d)$, return $\text{sign}(c)$.
- Otherwise, refine I_d to I'_d and restart with the computation of a new J.

At two positions the recursion takes place, namely when determining the signs of $h(a_d)$ and $h(b_d)$, and when refining I_d to I'_d. In addition, the algorithm makes an assumption that we have not yet proposed a solution for. It assumes that $\gcd(f_{\overline{\alpha}^{(d-1)}}, g_{\overline{\alpha}^{(d-1)}})$ can be computed. Theoretically, using the standard Euclidean algorithm, this task does not pose a problem. However, the demanded operations on such algebraic coefficients of large degree are simply infeasible, in particular, for arbitrary polynomials. The solution we propose relies on the fact, that both polynomials $f_{\overline{\alpha}^{(d-1)}}$ and $g_{\overline{\alpha}^{(d-1)}}$ are lifting polynomials at the same base point $\alpha^{(d-1)}$. Algorithm 2.8 enhances Algorithm 2.1 with the specialization property to compute $\gcd(f_{\overline{\alpha}^{(d-1)}}, g_{\overline{\alpha}^{(d-1)}})$.

Algorithm 2.8. Computing greatest common divisor with specialized subresultants

INPUT: $f, g \in \mathbb{Z}[t_1, \ldots, t_d]$; $\overline{\alpha}^{(d-1)} \in \mathbb{R}^{d-1}$

OUTPUT: $\gcd(f_{\overline{\alpha}^{(d-1)}}, g_{\overline{\alpha}^{(d-1)}}) \in \mathbb{R}[t_d]$

- $k \leftarrow 0$
- While $(\text{sign}(\text{sres}_k(f, g)(\overline{\alpha}^{(d-1)})) = 0)$ Do $k \leftarrow k + 1$
- Return $\text{Sres}_k(f, g)_{\overline{\alpha}^{(d-1)}} \in \mathbb{R}[t_d]$

We finally mention, that Algorithm 2.8 is also launched when computing the gcd in a modified version of Algorithm 2.5 in order to decide whether $\alpha \widehat{=}(f; \overline{\gamma}; I)$, and $\beta \widehat{=}(g; \overline{\gamma}; J)$ are equal.[6]

While Definition 2.17 is generic, we are restricted in this thesis to utilizations for dimensions 2 and 3 only. However, we still need to know how to compute the isolating intervals for integral and algebraic interval approximations. The theory on real root isolation is discussed next. Technical details on how to use algebraic interval representations

[6] The input now consists of two algebraic interval representations replacing the integral ones.

of dimension 2 to represent y-coordinates of algebraic curves are later given in §2.3.3 and §2.3.4. Chapter 5 discusses how to advance those ideas by one dimension such that we are able to represent z-coordinates for points on an algebraic surfaces by algebraic interval representations of dimension 3.

Real root isolation

Isolating the real roots of a univariate polynomial of arbitrary degree is a well-studied problem in (computational) algebra. Although not at the heart of the thesis, its central contributions rely on previous work in this field. In Chapter 5 we even face real root isolation concretely, when computing algebraic interval representations of dimension 3. The method that we rely on is the well-known Descartes method [CA76]; there are variants for inexact coefficients [EKK+05], and a modification of it [EKW07]. The technique is comprehensively discussed in [Eig08], to which we also refer for its encyclopedic description of other root isolations, for example, numerical solvers, the method based on continued fractions, and the subdivision scheme using Sturm sequences, as well as all their variants. To discuss all of them lies beyond the scope of this thesis. Thus, we only extract important information of the parts on the Descartes method from [Eig08], that also contains missing details in the presentation.

Before getting deeper into it, we should mention that most approaches, as well as the Descartes method, require the input polynomial to be square-free. If not, we have two options. The first consists of computing f's square-free part $f^\star$ either using $f^\star = \frac{f}{gcd(f,f')}$ or by deploying subresultants (as in Algorithm 2.2 or Algorithm 2.4) followed by a subsequent restart. As second possibility, we square-free factorize, and apply the real root isolator to each of the factors. In that approach, a subsequent sorting of the roots is often expected, which requires comparisons. On the other hand, later computations may benefit from the fact that defining polynomials are of smaller degree. Note that we do not compute the minimal polynomial for an interval representation. However, it is possible to interactively replace the defining polynomial by a simpler one, namely in the case that the gcd in Algorithm 2.5 is non-trivial.

The basic idea of the *Descartes methods* is to consider initially an interval that contains all roots and to repeatedly subdivide it until we are left with a situation where each interval is guaranteed to contain either no or exactly one root.

Theorem 2.18 (Descartes' rule of signs). *Let $f = \sum_{i=0}^{n} a_i x^i$ and $V(f)$ be the number of sign changes in $(a_n, \ldots, a_0)$ (ignoring $a_i = 0$). Let $\alpha_1, \ldots, \alpha_r$ be the positive real roots of f with multiplicities $m_1, \ldots, m_r$. Let $M^+ = \sum_{j=0}^{r} m_j$.*

Then $V(f) - M^+$ is non-negative and even.

For a proof we refer to [BPR06]. Using a Möbius transformation Descartes' rule of signs also gives a bound on the number of real roots of the polynomial f within an interval I. We denote this bound by $V(f; I)$. More details appear in [RZ03]; a variant using the Bernstein basis is presented in [HL93]. This basis has advantages with respect to splitting intervals. This splitting is essential in the following Algorithm 2.9 for real root isolation.

Algorithm 2.9. Real root isolation with bound on number of roots

INPUT: $f \in \mathbb{Z}[t]$, square-free
OUTPUT: list of disjoint intervals, with as many real roots of f than intervals, each containing exactly one real root of f

- Compute I_0 containing all real roots, and initialize a container Q with I_0
- While Q is not empty,
  - Pop an interval I from Q, compute $V(f;I)$.
  - If $V(f;I) > 1$, subdivide I into I_{left} and I_{right} and add them to Q.
  - If $V(f;I) = 1$, return I.
  - If $V(f;I) = 0$, remove I from Q

Remarks (on Algorithm 2.9).

- The algorithm works with any subroutine that correctly computes $V(f;I)$. Using Descartes' Rule of Sign is our preferred method.
- Computing a good I_0 is a problem on its own. Several bounds are known and we refer to [Eig08, §2.4] for a collection of some.
- The algorithm simplifies, as it does not check whether the boundaries of intervals are roots of f. However, for this introduction, we can assume, that no such root exists. Algorithmically, it can be handled by either explicitly checking whether the boundaries are roots. There also exists techniques, like random perturbations of the polynomial's coefficients, that still ensure the correctness of the computed isolating intervals for the original real roots.
- The algorithmic description misses to give a strategy on how elements of Q are popped, which actually does not play a role for the effectivity of the approach; but maybe affects the efficiency.
  More detailed, we can see Q as a subdivision tree. If naively traversed with a depth-first search strategy, its number of nodes (also measured in depth of the tree) can exceed a value that is linear bounded by $\deg(f)$. The situation slightly improves by firstly performing the Descartes test on a subdivided interval I', and make it a child of the tree ("put it into Q") only if $V(f;I') > 1$. In contrast, a breadth-first search ensures that the number of nodes in each depth of the tree is linear bounded by $\deg(f)$. Breadth-first search is crucial if a depth-dependent counting argument on the $V(f;I)$ becomes another criterion. The m-k-variant that we present below contains such a criterion.

Polynomials with inexact coefficients We require so far that the coefficients of f are from a subring $R \subset \mathbb{R}$ which can be handled exactly, for example the rational numbers. Thus, we refer to this approach as the *exact Descartes method (EDM)*. The expectation on coefficients to be given exact can be relaxed in some sense: The simple roots of a (square-free) polynomial continuously depend on the polynomial's coefficients. When perturbing f's coefficients by some (small) ε, an implication is: If I is an isolating interval for a simple root of f_ε then I is also isolating for a simple root of f, for sufficiently small ε. This fact opens the door for variants of algorithms for real root isolation: One alternative evaluates the Descartes test for coefficients that are expressed by interval approximations (e. g., [CJK02], [RZ03], [MRR05]). A generalization of this approach is given by the *bitstream Descartes method (BDM)* [EKK$^+$05]. It assumes that the coefficients of a polynomial f are given as potentially infinite bit-streams, that is, coefficients are known to arbitrary

precision, but, in general, never exactly. The coefficients are interfaced to the BDM by repeatedly asking for more bits, that is, it is required to compute a binary representation of a coefficient of arbitrary precision. We later give the technical interface in §2.3.4. As long as a Descartes test fails to determine the correct number of sign changes for a certain precision, the method demands for a better approximation, and restarts the test. A clever combination of the subdivision and the evaluations ensures that each coefficient is not too much over-approximated, which would directly lead to a slump in the overall performance.

A rather simple application of the adaptive precision is possible even for the exact setting. In contrast to interfacing a (possible) lengthy exact representation, we only provide an increasing number of initial bits, until the BDM is successful. Especially, for polynomials with nice separations bound, the bitstream version of the test succeeds with using less bits than for the exact version. But actually, the BDM exactly fits the needs for polynomials whose coefficients might be transcendental or arbitrary algebraic numbers. A very suitable example is the polynomial $f_{\overline{\alpha}^{(d-1)}}$ introduced in Definition 2.17. Using interval arithmetic (see §2.3.1) and the refineable representation of each involved $\alpha_1, \ldots, \alpha_{d-1}$, it is possible to compute a refineable interval approximation of $f_{\overline{\alpha}^{(d-1)}}$'s coefficients. Thus, the BDM is a very elegant way of computing the isolating intervals for the real roots of $f_{\overline{\alpha}^{(d-1)}}$. A usage for dimension 2 is given in §2.3.4 (page 64f), while we augment this approach for algebraic interval representations of dimension 3 in §5.4.2.

Refining intervals with Descartes However, isolating the real roots of such a polynomial is not the sole applicability of the (bitstream) Descartes method, or, actually, Descartes' rule of signs. Consider a leaf of the (implicit) subdivision tree with $V(f; I) = 1$ for its interval I. In order to refine I, it is only required to subdivide further, and keep the half I' for which Descartes' rule of signs still reports $V(f; I') = 1$. Of course, an actual implementation should avoid any further Möbius transformation of the polynomial to compute $V(f; I_{\text{left}})$ or $V(f; I_{\text{right}})$. These numbers are already known to be either 1 or 0 and sum up to 1. The 0-interval is discarded for our desired refinement. The required sign computation becomes more expensive with decreasing interval length, and more bits from the streams are expected. On the other hand, this approach naturally enhances the already required root isolation algorithm — in contrast to the pure bisection approach presented for algebraic interval representations that relies on exact sign computations. A nice interface for isolating and refining the reals roots of a (bitstream) polynomial is presented in §2.3.4 (page 64).

Remark. We remember again, that iterated refinements cannot decide the equality of two such isolated roots. This goal still requires symbolic computations, as we exemplary presented for algebraic interval representations that use specialized subresultants; see Algorithm 2.8.

Not each polynomial is square-free Assume a polynomial f being not square-free, and α being a multiple real root of f. It is easy to see, that the Descartes method in general, and the bitstream Descartes method in particular, do not terminate when executed on such an f. The reason is that the multiplicity of α is at least 2. Thus, for each interval I containing α it holds $V(f; I) \leq 2$, such that no (further) subdivision can lead to an I for which one of the two termination conditions of Algorithm 2.9 applies.

An important step into this direction has been made by the *m-k-Descartes method*

proposed by [EKW07]. It allows to isolate the real roots of a polynomial f that contains at most one multiple root, or, otherwise, reports the existence of more than one multiple root. The variant actually runs a usual Descartes algorithm, and for simplicity we do not distinguish the differences between the exact and the bitstream version in the following. The method is oblivious of the fact, that if f has a multiple root, the container Q never becomes empty. However, it is fed with additional knowledge on f, namely the *number m of distinct real roots*, and the *degree k of* $\gcd(f, f')$. We have presented in §2.1.1 how to compute these values. Utilizing these pieces of information, the m-k-variant interrupts the execution of the running Descartes method if one of two conditions is satisfied:

1. There are exactly $m-1$ intervals in Q indicating a simple root.
2. For all intervals I in Q it holds that $V(f;I) \leq k$.

[EKW07] state that the variant terminates with either of the two conditions. Intuitively, if f has at most one multiple root the first condition is eventually satisfied. In this case, the m-k-variant stops with success, while $V(f;I)$ for the single remaining interval I only states an upper bound of the multiple root's multiplicity with correct parity. Thus, the odd case still can transform to a simple root. It depends on the inquiring application how to deal with this restricted information. In case f has more than one (complex) multiple root, none of their multiplicities can reach k. However, for a sufficiently small interval I containing an r-fold root, it holds that $V(f;I) = r$ [Eig07]. Thus, condition two is fulfilled and the detection of more than one multiple root is reported by the algorithm.

Remark. Either conditions can be validated in case f contains exactly one real multiple root and further imaginary ones. However, it is hard to predict whether the algorithm terminates with success or not. It simply depends on the distribution of the roots and how the algorithm explores sign variations on related (and subdivided) intervals.

It remains to mention how to compute m and k. For polynomials with integral or rational coefficients, these values can be computed directly with the Sturm-Habicht sequence using Algorithm 2.3 and the modified version of Algorithm 2.8. If the coefficients of f are arbitrary in $\mathbb{R}$, the situation is, in general, not feasible. Again, there is a special case that is important for us. Consider a situation as in Definition 2.17, with a polynomial $f \in \mathbb{K}[t_1, \ldots, t_d]$ and a vector of real algebraic numbers $\overline{\alpha}$ with dimension $d-1$. We aim for the number of distinct real roots of $f_{\overline{\alpha}} \in \mathbb{R}[t_d]$, which is not necessarily square-free. The trick to compute m is afresh the specialization property that is deployed in the next algorithm.

Algorithm 2.10. Computing number of distinct real roots of a specialized polynomial

INPUT: $f \in \mathbb{Z}[t_1, \ldots, t_d]$; $\overline{\alpha}^{(d-1)} = (\alpha_1, \ldots, \alpha_{d-1}) \in \mathbb{R}$, each α_i in (recursive) algebraic interval representation

OUTPUT: The number of distinct real roots m of $f_{\overline{\alpha}^{(d-1)}}$

1. Compute the sequence $S = s_0, \ldots, s_n$ with $s_i := \operatorname{sign}(\operatorname{stha}(f)_{\overline{\alpha}^{(d-1)}})$. Observe, that $\operatorname{stha}_i(f)_{\overline{\alpha}^{(d-1)}} \in \mathbb{R}$. The sign computation is performed by Algorithm 2.7.
2. Use S to proceed with Step 3 of Algorithm 2.3

Remark. In order to apply the specialization property, we assumed that $\deg(f_{\overline{\alpha}^{(d-1)}}) = \deg_{t_d}(f)$; otherwise a proper reductum of f must be considered. This modification relies on Lemma 2.19.

Lemma 2.19. *Let* $d_\alpha = \deg_{t_d}(f_{\overline{\alpha}^{(d-1)}})$. *Then, for all* $j = 0, \ldots, d_\alpha$, *it holds that*

$$\mathrm{StHa}_j(f_{\overline{\alpha}^{(d-1)}}) = \mathrm{StHa}_j(f_{(d_\alpha)})|_{(t_1,\ldots,t_{d-1})=\overline{\alpha}^{(d-1)}}$$

This obviously extends to stha.

Computing k is again free of cost using the known sequence S in the first step of the Sturm-Habicht-version of Algorithm 2.1.

While the overall description is quite abstract, we mention that the m-k-Descartes method in combination with Algorithm 2.10 to compute m (and somehow k) has successfully applied when analyzing algebraic curves; see [EKW07] and [EK08a]. These publications also discuss what to do when a multivariate f is, in contrast to our assumption, not primitive. We also discuss this subtlety, when using the same ansatz to lift planar points onto algebraic surfaces in order to analyze them; see Chapter 5.

Remark (Low degree polynomials). For the sake of completeness we finally want to mention that there exists exact solution formulas for univariate polynomials of degree at most 4 by Cardano, Tartaglia, and del Ferro. Furthermore, it is possible to compute the isolating intervals for such polynomials off-line and to model the comparison of such numbers as a finite decision tree; see [ET03b] and [ET03a]. For the main parts of this thesis, the polynomial often have degree larger than 4 and it is not analyzed in how far these methods still work in combination with bitstream coefficients, that we also deploy a lot. Thus, we decided to launch the general approaches. However, we encourage to cross check the approaches for such low-degree polynomials. Depending on the results,[7] the specialized methods can become the default for low degrees.

2.1.3. Implicit functions and delineability

When presenting next algebraic curves and surfaces, we want to make use of implicit functions. Thus, we quickly introduce them and present the implicit function theorem. Although the statement of the theorem is true in a more general setting, we restrict it to a case, whose abstraction is still sufficient to cover its application for curves and surfaces. The restriction matches also the conditions of delineability that we also introduce here.

Given a relation, our goal is to provide a tool that converts it into a function, that is, the relation should be represented as the graph of a function. We do not aim for a single function, but there may be one for a restriction of the relation's domain.

Theorem 2.20 (Implicit Function Theorem). *Let* $f : \mathbb{R}^{d-1} \times \mathbb{R} \to \mathbb{R}$ *be a continuously differentiable function and let* $(\hat{u}_1, \ldots, \hat{u}_{d-1}, \hat{v}) \in \mathbb{R}^{d-1} \times \mathbb{R}$ *with* $f(\hat{u}_1, \ldots, \hat{u}_{d-1}, \hat{v}) = 0$. *If* $\frac{\partial f}{\partial v}(\hat{u}_1, \ldots, \hat{u}_{d-1}, \hat{v}) \neq 0$, *then there exist open sets* U, *with* $(\hat{u}_1, \ldots, \hat{u}_{d-1}) \in U \subseteq \mathbb{R}^{d-1}$, *and* $V \subseteq \mathbb{R}$, *with* $\hat{v} \in V$, *and a unique continuously differentiable function* $G : U \times V$ *such that* $\{(u_1, \ldots, u_{d-1}, G(u_1, \ldots, u_{d-1}))\} = \{(u_1, \ldots, u_{d-1}, v) \mid f(u_1, \ldots, u_{d-1}, v) = 0\} \cap U \times V$, *that is, the graph of* G *is precisely the continuous set* $f|_{U \times V} = 0$.

A proof can be found in [Kön93, §3.6], while [KP02] discusses various aspects of the theorem in detail. It is advantageous, that no knowledge on the exact G is required. The theorem only states about its existence. Very often, G cannot be solved with exact

[7] We consider running time and the stability of the specialized methods for bitstream coefficients.

formulas. Let us next establish a connection between implicit functions, multivariate polynomials, and real algebraic numbers in algebraic interval representation: We introduce a term that is well-known in cylindrical algebraic decomposition; see §2.1.6 for a short introduction and [CJ98] for a detailed survey.

Definition 2.21 (Delineation). Let $f \in \mathbb{R}[t_1, \ldots, t_d]$, $\deg_{t_d}(f) = n$, $A \subset \mathbb{R}^{d-1}$. The roots of f are *delineable* on A, and functions $f_1, \ldots, f_m$ *delineate the real roots* of f on A if for $(\alpha_1, \ldots, \alpha_{d-1}) \in A$ we have

- $m \geq 0$ and there are integers $w_1, \ldots, w_m$, $w_i > 0$ such that $f(\alpha_1, \ldots, \alpha_{d-1}, t_d)$ has m distinct real roots, with multiplicities $w_1, \ldots, w_m$.
- $f_1 < f_2 < \ldots f_m$ are continuous functions from A to $\mathbb{R}$.
- $f_i(\alpha_1, \ldots, \alpha_{d-1})$ is a root of $f(\alpha_1, \ldots, \alpha_{d-1}, t_d)$ with multiplicity w_i.
- If $\beta \in \mathbb{R}$ with $f(\alpha_1, \ldots, \alpha_{d-1}, \beta) = 0$, then $\exists 1 \leq i \leq m$ with $\beta = f_i(\alpha_1, \ldots, \alpha_{d-1})$.
- $\sum_{i=1}^{m} w_i = n$, which implies $\mathrm{lcf}_{t_d}(f) \neq 0$.

Observe that m is independent on the choice of $(\alpha_1, \ldots, \alpha_{d-1}) \in A$. As a result, for a multivariate polynomial $f \in \mathbb{R}[t_1, \ldots, t_d]$, the not yet specified implicit function over a condition fulfilling set U can be identified by f_i if U is a delineable subset $\mathbb{R}^{d-1}$. Even more, as the image of an implicit function is connected, it suffices to compute one of its values, that is, we need to describe β for $\overline{\alpha}^{(d-1)} = (\alpha_1, \ldots, \alpha_{d_1}) \in A$. If all elements of $\alpha^{(d-1)}$ are given in algebraic interval representation, to compute the algebraic interval representation of β requires to isolate the real roots of $f_{\overline{\alpha}^{(d-1)}}(t_d)$, for example using the bitstream Descartes method. This technique has already been used to analyze algebraic curves, and we come back to this point when analyzing algebraic surfaces in Chapter 5.

2.1.4. Algebraic plane curves

When increasing the dimension to 2, the vanishing set of a polynomial does not define a set of algebraic numbers, but it defines a curve. In this section we introduce algebraic plane curves, explore their properties, and show which other objects it can define.

Definition 2.22 (Algebraic plane curve). Let $\mathbb{K}$ be a field, and $f \in \mathbb{K}[x, y]$. The *algebraic plane curve* induced by f is the point set $V_{\mathbb{K}}(f)$. If $\mathbb{K} = \mathbb{R}$, it is named *real*, while for $\mathbb{K} = \mathbb{C}$ the set defines a *complex* curve.

First, we remark, that for the reason of intuition, we prefer the more descriptive variable names x and y over the abstract ones t_1 and t_2. Second, abusing notation, we often refer to *curve* f while actually meaning the point set $V_{\mathbb{K}}(f)$ induced by f. If some $p = (p_x, p_y)$ fulfills $f(p_x, p_y) = 0$, that is $p \in V_{\mathbb{K}}(f)$, we shortly say that p *lies on* f. If f factorizes, (see §2.1.1), each factor constitutes a *component* of a curve. For components of a curve Proposition 2.5 can be applied. An implication is, that the square-free part $f^\star$ of f defines the same curve as f. In real applications, it is usual to compute a (square-free) factorization of f first, and then to handle each factor as its own curve.

An algebraic curve has a vertical line at α if $f(\alpha, y) \equiv 0$. The existence of a vertical line in a curve complicates its analysis, while curves consisting of vertical lines only are almost trivial to analyze. Fortunately, it can be shown that factorizing $f = \mathrm{cont}(f) \cdot \mathrm{pp}(f)$, decomposes the curve into two components: The curve defined by $\mathrm{cont}(f)$ contains all vertical line components, while $\mathrm{pp}(f)$ is free of them. Applying the presented multivariate

square-free factorization without the subsequent multiplication (see §2.1.1), we can kill two birds with one stone: We obtain the square-free factors of f, and each factor defines a curve that consists either of vertical lines only, or it is free of such. In the following, when presenting more details on algebraic curves, we exclude the simple vertical case, and assume that a curve is primitive, that is, $\mathrm{cont}(f)$ is constant, and square-free. We identify f_x with $\frac{\partial f}{\partial x}(p)$ and f_y with $\frac{\partial f}{\partial y}(p)$.

Definition 2.23 (Points on curves). Let p be a point on some curve f and consider the *gradient* given by the vector $(f_x(p), f_y(p))^T$. We call p *singular* if the gradient is zero. Otherwise, p is *regular* and we define the *tangent* at p as the line through p and perpendicular to the gradient. The point p is *critical* if $f_y(p) = 0$. If p is a critical regular point it is *x-extreme*, if the minimal index n with $f_y^{(n)}(p) = 0$ is even. We call p an *event point* if it is singular or x-extreme.

Remark. We finally aim to decompose algebraic curves into x-monotone subcurves with special properties. This explains whty we call points with $f_y(p) = 0$ critical. In case splitting into y-monotone subcurves is desired, one would call points with $f_x(p) = 0$ critical.

It can be shown with Bézout's theorem that the number of points on f with $f_x(p) = 0$ is finite, and so for $f_y(p) = 0$. This implies that the number of singular points, the number of critical points and the number of extreme points is finite. The x-coordinates of critical points are defined by the roots of $\mathrm{Res}_y(f, f_y)$. These roots α_i, $0 \leq i < k$, decompose the x-axis into $k + 1$ (possible unbounded) open intervals I_i, $0 \leq i \leq k$.

Consider a non-critical point $p = (p_x, p_y)$ on f. By Theorem 2.20 for $d = 2$ we have that the curve defined by f is given locally around p by a function $y = g(x)$, that is, for a point $p = (p_x, p_y)$, we have $p_y = g(p_x)$. This holds in particular for all points p with $p_x \in I_i$.

Splitting a curve f at critical points decomposes the curve into connected and open sets of points. The points of each such sets meets the criteria of the implicit function theorem. An implication is, that each such set is x-monotone and we call its closure an *arc* of f. Arcs defined such are maximal sets respecting critical points. It can be shown that the number of maximal arcs is finite.

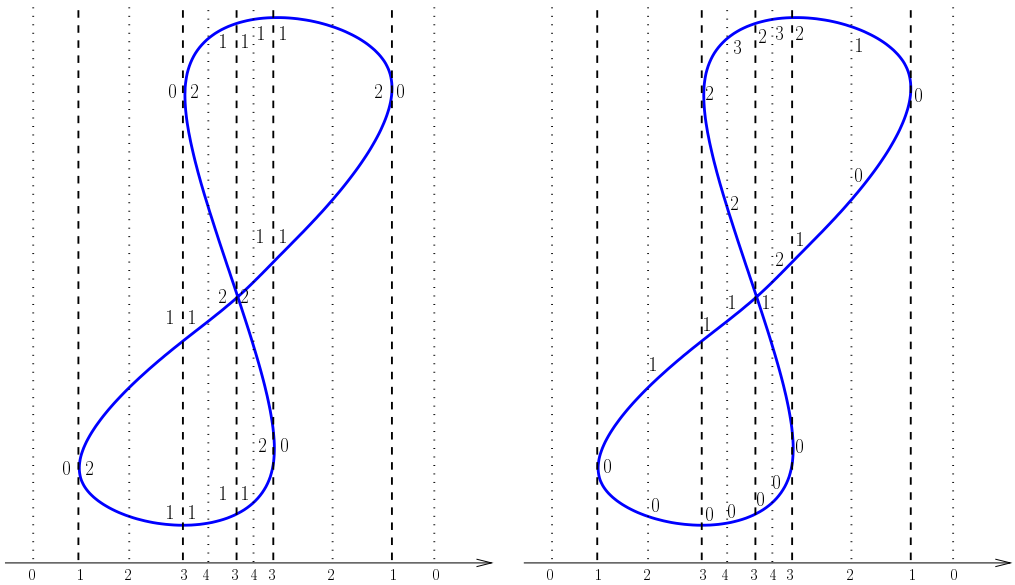
We actually distinguish three kinds of arcs: A *segment* has two finite endpoints, a *ray* has one finite endpoint and one unbounded end, and a *branch* has two unbounded ends. For an unbounded end we can distinguish whether it either approaches a horizontal asymptote, a vertical asymptote, or a tilted asymptote. The axis-aligned asymptotes of f can be computed. We state the corresponding theorem without proof.

Theorem 2.24 (Vertical and horizontal asymptotes). *Consider $f \in \mathbb{R}[x, y]$ as a univariate polynomial in y, with $\mathrm{lcf}_y \in \mathbb{R}[x]$ being its leading coefficient. If $x - \alpha$, $\alpha \in \mathbb{R}$ is a vertical asymptote of f, it holds $\mathrm{lcf}_y(\alpha) = 0$.*

Consider $f \in \mathbb{R}[x, y]$ as a univariate polynomial in x, with $\mathrm{lcf}_x \in \mathbb{R}[y]$ being its leading coefficient. If $y - \beta$, $\beta \in \mathbb{R}$ is a horizontal asymptote of f, it holds $\mathrm{lcf}_x(\beta) = 0$.

At finite ends, arcs are connected via critical points. An arc is incident to such a p either from left or from right. The *incidence numbers* of p can be encoded as pair (ℓ, r), where ℓ is the number of arcs incident from left and r the corresponding number for the right side;

Figure 2.1. Important numbers for a single curve. The values next to the x-axis encode the number of intersections of the curve with the vertical lines. Dashed for critical events, dotted for intervals induced by the events' x-coordinates.



(a) Incidence numbers **(b)** Arc numbers

see also Figure 2.1 (a). The incidence numbers of a non-critical point are equal to $(1, 1)$. For all incidence numbers (ℓ, r), it holds that $\ell + r \mod 2 = 0$, and $\sum_{p \text{ critical}}(\ell_p + r_p)$ is finite. In §2.3.3 we present an interface to provide information on incidence numbers for any point on a curve, especially the critical ones.

Definition 2.25 (Arc number). Let f be an algebraic curve, and $\alpha \in \mathbb{R}$ be an x-coordinate. We define $f_\alpha := f(\alpha, y) \in \mathbb{R}[y]$. Let $\beta_0 < \ldots < \beta_{r-1}$ be the r distinct real roots of f_α.

We say that a point $p = (p_x, p_y) \in \mathbb{R}^2$ is *supported by* α if $p_x = \alpha$ and $p_y = \beta_j$ for some $0 \leq j < r$. The value j is the *arc number* of p.

Figure 2.1 (b) gives an illustration. Observe, that β_i meets the conditions for algebraic interval representations of dimension 2. The isolating intervals can be computed with the bitstream Descartes method, if f_α is square-free, which holds for all $\alpha \in I_i$, for each valid i. If α is a root of $\text{Res}_y(f, f_y)$, f_α is not square-free. The m-k-variant of the bitstream Descartes method terminates: If successful, f_α has at most one multiple root. Otherwise, it detects the existence of more than one multiple root. In the latter case, we can either compute the square-free part of f_α using Algorithm 2.4 or apply a shear; see Definition 2.26.

By the implicit function theorem and the conditions on the incidence numbers, it can be shown that the number of points supported by all $\alpha \in I_i$ for some i, is constant. As an implication it is easy to see, that all interior points of an arc carry the same arc number. That is, to describe a (not necessarily maximal) arc it suffices to give an x-range[8] $X = [x_{\min}, x_{\max}]$ and three arc numbers, namely one for $x_{\min}$, one for $x_{\max}$, and the one that gives the constant arc number for the arc's interior points.

[8] We also allow for X the intervals $]-\infty, x_{\max}]$, $[x_{\min}, +\infty[$, and $]-\infty, +\infty[$

In the setting we described so far, it can be, that critical points share common coordinates. A technique to overcome this problem is a *change of coordinates* by applying a shear, for example, of the y-axis. This is possible as the number of distinct bitangents of a curve is finite.

Definition 2.26 (Shear). The *shearing* of a point $p = (p_x, p_y)$ with *factor* s is $\mathcal{S}_s(p_x, p_y) = \begin{pmatrix} 1 & s \\ 0 & 1 \end{pmatrix}(p_x, p_y) = (p_x + sp_y, p_y)$. It can also be applied to a point set P: $\mathcal{S}_s(P) = \{\mathcal{S}_s(p) \mid p \in P\}$.

The *shearing of a curve* uses the inverse $\mathcal{S}_{-s}$, that is, $\mathcal{S}_s f(x, y) = (f \circ \mathcal{S}_{-s})(x, y) = f(x - sy, y)$ and it follows $f(p) = 0 \Leftrightarrow \mathcal{S}_s f(\mathcal{S}_s(p)) = 0$.

Shearing is often applied to simplify the analysis of a curve. A shearing preserves the topological properties of the curve. However, it changes the geometry, except for y-coordinates. Aiming for a geometrical-topological correct analysis, a *back-shear* has to be applied, which is non-trivial due to algebraic numbers of high degree. We skip details, for example discussed in [EKW07], and henceforth assume, w.l.o.g. the possibility to compute incidence and arc numbers of a curve without shearing.

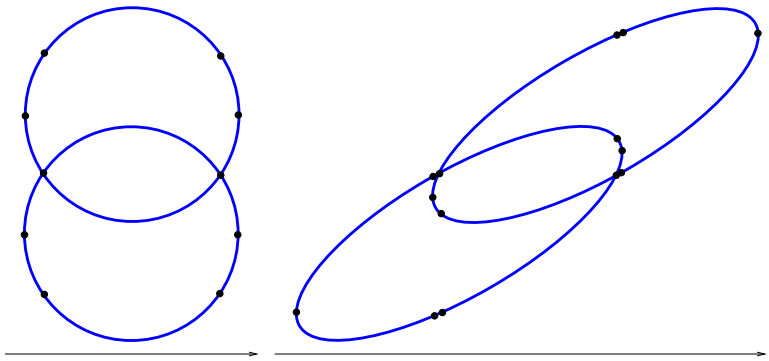


Figure 2.2. Shearing of a curve: (Left) The input. (Right) Its sheared version with $s = \frac{8}{7}$. Observe, that covertical degeneracies vanish on the right, however, the number of split points increased.

We next turn towards a pair of (coprime) curves f and g. A solution to detect their (candidate) intersections is to merge the sequences of roots of $\mathrm{Res}_y(f, f_y)$, $\mathrm{Res}_y(g, g_y)$ and $\mathrm{Res}_y(f, g)$. For the resulting intervals J between such numbers, refinements of the roots of f_r and g_r give the intersection scheme of f and g along the line $x = r$ with $r \in J \cap \mathbb{Q}$. No intersection of f and g takes place at such an r. It is more complicated if we consider an α that is a root of $\mathrm{Res}_y(f, g)$, as for it we can have an intersection of f and g along the line $x = \alpha$. A straightforward approach is to compute the square-free part of $gcd(f_\alpha, g_\alpha)$ using the stha-versions of Algorithm 2.1 and Algorithm 2.4. A more sophisticated solution using filter techniques (see §2.3.2) is presented in [EK08a] and [Ker].

For more detailed introductions to algebraic curves we recommend to read [Wal50] and [Gib98], while [Ker06] focusses on the goal to support their analyses via an algebraic kernel; see also §2.3.3 (page 56 ff).

2.1.5. Algebraic surfaces

We next introduce algebraic surfaces. They form the central input for our algorithms in Chapters 3, 4, and 5.

Definition 2.27 (Algebraic surface). Let $\mathbb{K}$ be a field, and $f \in \mathbb{K}[x,y,z]$. The *algebraic surface* induced by f is the point set $V_{\mathbb{K}}(f)$. If $\mathbb{K} = \mathbb{R}$, it is named *real*, while for $\mathbb{K} = \mathbb{C}$ the sets defines a *complex* surface.

Once again we prefer variable names that match to the coordinate axes, here of the real affine space, as we do not discuss complex surfaces in this thesis. Surfaces are very similar to curves, as both are supported by multivariate polynomials. Hence, we also abuse notation and talk about the surface f instead of the surface induced by f. A point $p = (p_x, p_y, p_z) \in \mathbb{R}^3$ *lies on the surface* if $f(p_x, p_y, p_z) = 0$. The factors of f define *components* of f. The square-free part $f^\star$ of f defines the same surface, and usually, it is recommended to compute a (square-free) factorization of f to handle each component of f as a surface of its own.

A surface contains a *z-vertical line* at $p = (p_x, p_y)$ if $f(p_x, p_y, z) \equiv 0$. We call a surface *z-vertical* if for each point $p = (p_x, p_y, p_z) \in V_{\mathbb{K}}(f)$ it holds $f(p_x, p_y, z) \equiv 0$. If the context talks about a surface we write *vertical* instead of z-vertical. As for curves, decomposing a surface $f = \mathrm{cont}(f) \cdot \mathrm{pp}(f)$ partitions f into two surfaces, one that is vertical, namely $\mathrm{cont}(f)$ and one that is not vertical, namely $\mathrm{pp}(f)$. The multivariate square-free factorization (without post-processive multiplying) splits curves into square-free vertical and non-vertical components. As vertical surfaces are easy to handle, we do not trace them further, and assume that a surface is square-free and primitive. Nonetheless, a primitive surface can still contain (isolated) vertical lines. We refer to Chapter 5 where we discuss this problem in depth.

The *gradient* vector of a point p on f is given by $(f_x(p), f_y(p), f_z(p))^T$, where $f_x = \frac{\partial f}{\partial x}$, $f_y = \frac{\partial f}{\partial y}$, and $f_z = \frac{\partial f}{\partial z}$, which allows to classify points on f.

Definition 2.28 (Points on surfaces). Let p be a point on an algebraic surface f and consider its gradient vector. We call p *singular* if the gradient is zero. Otherwise, p is *regular* and we define the *tangent* at p as the plane through p and perpendicular to the gradient. The point p is *critical* if $f_z(p) = 0$.

Consider a non-critical point $p = (p_x, p_y, p_z)$ on f, then Theorem 2.20 for $d = 3$ means that the surface defined by f is given locally around p by a function $y = g(x,y)$, that is, for a point $p = (p_x, p_y, p_z)$, we have $p_z = g(p_x, p_y)$. In Chapter 5 we introduce a decomposition of $\mathbb{R}^2$ into delineable sets for which the implicit functions as described exists. For each set we provide additional combinatorial data that helps to analyze the algebraic surface. We also say that the surface is *xy-functional* over such a connected two-dimensional cell. The closure of the function graph is called a *sheet*.

In contrast to curves, there is no left and right (zero-dimensional) end that together describe the closure of such a sheet. Actually, it is a one-dimensional set of points. Such a space curve models the connection of sheets. Definition 2.33 formally introduces space curves. There are also sheets that lop off towards $z = -\infty$ or $z = +\infty$ when approaching their "boundary". To compute how sheets are connected (or extend to infinity) is another major goal that we are aiming for in Chapter 5. Besides the actual computation, we

also discuss an interface to access this information. We next extend the definition of arc numbers to surfaces.

Definition 2.29 (Sheet number). Let f be an algebraic surface, and $\alpha, \beta \in \mathbb{R}$ be an x- and y-coordinate. We define $f_{\alpha,\beta} := f(\alpha, \beta, z) \in \mathbb{R}[z]$. Let $\gamma_0 < \ldots < \gamma_{r-1}$ be the r distinct real roots of $f_{\alpha,\beta}$.

We say that a point $p = (p_x, p_y, p_z) \in \mathbb{R}^3$ is *supported by α and β* if $p_x = \alpha$, $p_y = \beta$ and $p_z = \gamma_j$ for some $0 \leq j < r$. The value j is the *sheet number* of p.

As for curves, it is possible to apply a shear on a surface in order to remove degenerate situations that are with respect to the choice of the coordinate system. That is, the topological properties of the surface are preserved, while its geometry changes (with constant z-coordinates). This helps to analyze the topology of the surfaces. However, shearing also has drawbacks. Applying a shear increases the bit-lengths of f's coefficients and the resulting polynomial is dense with respect to z. Both negatively influences the running times of subsequent algorithms. In addition, obtaining geometric information with respect to the original system is often expected, but regaining it is a highly non-trivial task. These items are reason enough for us to abstain from shearing when analyzing algebraic surfaces in Chapter 5.

In Chapter 4 we consider special examples of surfaces, namely such that are (rationally) parameterizable.

Definition 2.30 (Parameterizable surface). A *parametric surface* S in $\mathbb{R}^3$ is given by a parametric equation in two variables, that is, the surface is the image of $\varphi : \Phi = U \times V \to \mathbb{R}^3, (u, v) \mapsto (X(u, v), Y(u, v), Z(u, v))$, where X, Y, Z are functions $U \times V \to \mathbb{R}$.

Example 2.31 (Parameterizable surfaces).

- The graph of a bivariate function is parameterized with $\varphi(u, v) = (u, v, f(u, v))$.
- A cylinder of radius r around the x-axis is given by $\varphi(u, v) = (u, r\cos(v), r\sin(v))$, with $x \in \mathbb{R}$, and $v \in [0, 2\pi]$.
- The unit sphere's parameterization is $\varphi(u, v) = (\sin(u)\cos(v), \sin(u)\sin(v), \cos(u))$, with $u \in [0, \pi]$ and $v \in [-\pi, \pi]$.

It is easy to see, that the same surface admits several parameterizations. Furthermore, if Φ is bijective except for an at most one-dimensional set, there is another nice property. This property is mandatory for rational surfaces.

Definition 2.32 (Rational surface). A surface S is said to be *rational* if

1. S is algebraic (i. e., defined by a polynomial $f \in \mathbb{Z}[x, y, z]$).
2. There exists a parameterization $\varphi(u, v) = (X(u, v), Y(u, v), Z(u, v))$ of S by functions X, Y, Z which are quotients of polynomials in u and v having rational coefficients.
3. $(u, v) \in U \times V$, where U and V are itself defined in a simple way by polynomial inequalities in u and v and that, except for a few equally simple curves and points, φ is bijective.

We present in §4.6.2 ring Dupin cyclides that are rational surfaces which generalize tori.

The union of two vanishing sets of trivariate polynomials f_1, f_2 can be modelled by multiplying $f_1 \cdot f_2$. The intersection of two sets defines a new geometric object.

Definition 2.33 (Algebraic space curve). Let $f, g \in \mathbb{K}[x, y, z]$, and coprime. The *algebraic space curve* induced by f and g is the point set $V_{\mathbb{K}}(f) \cap V_{\mathbb{K}}(g) = \{(x, y, z) \in \mathbb{K}^3 \mid (x, y, z) \in V_{\mathbb{K}}(f) \wedge (x, y, z) \in V_{\mathbb{K}}(g)\}$ If $\mathbb{K} = \mathbb{R}$, it is named *real*, while for $\mathbb{K} = \mathbb{C}$ the sets defines a *complex* space curve.

An algebraic space curve induced by f and g is also referred to as the *intersection curve of f and g*. A special space curve of our interest is the *silhouette curve* of a surface f defined by the intersections of f and f_z. The silhouette curve contains all critical points of f. From elimination theory introduced in §2.1.1 and especially Proposition 2.8 (assuming z-regularity), we remember that the vanishing set of $\mathrm{Res}_z(f, f_z)$ constitute extendible solutions for the silhouette curve and $\mathrm{Res}_z(f, g)$ constitute extendible solutions for the intersection curve of f and g. It might be the case, that for a fixed solution there is no such extension, a single one, or even more than one extension, which is also due to the fact that algebraic curves are Zariski-closed.[9]

2.1.6. Cylindrical algebraic decomposition (cad)

In this section, we shortly review *cylindrical algebraic decompositions* (*cad*) introduced by Collins in his seminal work [Col75], which basically provides a general framework for applied elimination theory. We do so, as basic steps in our work, especially in Chapter 5, adopt ideas that have already been supporting cylindrical algebraic decompositions. In contrast to cad, that facilitates quantifier elimination and thus endorses various potential applications in any dimension, we focus in this thesis on tools supporting low dimensional geometric problems.

The input for a cad consists of a finite number of d-dimensional integral polynomials, while the output is a subdivision of $\mathbb{R}^d$ into cells, where each input polynomial is sign-invariant within each computed cell — a cad. The algorithmic idea to compute it is a repeated two-step approach: The first step, the *projection*, eliminates one variable, while the second step, the *lifting*, constructs so called *stacks* based on information obtained in the first step. Actually, the algorithm is recursive. The projection is stopped when univariate polynomials remain, which decompose $\mathbb{R}$ into cells that are sign-invariant with respect to the polynomials. Lifting is applied using sample points for each lower-dimensional sign-invariant cell until the decomposition of $\mathbb{R}^d$ is obtained. A lifting step constructs a *stack* that is partitioned into cells that are sign-invariant. Cells that result in zeros in the polynomials prior to the projection are called *sections*, while the open intervals between (and semi-infinite intervals preceding and following all zeros) are called *sectors*. Each cell of a d-dimensional cad has an index $(c_1, \ldots, c_d)$, $c_i > 0$. For example $(4, 2)$ is the second

[9] In Algebraic Geometry, there exists a naturally induced topology, called the *Zariski-topology*. It is defined by the assignement of a set to be *open* if and only if its complement is the vanishing set of an ideal. Thus, a plane algebraic curve C is always Zariski-closed as it is given as the vanishing set of a polynomial f, that is, $C = V_{\mathbb{K}}(f)$. A more intuitive geometric consequence is that for each point p on C there exists a neighborhood U such that $C|_U$ is either an isolated point or star-shaped whose center is p. Note that in case of a non-singular point p, the curve C, restricted to U, is homeomorphic to a line segment. In descriptive language we obtain the following: When "walking" on an algebraic curve (i. e., not isolated), one never reaches a point where the curve has a dead-end. As an example, we mention that a line segment does not constitute an algebraic curve; only the supporting line is one.

cell (from bottom) constructed over the fourth cell (from left) in a cad of $\mathbb{R}^2$. Projection and lifting heavily relies on delineability; see Definition 2.21 in §2.1.3.

A crucial step of the cad is the projection. In the original work, a huge bunch of polynomials are computed. In particular, as input all $f_i \in \mathbb{Z}[t_1, \ldots, t_{d-1}][t_d]$, all coefficients of all f_i, all principal subresultant coefficients of f_i and f_i', and all principal subresultant coefficients of f_i and f_j with $i \neq j$ are considered. It is assumed that the f_i are t_d-regular, otherwise, proper reductums must be used when constructing the principal subresultant coefficient. Computing all these polynomials needs a significant amount of time, while the large number also leads to a very fine decomposition of $\mathbb{R}^{d-1}$. This, as a sequence, results in the lifting of many cells, which again is time-consuming. The projection has been improved by McCallum [McC] and Brown [Bro01b]. They show how to obtain an *order-invariant* decomposition (compare the definition in [McC]). For such a decomposition, it suffices to only consider the leading coefficients and the discriminants of (possibly reduced) polynomials to ensure delineability. We come back to this point when analyzing algebraic surfaces in Chapter 5. Besides these computations of "projection polynomials" other symbolic subalgorithms are required, for example, to compute multivariate greatest common divisors, or, during lifting, to convert real numbers in algebraic interval representations into their integral interval representations. Notice that all operations are carried out with pure symbolic computation, that require exact and efficient integral arithmetic.

Projections and liftings apparently result in a cad, which constitutes a decomposition into connected sign-invariant cells. An additional *adjacency* step computes how cells are interacting. It is said that two cells are *adjacent* if their union is also connected. There exists approaches to compute the adjacencies for the two-dimensional case [ACM84] and for the three-dimensional case [ACM88]. Adjacencies also open the door to join adjacent cells with the identical sign-invariant to the same topological component. Arnon calls such maximal sets *clusters*. Computing clusters reduces the number of liftings, as for each cluster only one lift is demanded, and liftings are usually costly as they involved algebraic numbers [Arn88]. Thus, clusterings enable possible time savings, but they must be weighted against the time to compute the cluster. When analyzing algebraic surfaces in Chapter 5, we implicitly cluster the cells of the first projection (into the two-dimensional plane) using CGAL's planar arrangements; see §2.4.

Cylindrical algebraic decompositions have various applications; a comprehensive list is given in the introduction of [CJ98]. We exemplary mention the possibility to compute the topology of semi-algebraic sets, to solve systems of polynomial equalities and inequalities, and robot motion planning. The later considers given algebraic objects, some of them movable, others not. We want to know whether the movable objects can be continuously moved, collision-free, from an initial configuration to a final one. A configuration is given as a point in a high-dimensional algebraic space, as each parameter describes position and orientation of one object. A solution exists if the initial configuration can be connected with the final one by a continuous path within a connected cluster of the high-dimensional cad. For the decision it suffices to compute the cluster of the two configurations, while the actual movements can then be produced from a collision-free path within the cluster, for example, by constructing it with the help of cell-to-cell paths. More details can be found in [SS83], [SSH87], and [Lat93].

2.1.7. Topology and CW complex

We close the theoretical foundations with some information on topology. An *k-simplex* is a topological space that is equivalent to a k-ball $\mathcal{B}^k$, that is, every k-simplex constitutes a k-dimensional manifold with boundary. A k-cell is a space that is homeomorph to a k-simplex. An *open* k-cell is homeomorph to the interior of $\mathcal{B}^k$. We call k the *dimension* of the cell. If the boundary of a topological k-space is the finite union of k'-spaces with $k' < k$, we say that it has the *boundary property*.

A *complex* is a topological space constructed from simplexes which are wisely connected. A complex allows to describe a complicated space in terms of connected simple spaces. In general, topology decomposes objects into k-cells. We only mention two complexes. The *simplicial complex* K for a set M is a subset of the power set $K \subseteq \mathcal{P}(M)$, that is, a family of subsets that are closed under set intersections. Geometrically, a simplicial complex K is a complex of simplexes such that the empty set and all boundaries of simplexes are contained in K, and for $s_1, s_2 \in K$ it holds $s_1 \cap s_2$ is a boundary of s_1 and s_2. Whitehead [Whi49] introduced an even stronger complex.

Definition 2.34 (CW complex). A Hausdorff space[10] X that decomposes into open cells $(I)_{i \in I}$ is called *cell-complex*, or *closure-finite weak-topology complex* (*CW complex*), if

1. for each $c_i \in X$ there is a characteristic continuous function $f_i : \mathcal{B}^k \to X$, such that the interior of $\mathcal{B}^k$ is mapped homeomorphically to c_i and the boundary of $\mathcal{B}^k$ is mapped to a finite number of cells with dimension $< k$ (boundary property) and
2. $M \subseteq X$ is closed if and only if $M \cap f_i(\mathcal{B}^k)$ for all i is closed.

Decompositions of curves into arcs, see §2.1.4, and surfaces that we finally analyze in Chapter 5 are CW complexes. A more basic introduction is given in [Hat02].

2.2. Implementing geometric algorithms

The description of geometric algorithms usually assumes the *real RAM*, that is, each basic operation is to be considered as being exact and running in constant time [PS85]. These assumptions ease the theoretical considerations of an algorithm. But, not keeping its practical limitations for a concrete implementation in mind, they quickly lead to disastrous results: Code crashes, produces mathematically wrong results, or does not terminate. An example is an incremental convex hull construction that constructs non-convex hulls. This and other classroom examples are given more detailed in [Sch96], [KMP+04], or more recently in [Sch08].

In theory, life is also often simplified by the *general position assumptions*, that is, any degenerate input with respect to the algorithm is precluded. For example, no three points in the plane should lie on a common line. In contrast to theoretical expectations, degenerate input is not rare in practical applications, as, for example, scanners and sensors only have finite precision. Algebraic curves and surfaces also have degeneracies, for example, singularities and tangential intersections. If aiming for an accurate result in the original coordinate system, we have to deal with them.

In order to tackle these problems Kettner and Näher aimed for *geometric programming*, which asks for geometric software that is correct, efficient, adaptable and extensible, and

[10] Any two points can be "housed off" from each other by open sets.

easy to use [KN04]. To fulfill such a task, the incorporation of two well-known paradigms is beneficial, namely the generic programming paradigm and the exact geometric computation paradigm, which we discuss in the sequel.

2.2.1. Generic programming (GP)

In the definition of Musser et al. [MS88], *generic programming* consists of the gradual lifting of concrete algorithms that abstract over details, while preserving the algorithm's efficiency and semantics. Basic and well-known abstractions that a supported by various programming languages are subroutines, data type abstraction, and inheritance, as object-oriented code can provide.

However, generic programming is more powerful. In C++ it extensively makes use of class- and function-templates. Such a template expects one (or several) parameters of concrete classes (or functions) that exactly fulfill requirements positioned by the template. So-called *concepts* define the abstract definition and requirements for data types, while types (e. g., classes) that exactly fulfill such specifications are referred to as the *models* of a concept. Models are allowed to implement more than one concept at the same time, and such classes can also provide functionality beyond the expectations of a concept. Concepts can also be organized hierarchically. We refer to the *refinement of a concept* if a derived version has stronger expectations on a model. For example, a refinement expects an additional type or function in order to model the stronger. An interesting sub-case of models that describe behaviors of objects are called *traits* classes. The notation has been introduced by Myers [Mye95]. His design allows to attach information to classes that are not modifiable, such as poiters. In contrast, we usually refer to a different interpretation of the name *traits class*. It provides basic types and operations on them. Instantiating a templated data structure or algorithm with such a class determines the structure's or algorithm's actual behavior; see the sorting example below.

Concepts and models are nothing specific to (generic) programming. Actually, mathematicians are very familar with such, for example in algebra. Several concepts exists: Group, ring, field, vector space. A group is modelled by a set of (abstract) objects and a binary operation "+" that has to fulfill the known conditions. Examples of groups are $\mathbb{Z}$ with their addition as binary operation, or Z_p with $p = 2, 3, 5, 7, 11, \ldots$ with addition modulo p. A "traits class" is also reflected in algebra: Switching the "+" operation from addition to multiplication leads to a multiplicative group. We remark further subtleties as to restrict the elements. But observe that an implementation can use traits classes to define certain groups. Refinements of concepts also exist in mathematics. A group is called abelian, if the binary operation is also commutative. An abelian group forms a ring, if there is a second binary and associative operation "$\cdot$" and the distributive law holds. Fields and vector spaces are other refinements of abelian groups.

Generic code splits into two parts:

1. the instructions that describe the algorithmic steps and
2. requirements that specify which properties its argument types must satisfy.

Example 2.35. A simple example is a sorting routine that relies on a less-based comparison strategy on objects; like insertion sort. In the execution of the sorting algorithm it must be decided whether one given objects is *smaller* than a second one. The instructions of the algorithm are independent of the actual type of objects, while we demand objects to

be *LessThanComparable*. This way, an unexperienced user in sorting algorithms can still deploy the implementation, by just knowing about the order of two objects.

More abstract, it suffices to implement a model for the intended type of objects in order to benefit from generic implementations. This is usually much simpler than the full implementation of an algorithm for these objects. Implementations that follow the generic programming paradigm reuse code, and thus avoid copy-and-paste which is often a source of error. Additionally, it implies less maintenance.

Generic programming also looses the drawbacks of object-oriented programming, such as a strong inheritance relationship, with additional memory consumption for virtual members and virtual-function table lookups. In contrast, it benefits from flexibility, type checking at compile time, and no loss of efficiency. Full details of generic programming in its various aspects can be found in the book by Austern [Aus99]. It also surveys the *Standard Template Library* (STL) [15] whose various basic data structures and algorithms are part of the C++ standard library since 1994. The *Boost* libraries [2] implement additional software in the spirit of the STL and to work hand-in-hand with it. A very nice overview is given in [Kar06]. The *Library of Efficient Datastructures and Algorithms* (LEDA) [10] provides fundamental data structures and algorithms from various domains, and basic objects for geometric computations. Fully focussed on geometric problems are the *Computational Geometry Algorithms Library* (CGAL) [3], and the *Libraries for Exact and Efficient Algorithms for Curves and Surfaces* (EXACUS) [6]. We present both in detail in §2.2.3 and §2.2.4.

Generic programming for computational geometry makes perfect sense, as it allows to decouple geometric constructions and predicates from topological considerations and combinatorial algorithms and data structures. Templating algorithmic frameworks or data structures implements the desired abstraction. The instantiation of such a class with a concrete traits inspires the skeleton with respect to the given geometric objects and operations on them. This way, a user with limited knowledge about the class-template, that is, the geometric algorithm or data structure, can use it with his own geometric objects, as long as he can provide a proper traits class. The expected operations usually implement geometric or algebraic computations. We detail this issue when discussing the exact geometric computation paradigm next. Beforehand, we want to mention the important objective for a concept to be minimal. A tight concept simplifies the development of a new traits class drastically, as less (maybe only slightly) different operations must be implemented. If they are too similar, it might be hard to crystallize their differences, and it is also dangerous that the same (algebraic) value is computed several times.

The ability of generic programming to decouple combinatorics from geometric predicates is also a very nice way to resign from the generic position assumption. That is, the developer of a generic geometric structure can implement all particularities of an algorithm from the literature with respect to degeneracies assuming that geometric predicates implement the desired operation. He never has to care about the details how to provide the correct answer. This is another task. Again, this consideration is cross-linked with the exact geometric computation paradigm; see §2.2.2. Although this strategy is valid, it should be taken with a pinch of salt: Checking a degeneracy is often costly, especially in the EGC approach. However, it might be possible to modify an algorithm such that this check and its (positive or negative) outcome is combinatorially deduced from (a set of) less expensive predicates.

We want to mention that we illuminate as our major example of a templated geometric data structure the details of CGAL's `Arrangement_2` class in §2.4.3 Actually, to broaden the applicability of the class with respect to other domains than the *bounded plane*, we discuss an important change of its template parameters in Chapter 4.

2.2.2. Exact geometric computation (EGC)

The computational path of a geometric algorithm is influenced by two types of basic operations: *Constructions* that create new geometric objects and *predicates* that determine conditional steps in an algorithm. The splitting of these basic operations from the generic algorithm can be established in terms of generic programming. From this abstraction, we can conclude that different computational paths, that is, different evaluations of operations, lead to different combinatorial structures and statuses. Although sometimes tolerable, numerical errors (as they are typical for floating-point arithmetic) in such evaluations, can quickly lead to an invalid or inconsistent status of an algorithm. In order to avoid such problems, we have to ensure that predicate evaluations always compute the mathematical correct result. This goal is expressed by Yap [Yap04] as the *exact geometric computation (EGC)* paradigm. While we expected the real RAM to compute each operation in exact fashion, this paradigm relaxes the exactness requirements with respect to computed results. To explain this more precisely: In numerical stable settings an inexact, but fast, number type can already suffice to compute the correct result of a geometric predicate. This is usually the case in non-degenerate situations. However, this still requires techniques to verify the correctness of the result. In more degenerate cases such an approach might fail, and one has to fall back to an exact computation. In the spirit of the EGC paradigm several techniques have been implemented to ensure exact predicate evaluations, such as lazy-evaluation, adaptive computations, and floating points filters; see [She96], [BEPP97], [MN00, §9.7], [FM02] and §2.3.2.

The example of a fully filtered geometry kernel is given in [KN04]. It is also possible to filter geometric constructions. If so, one first creates a non-expensive (approximative) representation that serves non-critical needs, but which suffices to be converted to an exact representation if needed, for example, in degenerate situations. One option is to represent the intersection point of curves by a construction graph (i. e., its construction history) along with a rough approximation of its coordinates. The work of Hanniel and Wein on Bézier curves [HW07] implements such a technique.

Combining these two paradigms in geometric programming leads to convenient code that allows an easy switch to other number types, other computation techniques (maybe with filters), or instantiate generic code with exactly those objects a user is demanding for. All in all, it is usually just a minor change in the code. Often, it only requires to change a few type definitions.

For the sake of completeness, we mention that *controlled perturbation* is another technique to attack the mentioned, not very practical, assumptions. It has been introduced by Halperin and Shelton [HS98]. Its central idea is to perturb the input in a controlled fashion, such that degeneracies vanish and finite precision suffices to implement consistent and correct predicates for non-degenerate cases. The scheme usually adapts the perturbation and the required precision in several rounds until a correct result for a slightly wrong input is obtained. Controlled perturbation lead to fixed precision algorithms for numerous

geometric applications; see [HL04], [EH05], [HS98], [Raa99]. It also constitutes a simple and generic framework [FKMS05],[MOS06].

Despite of the success of controlled perturbation, this thesis does not pursue this approach any further, but focusses on the EGC paradigm. In this light, we want to mention two software libraries that excellently show with some of their main contributions the EGC's right to exist.

2.2.3. The Computational Geometry Algorithms Library (CGAL)

CGAL - the *Computational Geometry Algorithms Library* has been started in 1997 founded by academic sites in Europe and Israel. Its goal at that time (as today) is to promote research in computational geometry to reliable and efficient software that serves both academic and industrial users. Since a few years CGAL is available as an open-source licence. For users who want to hide their developed code using CGAL from the public (be they industrial or academic) GEOMETRYFACTORY [8] sells proper licences.

CGAL follows the generic programming (see [BKSV98]) and the exact geometric computation paradigms, which means that properly instantiated it always computes the correct result and never fails. For a detailed explanation of this topic we refer to [3, "The CGAL Philosophy"].

Central part of the library are *geometric kernels*. A geometric kernel contains constant-size non-modifiable basic geometric objects (e. g., in two-dimensional Cartesian coordinates) and a large set of basic operations on them. In addition to the kernels, CGAL partitions its code with respect to a wide range of geometric problems or data structures into *packages*. We exemplary mention convex hulls, triangulations, Voronoi diagrams, meshing and subdivisions, geometric optimizations, kinetic data structures, and the `Arrangement_2` package that we strip down in §2.4.3. The main classes and algorithms of each package are usually templated and expect traits classes that define the geometric objects considered and the required operations on them. Of course, CGAL provides traits classes for well-known and wide-spreaded objects, such as segments, lines, circles, triangles, meshes and more. Very often, one of CGAL's basic geometric kernels (2D, 3D, dD) already fulfills the requirement to serve as a model for a templated algorithm or data structure.

The *application programming interface* (API) of the library and each package is implemented in the spirit of the STL. This way, an easy and convenient connection of CGAL with other software through iterators and functors (as, e. g., the BOOST libraries) is ensured.

The basis of CGAL is constituted by non-geometric support facilities, such a generators, iterators, I/O-capabilities, visualization interfaces, and a tremendous support for number types and algebraic structures, like polynomials. The later entities have been redesigned in for CGAL's current public release 3.3 with respect to the experience that the EXACUS-project (see §2.2.4) gained in this area. It was a non-trivial task to exchange nearly the full support by a much more powerful implementation, while still keeping backward compatibility issues. The corresponding chapter [Hem07a] of CGAL's manual pages gives a detailed introduction to that important part of the library. Main basic number types[11] that we deal with in this thesis are taken from LEDA and CORE, that is, we rely on their exact implementation of *integers*, *rationals*, and *bigfloats*, as well as the interval type from BOOST.

[11] CGAL's `Number_type` package has also received non-trivial adaptions with the integration of EXACUS.

CGAL is a living project, thus it is continuously improved and new geometric problems are tackled every day by a large number of developers worldwide. Code quality is ensured by CGAL's *Editorial Board* that reviews new submissions of packages, and an exhaustive testsuite. This quality is known in the academic community and also for a growing number of industrial users, which states the success of CGAL. For all further details of the library we refer to its website [3] or its comprehensive manual [CGA07].

Ongoing work in CGAL that is touched and influenced by the aura of this thesis, but beyond our actual contributions, is the design and the implementation of algebraic kernels, mainly in one and two dimensions, whose details we present in §2.3.3.

2.2.4. Libraries for Exact Algorithms for Curves and Surfaces (EXACUS)

The EXACUS-project has been founded in 2001 at the Max-Planck-Institut für Informatik in Saarbrücken in order to implement ***Ef**ficient and **Ex**act **A**lgorithms for **Cu**rves and **S**urfaces* as a collection of C++-libraries. The focus of the project has always been to tackle problems in computing with curved objects that are algebraically defined following the exact geometric computation paradigm. These goals turned out to be also a demanding source for missing basic machinery, as, for example, integrating implementations for efficient and certified real root isolation.

While in the first years of its development it was advantageous to experiment with design rationales. With growing maturation, the separation from CGAL has become disadvantageous, as CGAL also started to dig into the non-linear world. Therefore, the EXACUS-developers decided in 2005 to merge their libraries as new packages into the more prestigious and popular CGAL. Thus, EXACUS is no longer on a release track, instead class-by-class moves. This relocation is an on-going task, as first CGAL should not break up, second demos in EXACUS are expected to work during their move to CGAL, and third EXACUS' development process should smoothly migrate towards CGAL, too.

We shortly repeat in the following EXACUS' main libraries with their content and their status with respect to the move. Thus, although more detailed, the article published in 2005 [BEH+05] turns out to be slightly outdated. The goal of our description of the libraries is to give an overview, while terminology that we use is either taken from the standard literature on this topic, or, if relevant for the thesis, given more detailed in §2.4 and §2.1.

SUPPORT This library provided basic support for non-geometry-related objectives. It used to contain memory allocation, I/O-methods, timers, basic enumerations and the `Handle_with_policy` class that implements a (possibly hierarchical) reference-counting scheme [Ket06]. Actually, main parts were loan from CGAL. These days, SUPPORT is not existing anymore. Classes that do not have an adequate alternative in CGAL or BOOST have been integrated in CGAL's basic packages, such as the mentioned `Handle_with_policy`.

NUMERIX The support for number types and algebraic structures developed in this library has been the successful prototype of CGAL's new and current basis for this business. Thus, this part has already moved completely to CGAL. Recently, the EXACUS' polynomials also have been integrated as `Polynomial` package of their own in CGAL [Hem07c]. Parallel to it, various representations for real algebraic numbers

and a couple of real root isolators exists; see [EK08b] for relevant parts in CGAL's `Algebraic_kernel_d` package. The library also contains other smaller classes, whose final role in CGAL is not determined yet. Some of them will move mostly unchanged, others can be expressed in terms of a more sophisticated design chosen for classes that already moved to CGAL. We omit details on this.

SWEEPX contains a generic sweep line algorithm [BO79] (see also §2.4.2 for a review of the algorithm) whose output is represented as LEDA-graph enhanced with geometric information. Actually, it is based on LEDA's implementation for line segment intersections [MN00, §10.7]. The library also provides a generic implementation to perform regularized boolean set operations on polygons whose boundaries are described by curved arcs. We stopped to develop this code, as it is published under a special restrictive licence and CGAL's `Arrangement_2` package offers much more flexible and extendable counterparts.

In contrast, we already extracted and improved an important module from SWEEPX as package in CGAL. A framework to represent points and arcs on curves that can be analyzed is now available as CGAL's `Curved_kernel_via_analysis_2` package. This package plays an important role for our work. We are using it instantiated with algebraic curves. Its details are presented in §2.4.4. The corresponding visualization [Eme07] also has already found its way into CGAL, and can even be used to render arrangements on a surface; see §4.6.2.

CONIX,CUBIX,ALCIX Each of these libraries implements the analysis of a single algebraic curve and the analysis of pairs of them. CONIX has been implemented first and supports curves of degree up to 2 (*conics*, [BEH+02]), while CUBIX can deal with curves of degree up to 3 (*cubics*, [EKSW06]). ALCIX is the newest library. Its analyses does not have any restriction on the degree of the supported planar curves;[12] see [EKW07], [EK08a]. §2.3.3 repeats its main achievements. Very recently, the development of ALCIX has stopped, and its ingredients have been interfaced as CGAL's new `Algebraic_curve_kernel_2`; we refer to §2.3.3 for more details. The other libraries might be integrated as filters (see §2.3.2) for low-degree curves. Further classes, such as combinatorial representation in CONIX, will be integrated elsewhere in CGAL.

QUADRIX The library currently still implements two approaches with respect to algebraic surfaces of degree 2, so-called quadrics. One approach uses a parameterization of the intersection curves [DHPS07], while the other approach projects them onto the xy-plane. For the latter, a specialized planar curve (and pairs of them) can be analyzed and lifted back [BHK+05]. In this thesis, we show how to box the approach using CGAL's new hierarchies.

For a short time, we added algebraic surfaces of arbitrary degree to this library. Obviously, this addition was only temporarily as the whole library is planned to be maintained as a new package in CGAL, that is, the surfaces already found their place in CGAL's new `Algebraic_kernel_d` package. Chapter 5 takes up the discussion of algebraic surfaces.

[12] Theoretically. In practice, the required running time constitutes limits on reasonable algebraic degrees of the curves.

Main contributors of EXACUS are the authors of [BEH+05] and Pavel Emeliyanenko, Michael Kerber, and Sebastian Limbach who joined more recently. Both libraries, EXACUS and CGAL, provide, besides other libraries, a large set of various classes and tools on which we rely in this thesis. We continue to present our kit.

2.3. The arithmetic and algebraic tool kit

2.3.1. Arithmetic and number types

Geometric algorithms are closely coupled with arithmetic. As we learned in §2.2, geometric algorithms assume the real RAM, which is not modeled by computers. In contrast, the hardware provides standardized fixed-size integers and IEEE 754 [IEE85] floating-point arithmetic. Both have the drawback of limited precision, that is, it is not possible to model arbitrary large or precise values. The hardware floating-point numbers (like `double`) model only a finite and discretized subset of $\mathbb{Q} \subset \mathbb{R}$, which implies rounding errors. Both facts go against the conditions for the real RAM. In fact, also no software type exactly fulfills these conditions.

Consider a bit-array of variable length, that models arbitrary-size integers. Here, the variable length is a contradiction to the constant-time operations assumed. Similar for rational numbers modeled as a pair of integers. Usually, rational numbers are considered as the fundamental arithmetic type for geometric applications, as it allows to input exact information, for example, the endpoints of a triangle. In terms of software, several libraries are available to model arbitrary-size integer and rationals. Examples are GMP [9] MPFI [11], MPFR [12], LEDA [10], and CORE [4]. A special subset of rational numbers, namely floating-point numbers whose precision can be determined at run-time, so-called *bigfloats*, are also provided by some of the libraries.

But rational numbers are not the end of the road, as for certain geometric operations, such as the computation of the intersection of objects, we quickly reach (real) algebraic numbers of higher degree. So the dilemma is, how to deal with them, when only rational arithmetic is effectively available. As presented in §2.1.2 various methods exist to model algebraic numbers. The exact approach using algebraic expressions is implemented by CORE's `Expr` number type and LEDA's `real` number type. On the other hand, CGAL provides a generic type to represent real algebraic numbers using a square-free polynomial and an isolating interval; see Definition 2.16. Being such generic allows to select both main types: For the interval boundaries usually a rational number is chosen, while intervals of bigfloats are also conductable. The type of the polynomial's coefficients is also selectable. Beyond integral coefficients, it is possible, for example, to represent roots of $f \in \mathbb{Q}(\sqrt{2})[t]$. A special subset of algebraic numbers (for example to represent such coefficients) are field extensions by square-roots, for example, $\mathbb{Q}(\sqrt{2})$. CGAL provides a number type `Sqrt_extension` that allows to represent *one-root numbers* α in the form $\alpha = a + b \cdot \sqrt{c}$, where usually $a, b, c \in \mathbb{Q}$. However, a nesting is also possible, that is, some cases require that a, b, c are already of type `Sqrt_extension`. This nesting poses no problem for this type. Example usages are: Rotating curves by algebraic angles [BCW07], or representing a parameterization of the intersection curve of two quadrics [LPP06]. We give another in Chapter 3.

All these libraries are freely available for open-source academic developing.

In order to combine related number types, CGAL defines an *ArithmeticKernel* concept. Two models are available: One for the number types of LEDA and one for the number types of CORE. Each class consists of type definitions for integers (`Integer`), rationals (`Rational`), exact floating-point numbers (`Exact_float_number`), and algebraic numbers using algebraic expressions (`Field_with_kth_root`). If not stated otherwise, we are using the CORE-version (`CORE_arithmetic_kernel`).

Interval arithmetic

Performing arithmetic on exact algebraic numbers is costly. However, it is often the case, that an approximative solution suffices to deduce the correct answer. Interval arithmetic is one technique to achieve this goal. Instead of an exact value, we store an interval that approximates the value from below and above, also called the *inclusion property*. Each arithmetic operation preserves this property, that is, the exact result of the operation is also contained in the resulting interval. Several variants of interval arithmetic exists. Some of them try to minimize an intrinsic drawback of the method, namely the overestimation after an arithmetic operation. In our setting, we rely on BOOST's [2] interval arithmetic capabilities. Its implementation allows to choose the number type of the interval boundaries, for which we typically choose rational numbers or LEDA's bigfloats. Note that CORE's `BigFloat` type already implements an interval. Interval arithmetic is usually chosen as a filter, for example, to detect whether an algebraic expression may be equal to 0.

2.3.2. Filters

In geometric predicates we are mainly interested in the sign of an algebraic expression. Though, exact or multi-precision arithmetic produces correct results, their usage is quite expensive compared to the unit-cost model of constant-precision floating-point arithmetic in hardware, which often computes an almost correct result. The error propagation is usually of small amount. A wrong sign happens to appear if the value of which the sign is sough is (close to) 0. Geometrically, we can identify degenerate or near-degenerate situation for such cases. In case the value is not (close) to 0, the computed sign is usually correct. The solution to this dilemma is a method that combines approximative methods with a correctness guarantee for the case it succeed. Before we dig into the details, let us introduce the concept of a filter generically.

Definition 2.36 (Filter). A *filter* is a technique to compute a decision with an approximative method that also provides a *certificate* saying that the computed decision is identical to the decision when computing it with an exact method.

If the certificate cannot guarantee the correctness of the decision computed by approximated methods, we call it a *filter failure*. In this ineffective case, another method must be used to compute the correct decision, for example, a filter with more precision, or the exact method.

For a concrete application it has to be checked, whether a finally correct result is required. In the EGC paradigm, that we follow, this is mandatory. When utilizing a filter it is expected that it often succeeds, and the costs of the remaining exact fall-backs (where no filter applies) will be amortized over many calls of a predicate. Finding the optimal filter is non-trivial, and depends on various factors. The structure of operations

in a predicate and how each affects the computational error are such factors. On the other side, the input data also influences the success of a certain filters, as it fails more often in (almost) degenerate situations. Typically, a cascade of filters is a good idea. It first tries the less precise and fastest one, and in case of failures it continues with more precise and more expensive ones. We next present some filter techniques used in geometric algorithms.

Arithmetic filters We already mentioned in the introduction to filters that inexact arithmetic computation often leads to correct result (e. g., in terms of a computed sign). Various techniques exploiting this fact exist. One of the easiest one is interval arithmetic that we already introduced in §2.3.1. Arithmetic expressions and polynomials can be evaluated using interval arithmetic. It should be remarked, that a naive way may lead to unnecessarily bad results, or in other words: There exist evaluations schemes that minimize the (expected) error.

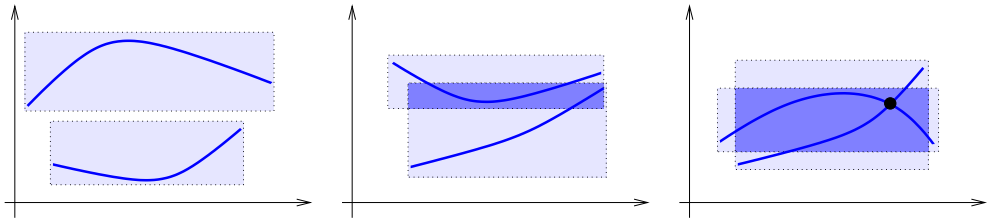
Interval arithmetic is a very efficient way to check, whether an arithmetic expression can evaluate to zero or not: If the resulting interval does not contain zero, the sign is determined. It depends on the application, of which type the interval boundaries are. It is very common to use hardware floating-point arithmetic for this purpose. However, we mainly use rational arithmetic, as the boundaries of isolating intervals of real algebraic numbers are usually represented as such. This enables to quickly check, whether a polynomial at some algebraic α can be 0. For example, Algorithm 2.6 can be enhanced with such a filter, that is, before computing the costly gcd. Anyhow, an even better filtering of a gcd-computation can be established with modular arithmetic; see below.

Interval arithmetic is a dynamic filter, that is, no prior analysis of the arithmetic expression is required. To its contrary, static filters apply an off-line analysis of possible errors, and design the filter with respect to this analysis [FV96]. As we are not using static filters, we skip their discussion.

Modular arithmetic Modern computer algebra systems heavily rely on modular arithmetic, which also holds for the algebraic computations that we are executing. Together with the Chinese remainder theorem it speeds up several algebraic algorithms, like the gcd or the resultant computation; see, for example, [vzGG99]. In addition, it can be used as a very efficient filter. The reason it that it is often possible to exclude that some value is zero by computing its modular correspondent with respect to one prime only. The modular correspondent requires only a fixed number of bits, which is the crucial fact for the efficiency of the filter. We want to mention that the vast majority of algebraic computations conducted in this thesis are filtered with modular arithmetic in the actual implementation. For details see [Hem07b] and [HH07].

Geometric filtering Filters are not restricted to arithmetic expressions. An approximate version of a geometric object also allows to derive a correct decision in some cases. A well-known technique is to filter a routine that computes the intersections of two geometric shapes. For that purpose each object can be enhanced with a *bounding box*. The rationale of the intended filter is, that two such objects only intersect, if their bounding boxes intersect. Figure 2.3 lists the three possible cases. Such boxes can be represented with rational or even fast floating-point arithmetic. If they are axis-aligned, their intersection test reduces to a few comparisons.

Figure 2.3. Geometric filtering by bounding boxes



(a) Bounding boxes do not overlap, thus no intersection

(b) Bounding boxes overlap, but no intersection contained

(c) Bounding boxes overlap, an intersection is contained

We exemplary mention the intersection tests of two arcs on coprime conics. If their bounding boxes do not intersect, the filter avoids to try to compute intersection points whose coordinates are algebraic of degree up to 4. Other examples for applying bounding box filters are given in [PTT06] and [Ker08].

Combinatorial deduction We have already seen an example of combinatorial deduction, namely the m-k-Descartes method, where additional information on a non-square-free polynomial allows to lead the Descartes method to a termination.

The rationale of a combinatorial deduction is to use available combinatorial information to simplify the problem, or to exclude a non-trivial set of solutions, similar to a branch-and-cut strategy in combinatorial optimization. In what follows, we often use the degree of a polynomial as a bound on the number of possible solutions. An example is a specialized implementation to analyze algebraic surfaces of degree 2 as it is conducted in Exacus' ConiX library. We present another application in in §5.4.2.

2.3.3. Algebraic kernels

Most geometric predicates required in algorithms of computational geometry are expressed in terms of algebraic computations. In order to be prepared for such computations Cgal follows the generic programming paradigm to specify *algebraic kernel* concepts.

Concepts

In Cgal, there is no single algebraic kernel concept. In contrast, the project has introduced a hierarchy of concepts that defines what computations are expected from different kinds of algebraic kernels. The concepts have been designed by the author in collaboration with Michael Hemmer, Menelaos Karavelas, and Monique Teillaud in the lifetime of the Acs-project [1] and improved in a series of technical reports [BHK+06a], [BHKT07]. The final review by Ron Wein [Fab07] lead to the current version [BHKT08] that we sketch next. The hierarchy consists of three layers. Each layer expects basic algebraic types and operations on them.

AlgebraicKernel_d_1

Types:

- `Polynomial_1` for univariate polynomials
- `Coefficient` its coefficient type
- `Algebraic_real_1` for real algebraic numbers (real roots of univariate polynomials)
- `Boundary` is the type for the boundaries of isolating intervals

Operations:

- On polynomials, the following self-explaining basic operations are expected: `Is_square_free_1`, `Make_square_free_1`, `Square_free_factorize_1`, and `Is_coprime_1`, `Make_coprime_1`.
- `Solve_1` is expected to implement a real root isolation, while `Sign_at_1` computes the sign of a polynomial at a given algebraic real.
- With `Lower_boundary_1` and `Upper_boundary_1` it is possible to approximate a single algebraic real, while `Refine_1` takes care to improve the approximation.
- Two real algebraic numbers can be compared with `Compare_1`, and if they are not equal) `Boundary_between_1` returns an intermediate value between them

AlgebraicKernel_d_2

This concept refines the univariate concept, by adding bivariate types and operations.

Types:

- `Polynomial_2` for bivariate polynomials (using `Coefficient`)
- `Algebraic_real_2` for zero-dimensional solutions of equational systems defined by bivariate polynomials

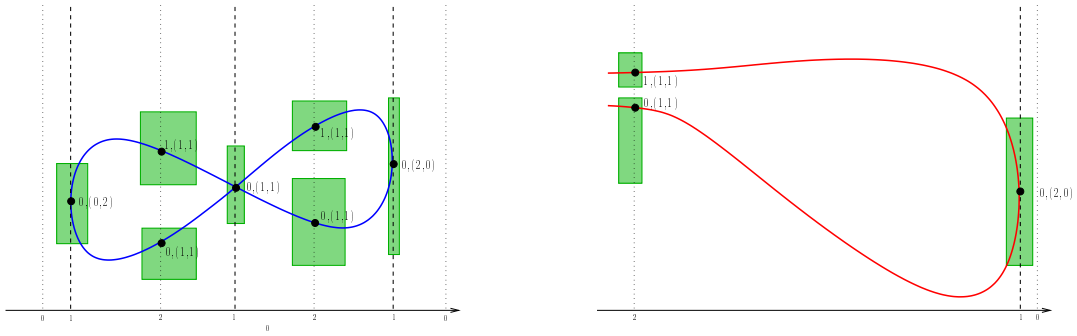
Operations:

- The polynomial operations naturally extend to the bivariate case: `Is_square_free_2`, `Make_square_free_2`, `Square_free_factorize_2`, and `Is_coprime_2`, `Make_coprime_2`.
- Central operations of the concept are to compute the zero-dimensional solutions of bivariate systems with `Solve_2` and to determine the sign of a bivariate polynomial at a given `Algebraic_real_2` with `Sign_at_2`.
- For a single solution, access to its individual coordinates is granted by `Get_x/y_2` that returns instances of type `Algebraic_real_1`. The two coordinates can be approximated independently as "interval" with `Lower_boundary_x/y_2` and `Upper_boundary_x/y_2`; a coordinate-specific approximation can be improved with `Refine_x/y_2`.
- For possible performance tuning, specialized (lexicographic) comparisons on two solutions are expected: `Compare_x_2`, `Compare_xy_2`, `Compare_y_2`. If a coordinate is not equal, it is possible to compute a value between two with `Boundary_between_x/y_2`.

AlgebraicKernelWithAnalysis_d_2

This most refined concept expects two additional types that interpret bivariate polynomials as real algebraic curves in the plane; see Definition 2.22.

- `Curve_analysis_2` analyzes a curve in the spirit of a two-dimensional cylindrical algebraic decomposition, that is, a y-per-x-view is established. To be more



(a) Bounded curve **(b)** Unbounded curve

Figure 2.4. The analysis of a single curve provides information on the curve at each x-coordinate, in particular the critical ones, and for representative boundaries in the open intervals induced by them. For each queried x-coordinate a *status-line* is constructed that stores how often the curve intersects the line, the arc number and incidence numbers for each intersection and a geometric approximation (green box). The analysis also provides access for information on possible vertical asymptotes of the curve; this case is not exampled in the figure.

precise, for each x-coordinate $x_0 \in \mathbb{R}$ it is possible to access a `Status_line_ca_1` that provides information about the curve's geometry and topology at x_0: The number of distinct intersections of the curve with the line $x = x_0$, their coordinates, and how branches of the curve to the left and right are connected with these intersections (also known as incidence numbers); see Figure 2.4 for examples of analysis of single curves and §2.1.4 for basic terminology on (algebraic) curves.

- `Curve_pair_analysis_2` provides, in the same spirit, y-per-x-information for pairs of coprime curves at each x-coordinate. For a given $x_0 \in \mathbb{R}$ an instance of `Status_line_cpa_1` describes the pattern how the two curves intersect the line $x = x_0$. Figure 2.5 gives an example of an analysis for a pair of curves.

The crucial fact is, that a given curve or pair of curves only has a finite number of different local topologies. That is, if only the topological information is desired, it suffices for a curve (or a pair of curves) to compute the status line instances at all x-coordinates of the event points and at a (rational) representative x-coordinate for each of the intervals that are induced by the events' x-coordinates. Implementations are recommended to take care of it and to benefit from this issue. Solely, the geometric information at a specially queried x-coordinate requires localized computations, that is, to compute another status line at a non-representative point in an induced open interval. In general, it is advised to compute status lines only on-demand, and to cache them after they have been computed for the first time.

Models

Concepts for algebraic kernels should also be modelled. Some already have been published, each implementing a different strategy.

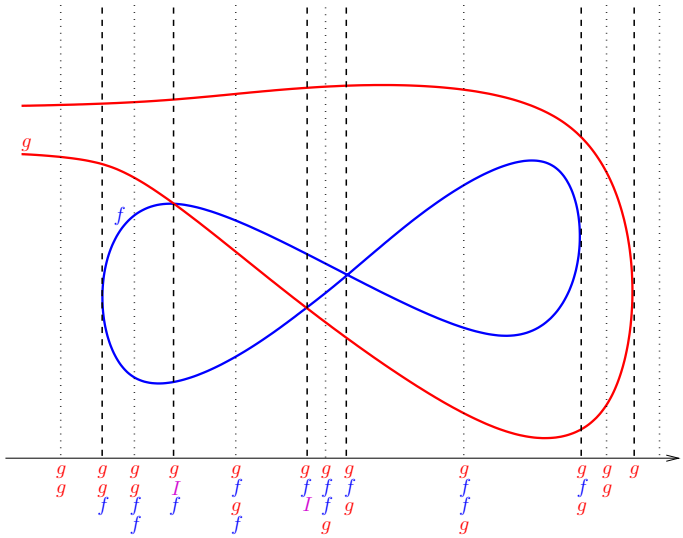


Figure 2.5. Analysis of a pair of curves f and g: For each critical x-coordinate and for each induced open interval, we construct a *status line* that stores a string reflecting the intersection pattern of the two curves in increasing y-order along the line. The character 'I' in the string encodes an *intersection* of curves f and g.

- A purely univariate model has been proposed by Lazard et al. [LPT08], whose polynomial's coefficient type is GMP's type for arbitrary-length integers, while the boundary type uses MPFR. Real roots are isolated by using the *interval Descartes* method taken from RS [14], [RZ03]. The refinement of their intervals applies quadratic refinement by Abbot [Abb06]. There is no choice of number types.
- The SYNAPS project [16] also implements a univariate model for whose real root isolation several approaches are available: Using Sturm sequences, using sleeves (i. e., *lower* and *upper* bounds on the polynomial), and several implementations for continued fractions [TE08] (some with enhanced support from the NTL [13]). Again, SYNAPS defines the number types.
- CGAL implements a univariate algebraic kernel class-template called

  ```
  Algebraic_kernel_d_1< AlgebraicRealRep, RootIsolator >
  ```

  See [HL07] for details. It has its origin in EXACUS' NUMERIX library. Its flexibility consists of the parameters: The first allows to choose the representation of the algebraic real type while the second determines the method for real root isolation. From the first parameter it also deduces the type of the univariate polynomial, its coefficient type, and the boundary type of the isolating intervals.
  The authors provide different choices for each parameter: For algebraic reals there exist `Algebraic_real_rep` using rational boundaries and `Algebraic_real_rep_bfi` that represents boundaries as intervals of bigfloats. Quadratic convergence for interval refinements is enabled by using `Algebraic_real_quadratic_refinement_rep_bfi`; see [Abb06]. None of them is restricted to a certain number type to represent the polynomial's coefficients. Several valid choices exist in CORE, LEDA, and CGAL; even CGAL's `Sqrt_extension` type is conceivable.

To isolate roots, there is the choice between the Descartes method as proposed in [CA76] and the bitstream Descartes method approximating the exact coefficients with ever-growing precision; see [EKK+05] and [Eig08]. If not stated otherwise, we select as default the bfi-version with Abbot's refinement and the bitstream variant for real root isolation.

The different models have been compared with each other on polynomials with different characteristics (increasing bit-length of coefficient, increasing degree, Mignotte polynomials, and more) [EHK+08]. However, there is no superior implementation for every input. We only want to remark that for large bit-lengths, the bitstream Descartes method scales best; we expect remarkable bit-lengths in the applications that are presented in Chapter 4 and Chapter 5.

Classes that model bivariate concepts are also available.

- CGAL's `Algebraic_kernel_for_circles_2_2` is a model of the *AlgebraicKernel_d_2* concept. It supports the algebraic computations that are demanded from CGAL's `Circular_kernel_2`. In particular, the types to represent polynomials are specialized to circles, and the types for real algebraic solutions are limited in its degree by 2. More details can be found in [BHK+06b] and [PT07]
- Very recently Michael Kerber has re-interfaced the ingredients of EXACUS' ALCIX library which now forms CGAL's first model of the *AlgebraicKernelWithAnalysis_d_2* concept, called `Algebraic_curve_kernel_2` (or `ACK_2` for short in this thesis). It refines a given univariate algebraic kernel. Central to this model are the analyses of curves and of pairs of them. This is very advantageous in cases where the kernel is mainly used because of these features. On other side, the resulting inevitable y-per-x-view also has some drawbacks with respect to other functionality: Due to this projection ansatz the representation of y-coordinates is not explicit but only approximative. That is, a symbolic, usually costly, computation is required when eventually accessing (`Get_y_2`) or comparing (`Compare_y_2`) arbitrary y-coordinates. Thus, it is recommended to check whether the projected application actually craves for these operations. Another example is the implementation of `Solve_2` that first analyzes two algebraic curves and then queries the corresponding pair to report the zero-dimensional solutions. This might pose a computational overhead, and one should carefully check whether it can amortize. As we mainly compute arrangements, we are not suffering from these problems. All required predicates provided by the `Curved_kernel_via_analysis_2` fully rely on the analyses of curves directly, using exhaustively the (combinatorial) y-per-x-information; see §2.4.4.
  It should be remarked, that implementing robust and efficient curve analyses is a research topic on its own and we desist from going into full detail. However, below, we review main results from this area of research and emphasize, in particular, highlights of CGAL's new fully-fledged bivariate algebraic kernel. For more details on the kernel's design, we refer to [EK08b].
- This reference actually describes a prequel of the previously described kernel. This prequel is still available internally, which allows to cope with still existing analyses of curves in EXACUS's libraries: Technically, the `Algebraic_curve_kernel_2` can be compiled in *wrapping mode*. Then it expects, besides the parameter for the univariate kernel, a second parameter: *CurvePair_2*. The parameter must be instantiated with an EXACUS-type that analyzes a pair of curves. Note that this type comprises

as nested type EXACUS' counterpart of the analysis of a single curve. In other words: The `Algebraic_curve_kernel_2` in wrapping mode mainly rewrites the deprecated interface of EXACUS-classes to fulfill the *AlgebraicKernelWithAnalysis_2* concept. Four such classes for pairs of curves exists:

- `Conic_pair_2` taken from CONIX, for algebraic curves of degree at most 2; see [BEH+02] and [Hem02].
- `Cubic_pair_2` taken from CUBIX, for algebraic curves of degree at most 3; see [EKSW06] and [Eig03].
- `P_quadric_curve_pair_2` taken from QUADRIX, for algebraic curves that represent projected silhouettes and intersections of quadrics. Such curves do not exceed a degree of 4; see [BHK+05] and [Ber04].
- `Algebraic_curve_pair_2` taken from ALCIX, for algebraic curves of arbitrary degree; see [EKW07] and [EK08a]. These classes are not maintained anymore. The code already has moved into the non-wrapping `Algebraic_curve_kernel_2` of CGAL.

The wrapping allows to still use the specialized analyses, in particular, for conics, and, as we see in Chapter 3, for projections of quadric intersection curves. Of course, the long-term plans are to consider the low-degree analyses as possible filters for the non-wrapping `Algebraic_curve_kernel_2`. However, this requires reliable performance comparisons and some developing time.

- A kernel that can deal with rotations is currently in an experimental status. The `Rotated_algebraic_curve_kernel_2` allows to rotate algebraic curve around a given point by angles α whose sin is a (nested) one-root number. For example, $\sin(45°) = \frac{1}{2}\sqrt{2}$. To do so, the kernel uses as `Coefficient` type CGAL's `Sqrt_extension` number type. Further details can be found in [BCW07] (for conics) and [Ker].
- Finally, there exists `Filtered_algebraic_curve_kernel_2` fulfilling the most refined *AlgebraicKernelWithAnalysis_2* concept. It tries to prevent costly algebraic computations, like resultant computations, by upstream filters using approximative bounding boxes. Details and results can be found in [Ker08].

Analyzing algebraic curves

The effectivity and efficiency of an exact bivariate kernel model can depend on the underlying analysis of algebraic curves. In particular, if there is no restriction on the degree, the exact analysis of algebraic curves and computing the solutions of a bivariate zero-dimensional polynomial systems are challenging tasks. The cad-approach, as presented in §2.1.6, states a generic solution.

If only aiming for the analysis of a single curve, it is very popular to restrict the computation to its topology; see [GVEK96], [GVN02], [SW05], and [MPS+, §3.6]. It is common that such approaches chose a generic coordinate system This avoids the handling of degenerate situations with respect to the coordinate system (e. g., covertical x-extreme points). Only some of them are available in software, and none of them fulfills the desired *AlgebraicKernelWithAnalysis_2* concept.

For more than one curve, most solutions[13] restrict the maximal allowed degree, for example circles [DFMT02], [WZ06], conics [Wei02], [BEH+02], [EKP+04], cubics [EKSW06],

[13] Some of them actually do not focus on the analysis of curves, but have to do it somehow in order to support arrangements of theses curves.

and projected intersections of quadrics [BHK+05]. There also exist solutions not restricted in the degree, but specialize for a certain input, namely Bézier curves [HW07] and non-singular algebraic plane curves [KCMK00], [Wol03]. As mentioned before, some of them can be used to define a model of the *AlgebraicKernelWithAnalysis_2* concept. However, there are only two implementations, that pose no restriction on the input curves: SYNAPS [16] claims to fulfill the the most powerful algebraic kernel concept; however, detailed information and access to the implementation is missing. The choice of number types is fixed.

The second one is the matured implementation in CGAL's `Algebraic_curve_kernel_2`. This is the most generic implementation and full details are given in a sequence of publications [Ker06], [EKW07], [EK08a], [Ker]. The solutions has several advantages. It

- fulfills the *AlgebraicKernelWithAnalysis_2* concept,
- has no restrictions on the input, that is, curves can have arbitrary degree, and contain degeneracies, like singularities, covertical intersections, vertical asymptotes, and isolated points, and
- is available in CGAL.[14]

Thus, its key contributions consist in the exact topological and geometrical analysis of single arbitrary real algebraic curves and pairs of them.

Its efficiency is established by several levers. One is an extensive caching strategy, another reason is the lazy-evaluation scheme, that is, certain results are only computed on demand and then stored for further queries. However, the main lever for efficiency is a clever combination of (unavoidable) exact computations, like resultant and greatest common divisor, with certified numerical (filter) methods, for real root solving. The chosen approximative methods often replace usually costly symbolic computations, while still guaranteeing the correctness of the overall result. The central approximative tool is the bitstream Descartes method (see §2.3.4) for the square-free case, and its m-k-variant for non-square-free polynomials. It is used to compute the local topology of a curve at some algebraic x-coordinates α, by mainly isolating the real roots of $f_\alpha := f(\alpha, y) \in \mathbb{R}[y]$, where f is the defining bivariate polynomial of a curve. The value α is chosen among the x-coordinates of the curve's critical events which are usually of non-trivial degree, and rational values in between. How to realize this technically is described in §2.3.4. However, there are still cases, where such approximate methods fail. For exact symbolic computations, the *Sturm-Habicht sequence* (see §2.1.1, also known as *(signed) subresultant sequence* [BPR06, §4]) is used. In fact, it is the computation of this sequence that mainly limits the practicality of the approach for higher degrees. A key goal for the future is to replace the resultant computation with a modular version, as it it already done for the gcd; see [Hem08, §2.3]

An important information is, that the obtained analyses are expressed with respect to the original coordinate system, that is, they do not expect the input curves to be in general position. However, an internal change of coordinates (a shear; see §2.1.4) can be applied, for example if the curves have vertical asymptotes or covertical critical points. A subsequent *back-shear* step recovers the original geometric information from the sheared version. Besides the polynomial sequences, it is the shear-and-back-shear approach that has significant influence on the running time. Ideas to avoid the change of coordinates more often might be implemented in a future version.

[14] Contained in an internal release, but subject to be publicly available with one of the next official releases of CGAL.

Our choice In our central chapters we are demanding for bivariate algebraic kernels with analyses that handle curves of degree 4 and even more. For that purpose, we mostly rely on CGAL's new `Algebraic_curve_kernel_2`, especially in Chapters 4 and 5, while the experiments in Chapter 3 are still with respect to the quadric-specific analyses of projected curves implemented in EXACUS' QUADRIX library.

The actual reason why we are demanding for such kernels is to compute arrangements of algebraic curves. We discuss arrangements in §2.4, in particular two-dimensional ones. For that purpose a set of geometric types and operations is required. A generic implementation providing these is presented in §2.4.4; it relies on analyses of curves.

2.3.4. Interfaces for the bitstream Descartes method

We close our discussion of the algebraic tool kit with technical details on how to interface the bitstream Descartes method. In fact, there are two muffs to couple. For both the generic programming paradigm does a good job. The implementation of the bitstream Descartes method (BDM) provided by Arno Eigenwillig maintains a subdivision tree whose nodes and leaves represent intervals enhanced with sign variations. For a given polynomial, the tree is explored by interfacing the polynomial's inexact coefficients with an instance of a model that fulfills the *BitstreamDescartesRndlTreeTraits* concept. We first present the concept, followed by a list of available models. On the other side, a potential user is expecting a very simple interface to get the isolating (and refineable) intervals of the real roots for a queried polynomial with bitstream coefficients. We finally discuss solutions how to interface these pieces of information.

The traits concept for the bitstream Descartes method

An instance of a traits class modelling a polynomial with bitstream coefficients and fulfilling the *BitstreamDescartesRndlTreeTraits* concept is expected to provide the following types.

`Coefficient` The model-specific coefficient type supplied during construction.

`Integer` A type for infinite-precision integer arithmetic equipped with `operator>>`, and `operator<<`. Examples are `leda::integer` or `CORE::BigInt`.

`Boundary` Instances of this type are used to express computed interval boundaries. Examples are `Exact_float_number< Integer >`, `leda::rational` or `CORE::BigRat`).

It is also required to define a small set of functors related to the types which mainly ensure that one can *approximate* a `Coefficient c` to any arbitrarily small absolute error $2^{-p}, p \in \mathbb{Z}$ and to deliver that approximation scaled with 2^p as an `Integer i`. Another expected functor is responsible to *locate the leading 1-bit* in the bitstream of the polynomial's leading coefficient.

The main functors are accessed only once for a single polynomial. This enables that the providing instance can maintain an internal status, An example is to hide some non-trivial approximation or evaluation process. This is sometimes the reason that enables the isolation at all; below, we present such a model. Finally, there is also a functor to *convert* the internal representation of the intervals' boundaries using two `Integer` and one `long`, to the user-supplied type `Boundary`.

Models

The first model that fulfills *BitstreamDescartesRndlTreeTraits* actually wraps a polynomial f, whose coefficients are integral and exactly known. At first glance, this strategy seems weak-minded. Why do we not use all pieces of information that are available? The answer is simply that not all information might be required. Remember that the isolation counts the number of sign changes of a polynomial in order to determine a bound on the number of real roots in an interval. But computing a sign only needs a large precision if it is zero or close to zero. In numerically more stable situations less precision usually suffices to compute the correct sign. Thus, the bitstream Descartes method first ask for a rough approximation of the coefficients (each normalized to be contained in $[-1, 1]$), and demands for more bits only until it is able to decide the Descartes test. Only in degenerate or near-degenerate cases, full precision is essential. For further details, we refer to [HL07] and [EHK$^+$08], that also contain various sets of experiments, even in comparison with other real root isolators.

The second model isolates the real roots of a polynomial with true non-rational coefficients, namely $f_\alpha := f(\alpha, y) \in \mathbb{R}[y]$, where $f \in \mathbb{Z}[x, y]$, and $\alpha \in \mathbb{R}$ in integral interval representation $\alpha \widehat{=} (p; I)$. We identified this setting in §2.3.3 among the task to analyze algebraic curves. Remember that I is refineable to arbitrary small length, which opens the door to approximate f_α's coefficients to any precision using interval arithmetic. An instance of such a traits is constructed from f and α and keeps the current approximation of α as internal status. In addition, the traits instance maintains a map to cache already computed approximations if needed for another coefficient. This is basically crucial, as it is recommended to only provide the number of bits of a coefficient currently requested by the bitstream Descartes method. Otherwise, too much precision can have a negative effect on the method's performance. Although the Descartes test definitely computes the correct result, it will spend too much time due to overwhelming precision and second, computing many bits is also a costly task on its own.

Very recently, a new generic model has been added: `Bitstream_coefficient_kernel`. It implements all necessary functions in terms of two simple operations on the `Coefficient`:

- given a coefficient c, compute its approximation as interval of `Bigfloat` numbers of a demanded precision
- check whether $c = 0$

Observe that the second actually contradicts the "bitstream" philosophy, but sometimes, it is possible (by filters or symbolic computations) to decide this test. In the case that this test is available, the model is able to support the computation of a stronger starting interval for the actual real root isolation. We remark that the previous two models already rely on this wrapper.

In the spirit of the second model, we present in §5.4.2 (page 228 ff) another class that models the *BitstreamDescartesRndlTreeTraits* concept in order to isolate the real roots of a trivariate integral polynomial whose x- and y-coordinates are substituted with algebraic numbers.

Maintaining the subdivision tree

We have learned that the Descartes method can be modelled as a binary subdivision tree whose correct traversal is essential in some cases, for example, in the m-k-variant.

Thus, besides the model of the *BitstreamDescartesRndlTreeTraits* concept, a class is required that implements Algorithm 2.9 or one of its variants. It is responsible to initialize the subdivision tree and to update it with respect to the computed sign variations. That is, for the standard approach, it applies breadth-first search until only intervals with sign variation 0 or 1 are left, while for the m-k-variant it also has to check the additional termination conditions; see §2.1.2. The polynomial itself is interfaced by the user with a proper instance of a bitstream traits. He actually does not care about any internal tree maintenance. In contrast, he is finally aiming for basic interests such as the number of real roots, the left and right boundaries for the isolating intervals, and a lever to refine each. For certain variants, for example, the m-k-method, an extended set of information is expected. We exemplary mention to check whether an isolating interval surely contains a simple or multiple root, or which interval contains the multiple root.

CGAL's `Bitstream_descartes` class is a model of CGAL's *RootIsolator* concept, that is, it can be used as a root isolator in the generic univariate algebraic kernel that we introduced in §2.3.3. It extensively uses C++ derivations and virtual functions in order to specialize with respect to some variants. For each variant (containing the standard and the m-k-method) an individual constructor exists. Variant-specific base classes ensure the maintenance of the subdivision tree with respect to the constructed instance.

Access to information is given by some self-explaining members: `number_of_real_roots`, `left_boundary(int i)`, `right_boundary(int i)`, and `refine_interval(int i)`. Internally, virtual functions dispatch among the different variants, which ensures that (the correct leaf of) the correct tree is accessed. Calls to the members `is_certainly_simple_root(int i)` and `is_certainly_multiple_root(int i)` are only allowed in case the m-k-variant constructor has been used. Otherwise, virtual functions look-ups indicate an error. The detection of more than one multiple root by the m-k-variant triggers to throw a C++-exception. It can be caught in order to trigger a different way, for example, using a shear.

As final note, we mention that for the analysis of an algebraic curve there exists a special back-shear variant [Ker06]. In §5.4.2 we present a variant, that is actually abusing the interface to merge various root isolators. But for now, we skip further details.

2.4. Arrangements

Arrangements are widely known in the field of computational geometry. They have been studied since decades serving as key ingredients for many theoretical results and practical applications.

Definition 2.37 (Arrangement). Given a d-dimensional connected space $\mathcal{D}$ and a finite set of geometric objects O that reside in $\mathcal{D}$. The *arrangement* $\mathcal{A}(O)$ is the subdivision of $\mathcal{D}$ induced by O into a finite number of relatively open cells of dimension $0, 1, \ldots, d$. A d-dimensional cell in $\mathcal{A}(O)$ is a maximal connected subset of $\mathcal{D}$ that is not intersected by any object in O.

The restriction to finite number of cells is quite natural, as otherwise, the description of a subdivision with an unbounded number of cells can only be established if it has a special structure, for example, a periodic behavior.

First research on arrangements concentrated on theoretical results especially on linear arrangements [Ede87]. It turned towards the analysis and computation of arrangements

induced by curved objects; see [SA95], [Hal04], [AS00]. While most of these results concentrate on theoretical aspects, practicality issues also came to the fore of research in past years. This comprises to strengthen robust implementations and to improve the usability of arrangements. A detailed survey is given in [FHK$^+$, Chapter 1] that we recommend for further reading. Our contribution pursues the work on arrangements in this spirit, and especially enlightens the specialty of two-dimensional arrangements in a three-dimensional world.

Arrangements are a popular and important (sub)structure in various fields. Well-known examples are computer vision, robot motion planing, geographic information systems, and computer-aided biology; see for examples [HS94], [HS98], [FH00], [CL07]. These and other applications benefit from big advantages of an arrangement: It provides exact access to a continuous problem in discretized chunks, that is, it models the decomposition of $\mathcal{D}$ into a finite number of (open) cells, whose boundaries are described with a finite number of elements. The representation is complete, that is, no detail for a given input is missing. Often, problems can be reduced to operations on arrangements, for example, existence decisions can be expressed in terms of point location. Or the theoretical complexity analysis on arrangements can serve as a source of bounds, if one can formulate another problem in terms of a special arrangement, or just one of its cells. One technique to transform a problem into "arrangement"-lingo is duality, that is surveyed in [dBvKOS00, Chapter 8]. We desist from collecting the wide range of theoretical results on arrangements in order to concentrate on the aspects of algorithm engineering when aiming for a generic and efficient implementation. Questions here are: How to cope with degeneracies? How to compute an exact result?

Let us start with arrangements where $\mathcal{D} = \mathbb{R}^3$.

Problem 2.38 (Three-dimensional arrangement). Given a set of surfaces S in $\mathbb{R}^3$, compute the arrangement $\mathcal{A}(S)$ induced by S, that is, compute a representation of the subdivision of $\mathbb{R}^3$ induced by S. The resulting cells of dimension 0, 1, 2, and 3 are called *vertices*, *edges*, *faces*, and *volumes*.

We want to spot that the definition makes no assumptions on how surfaces are defined, except that they induce a finite number of cells. In §2.1.5 we introduced algebraic surfaces which form the central geometric input objects throughout this thesis. We are not aware of robust code that implements Problem 2.38 for such (generic) surfaces. We can restrict to the linear case. An arrangement induced by the closure of half-spaces under boolean set operations in constituted by CGAL's `Nef_3` package; see [HKM07] and [Hac07]. Thus, the implementation supports non-manifold situations, as for example tight-passages required in robot motion planning. The basis of this implementation goes back to Nef's seminal book on polyhedra [Nef78]. In its representation, each vertex is surrounded by a so-called *sphere-map* which encodes the local neighborhood around the vertex. Elements of different neighborhoods are connected with respect to the topology induced by the given half-spaces. These connections are stored in a structure called the *Selective Nef Complex (SNC)*. Although this idea is promising to work also for curved surfaces, we do not follow this strategy in Chapter 5, but use elimination theory, which leads us to two-dimensional arrangements.

Problem 2.39 (Two-dimensional arrangement). Given a set of curves $\mathcal{C}$ in $\mathcal{D}$, with

$\dim(\mathcal{D}) = 2$, compute the arrangement $\mathcal{A}(S)$ induced by C, that is, compute a representation of the subdivision of $\mathcal{D}$ induced by $\mathcal{C}$. The resulting cells of dimension 0, 1, and 2 are called *vertices*, *edges*, and *faces*.

In contrast to the three-dimensional case, we here let the choice of the actual domain $\mathcal{D}$ open. For this background information we set $\mathcal{D} = \mathbb{R}^2$ and interpret it as the xy-plane, a quite natural setting when considering arrangements. However, Chapter 4 interprets $\mathcal{D} = \mathbb{R}^2$ only as a special case of a two-dimensional parametric surface; see Definition 2.30. Similar to Problem 2.38, the type of curves is not specified, however they respect the usual definitions.

Definition 2.40 (Curve). A curve is a function $\gamma : I \to \mathcal{D}$ with

1. I is an open, half-open, or closed interval with endpoints 0 and 1;
2. γ is continuous and injective except for *closed* curves where we allow $\gamma(0) = \gamma(1)$;
3. if $0 \notin \mathcal{D}$, that is, the curve has no start point, the curve *starts at infinity* or more precise: $\lim_{t \to 0+} |\gamma(t)| = \infty$. We have a similar condition if $1 \notin \mathcal{D}$;

The task at issue is to transform the continuous problem into a finite, discretized representation by means of combinatorial algorithmic steps. As already learned in §2.2, such steps are driven by evaluations of predicates, that is, by continuous functions whose output is discrete. This simplification of the continuousness quickly opens the door to wrong results, especially in numerically unstable situations. Before we present the two main algorithmic (and combinatorial) approaches to compute $\mathcal{A}(\mathcal{C})$ in §2.4.2, we introduce in §2.4.1 the data structure that is used to represent a two-dimensional arrangement.

2.4.1. The Doubly-Connected-Edge-List (DCEL)

A well-known data structure to represent two-dimensional subdivisions is the so-called *doubly-connected-edge-list*, or DCEL for short [dBvKOS00, §2.2]. This data structure allows easy and convenient constructions, updates, and queries of subdivisions. We give a short introduction to the DCEL, while [Ket07] gives full details and references to similar structures.

A DCEL (mainly) consists of three types of kinds or records, namely *vertices*, *halfedges*, and *faces*. It provides methods to insert and delete records, Euler operators, and iterators to traverse the structure. All records of one type are stored independently from other types in either double-connected lists or containers. Each single record can be accessed by a handle (see for example [Hac07]). Each item also stores its own adjacency and incidence relations with respect to other records. In addition, each vertex and each halfedge is associated with geometric information.

Halfedge Central items to the structure are halfedges. A halfedge is directed and always coexists with its *twin* halfedge of opposite direction. The two twins are connected by pointers, and as a pair they represent a geometric *curve* that is not intersected in its interior by any other curve stored along with halfedges in the DCEL-instance.

The directed halfedge points to a vertex. Is also has an implicit incident *face* to its *left* which is usually referenced by a pointer. Both pointers are not required by a minimal DCEL that optimizes storage. However, for reasons of convenience and efficiency, it is recommended and usual to include them. In contrast, a pointer to the

next halfedge that has the same incident face is inevitable. It has to hold, that the origin of the next halfedge is identical to this halfedge's destination. In fact, the next and the twin pointer are the only mandatory ones, all other pointers are optional — though recommended.

Vertex A vertex represents a zero-dimensional feature of the decomposition, that is, it is associated with a geometric *point*, be it the end of a curve, the intersection of curves, or even both.

A pointer stores an incident *halfedge* that directs to the vertex. All halfedges targeting a vertex can be connected by a (bidirectional) circular linking. Although not part of the original DCEL-design, we, that is, when using the DCEL for arrangements, allow that no halfedge is incident to a vertex. In this case, the pointer is simply `NULL`. However, such an *isolated vertex* is not slobbing around. A *face* pointer (which is `NULL` otherwise) indicates the face that contains the isolated vertex.

Face A face represents a *two-dimensional connected set* implicitly, that is, no actual geometric object is associated with it. To obtain geometric information, a face is surrounded by a circular list of halfedges that have the face to their left. The linking is established with the help of the halfedges' *next* pointers, or more precisely: Each face is surrounded by halfedges that wind in counter-clockwise order along the outer boundary of the face. We call it the *outer connected component of the boundary (OCCB)*. The face knows an occb-pointer to one these halfedges.

Nevertheless, this simple design actually allows to only represent decompositions whose faces are simply connected. But in general, two-dimensional arrangements can contain faces that are not simply connected; for an example we refer to the already mentioned isolated vertices, or to Figure 2.6, where faces F_2 and F_3 are completely inside F_1. Face F_0 even surrounds all other faces.

Definition 2.41 (Hole). A connected set H is called a *hole of face* F if it makes F locally non-simply connected. That is, there is a simply connected subset of F that gets non-simply connected when we remove H from F.

Holes can be two-, one-, and zero-dimensional, and their number can be arbitrary, but finite. In order to support the different cases, each face maintains two additional lists: One for isolated vertices and one for *inner connected components of the boundary (ICCB)*. An inner component of a face F is similar to its outer counterpart, namely a list of halfedges having F to their left. However, they wind in clockwise order. This way, the cycle of twin edges describes a two- or even one-dimensional set that is *excluded* from F. In the example of Figure 2.6, the inner CCB defined by E_2 removes a two-dimensional set, while the inner CCB defined by E_3 is only one-dimensional.

Remarks.

- Actually, there is no geometric way to distinguish outer and inner CCBs. By topological inversions each CCB could become outer.
- However, for the plane, the common convention is to define the CCB as outer which winds, as written, (once) counter-clockwisely around the normal-vector of the plane

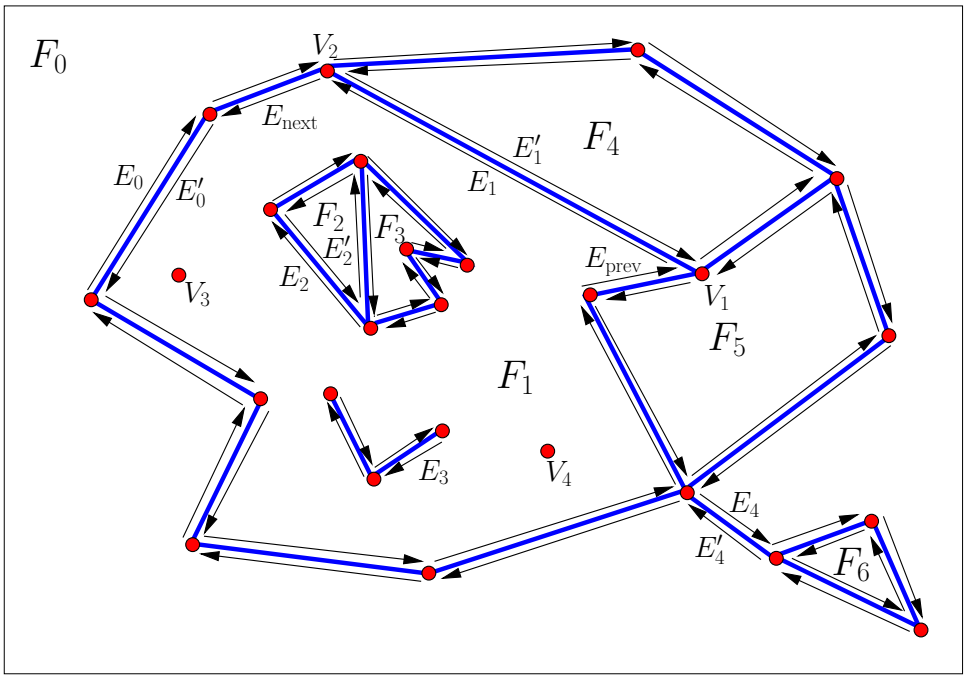


Figure 2.6. How to use the DCEL to represent a planar arrangement (of interior-disjoint line segments): The unbounded face F_0 has a single connected component that forms a hole in it. This hole is separated with a halfedge-cycle containing E_0, a so-called inner CCB (connected component of the boundary) of F_0. The hole itself comprises several faces, for example F_1, whose outer CCB is the halfedge-cycle defined by E_1. Along this outer CCB, E_1 is preceded by E_{prev} and succeeded by E_{next}. The halfedge E_1 connects V_1 with V_2, while together with its twin E_1' it represents the line segments that connects the points associated with V_1 and V_2. This segment separates F_1 from F_4. E_1' defines the outer CCB of F_4. Note that, in contrast to E_1 and E_1', the edges E_4 and E_4' do not separate different faces. The face F_1 also has *holes*: The two-dimensional hole separated with F_1's inner CCB defined by E_2, the one-dimensional hole separated with F_1's second inner CCB defined by E_3, and two isolated vertices V_3 and V_4. All other faces only have a single outer CCB.

located at an interior point of the face, that is, in the left area of the CCB (as the halfedges have their incident face to the left).
- In Chapter 4 we use a DCEL for non-planar two-dimensional subdivisions. For such, to characterize CCBs by windings makes less sense. Thus, we next introduce the nesting graph in Definition 2.42.
- A general purpose halfedge data structure is presented in [Brö01a]. It discusses these and other aspects.

Definition 2.42 (Nesting graph). We construct the *nesting graph* of faces. Nodes of the graph correspond to faces, while we add an oriented edge from node `f1` to node `f2` if `f2` is separated from `f1` by an inner CCB of `f1`. That is, there is a twin of halfedges `e1` and `e2` (with `e1->twin() == e2` and `e2->twin() == e1`) such that `e1->face() == f1` and `e2->face() == f2` and `e1` belongs to an inner CCB of `f1` (and `e2` belongs to an outer CCB of `f2`).

The DCEL-representation for each decomposition of the plane with bounded curves (and points) always has a face that has no outer CCB. This face corresponds to the plane having holes in it. Thus, the nesting graph of such a decomposition is a tree, whose root is the face without outer CCB, the *outermost face*. The root's direct children are the faces separated by the inner CCBs of the outermost face. Note that a single inner CCB can result in more than one children; see Figure 2.7. Actually, one could extend the nesting graph with special nodes for isolated points and one-dimensional holes. In fact, adding them would complete the representation of the DCEL as graph. However, for our purposes, they are irrelevant. We emphasize that the way CCBs are assigned to the list of outer and inner CCBs of a face fully determine the nesting graph's edges. As written, by topological inversion we can make every face the root of the tree, though, this results in another nesting graph (with other assignments of CCBs to the list of lower and outer CCBs of a face).

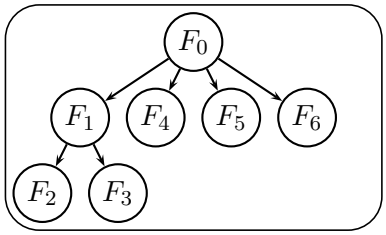


Figure 2.7. Nesting graph (here: tree) for the DCEL of Figure 2.6

We already mentioned that curves stored in a DCEL are required not to pair-wisely intersect in their interior. Consider a closed curve, for example $p = \gamma(0) = \gamma(1)$ that would be embedded by a pair of halfedges. However, each halfedge forms a self-loop, that is, it points to its originating vertex. This implies, that there must be a vertex, which must be constructible on γ. Although self-loops are not forbidden by design, algorithms constructing a DCEL avoid them, for example, as they require to split curves into x-monotone sub-curves. Such a split implies that each connected component of a boundary consists of at least two halfedges. We especially want to single out this fact for each outer component and also for inner components that describe a one-dimensional set. We

also remark, that a face that does neither contain an inner CCB nor an isolated vertex is simply connected. The DCEL, as described here, suffices to support an arrangement that is embedded in an orientable surface which is homeomorphic to an (open) disc. For arrangements on parametric surfaces, that we discuss in Chapter 4, we have to extend the DCEL further.

We also benefit from the DCEL's advantages of easy traversals of its items and the possibility to support their efficient overlay [dBvKOS00, §2.3]. In general, the DCEL, also known as Halfedge-Datastructure (*HDS*), is widely known and used in computational geometry and inescapable, for example, in two- and three-dimensional triangulations. This also holds for CGAL. However the `Arrangement_2` package uses a specialized version. In order to unify these difference the two implementations are currently in a redesign process [KC08]. Its goal is to provide an implementation that serves throughout all packages of CGAL that require an HDS. The main improvement is the introduction of optional border-edges and *HalfedgeCycles*. Halfcycles are intended to unify outer and inner connected components of the boundary. In Chapter 4 we only touch these extensions in our discussion, as they are not yet used productively. For three-dimensional regular complexes[15] Bru and Teillaud suggested another extension of CGAL's `HalfedgeDS`, called *cellular data structure* [BT08].

2.4.2. Computing planar arrangements

Unfortunately, the curves in the given input $\mathcal{C}$ are usually neither x-monotone nor disjoint in their interior, that is, the input typically consists of non-x-monotone (or even vertical) curves that intersect or (partially) overlap. We wish to construct a DCEL that describes the subdivision $\mathcal{A}(\mathcal{C})$ induced by $\mathcal{C}$ using only weakly x-monotone curves, see Definition 2.43.

Definition 2.43 (Curve continued). We extend Definition 2.40.

4. A curve γ is called *weakly x-monotone*, if for $t_1 < t_2, t_1, t_2 \in I$ it holds that $\gamma(t_1) <_{\text{lex}} \gamma(t_2)$, where $<_{\text{lex}}$ denotes smaller in lexicographic xy-ordering. Observe that also vertical curves are classified to be weakly x-monotone.

Such a decomposition has the advantage that maintenance is simplified, but also enables us to easily extend the DCEL towards a vertical decomposition [dBvKOS00]. Algorithm 2.11 gives a naive construction for a DCEL.

Algorithm 2.11. Construct DCEL naively

INPUT: Set of curves $\mathcal{C}$ in $\mathbb{R}^2$
OUTPUT: DCEL that represents $\mathcal{A}(S)$

1. Split each non-x-monotone curve of C into weakly x-monotone sub-curves $\mathcal{C}'$.
2. Compute all intersections of curves in $\mathcal{C}'$ and subdivide them such that they are interior disjoint
3. Use Euler operators to modify the DCEL with respect to the split input. Optional pointers might link to the originating curve(s) of $\mathcal{C}$.

This approach, however, requires a quadratic number of intersection tests, and does not exploit proximity of curves for intersection tests, or actually non-proximity to avoid

[15] A three-dimensional regular complex is a finite decomposition $\mathbb{R}^3$, whose cells are pairwise interior disjoint and the boundary of a cell consist of the union of other cells [ES94].

them. In practice, two other approaches are more common to construct a DCEL. Similar to the naive approach, their first step consists in breaking the input into weakly x-monotone curves. Thus, we henceforth assume that $\mathcal{C}$ consists of such curves.

The sweep line approach The basic idea goes back to Bentley and Ottmann [BO79] who gave an algorithm to count and compute the intersections of line segments. Luckily, by observing the execution path of the algorithm it is possible to construct the induced DCEL; we give more details below.

We give a sketch of the algorithm that works for line segments $\mathcal{C} = \{s_1, \ldots, s_n\}$ that fulfill the general position assumption. More detailed descriptions, which also discuss the degenerate cases, can be found in [dBvKOS00, Chapter 2], or [MN00, §10.7].

The main idea is to sweep with a vertical line, the *sweep line*, from left to right over the plane. At every position the sweep line is intersected by some segments of input in a certain order. The crucial observation is that this order only changes at a finite number of *events*, which are exactly the positions where the topology of the segments intersecting the sweep line changes, and thus, also the topology of the induced arrangements: These events are the minimal and maximal ends of segments and intersection points of segments.

The sweep line algorithm maintains two dynamic data structures. The *status-line*[16] L represents an intersection pattern of the input segments with the sweep line at its current position. It is empty at the beginning of the sweep and also exhausted when the sweep ends. Events are maintained in a priority queue that sorts its entries lexicographically by coordinates. This *event-queue*[17] Q is initialized with the minimal and maximal ends of the input segments. The sweep of the line actually consists in extracting at any time the next minimal event from the event-queue and to update the structure with respect to the local situation at the event. The process keeps two invariants valid:

1. Events with smaller lexicographic coordinates than the current event (*to the left of the sweep line*) have already been discovered and handled.
2. *At least* the following events are stored in the event-queue: (a) All endpoints of input curves that have greater lexicographic coordinates than the current event (*to the right of the sweep line*) and (b) the next intersection of two segments that are *currently adjacent* in the status-line.

Observe that at the beginning of the algorithm, the invariants are fulfilled by how we initialized the dynamic structures. Algorithm 2.12 describes how to sweep over the line segments, actually, its main loop discusses the (possible not required) updates of the structures when *sweeping over the current event*.

[16] Some texts call the status-line also *Y-structure*.
[17] The event-queue is also referred to as the *X-structure*.

Algorithm 2.12. Sweeping line segments

INPUT: Set of line segments $\mathcal{C}$ in $\mathbb{R}^2$

OUTPUT: Lexicographic processing of events and how they are connected with sub-curves

- Insert minimal and maximal point of each segment in $\mathcal{C}$ into Q
- While the Q is not empty
  - Extract Q's current minimal event ev (and remove it).
  - If ev is the minimal endpoint $p_{\min}$ of some s_i, we insert s_i into L. This requires to compute the relative vertical alignment of $p_{\min}$ with the segments already existing in L. We either hit a segment s_j or $p_{\min}$ is positioned in between segments s_{below} and s_{above} (if existing). In the former case we have to compare whether s_i is below or above s_j right after their intersection at $ev = p_{\min}$, which also defines now unique s_{below} and s_{above}. Check if s_i intersects *to the right of* ev with s_{below} and if so, insert the next intersection into the Q. Do the same for s_i and s_{above}.
  - If ev is a maximal endpoint $p_{\max}$ of some s_i, then s_i is located between s_{below} and s_{above}. We remove s_i from L and check whether s_{below} and s_{above} intersect lexicographically larger than ev. If so, we insert the next intersection into Q (if not already existing).
  - If ev is the intersection of some s_i and s_j (where their order in L is: $s_{\text{below}} < s_i < s_j < s_{\text{above}}$), we exchange them in L. Then, s_j is *above* s_i and we check next for a future intersection of s_{below} and s_j and for a future intersection of s_i and s_{above}. If such exist, we insert them into Q.

Remarks.

- Note, that in each step s_{below} and s_{above} might not exist. If so, the corresponding cases can be ignored.
- The algorithm neither reports intersection points nor constructs a DCEL. However, having a continuously look on the algorithm's executional steps by some *entity*, this entity can simply extract intersection points or construct the DCEL that emerges to the left of the sweep line. Technically, the visitor design pattern [GHJV99] describes such entities. We refer to §2.2.3 that discusses details on how CGAL's `Sweep_line_2` class is combined with visitors for different purposes.
- The algorithm assumes general position of the segments. However, by carefully extending individual steps it is possible to handle isolated points, vertical and/or overlapping segments, more than two segments running through a common point, or events that share a common x-coordinate (i.e., *covertical events*). LEDA's and CGAL's implementation mind all these degeneracies.
- The running time of the algorithm is $O((n+k)\log n)$, where n is the number of input segments and k the number of intersections. It requires space $O(n+k)$, which can be improved to $O(n)$: We only have to revise Q from future intersections of segments that just lost their adjacency in L. When computing a DCEL, this strategy is not advised as the output needs space $O(n+k)$ anyhow, and the re-computations of intersections and maintenance operations for the event-queue harm the practical performance; see again [MN00, §10.7].

Already Bentley and Ottmann experienced the fact that their idea is applicable to any set of x-monotone curves, such as half-circles. A generalized description is given in [SH89].

In contrast to the linear case, some difficulties must be tackled:

Problem 2.44 (Sweeping non-linear curves).

- Two non-linear curves can intersect more than once.
- The order of two non-linear curves to the right of an intersection is not always the reversed order the curves had to the left of the intersection.

However, solutions to both problems exist. For the first, it actually suffices to only compute the next intersection. However, it is encouraged to augment the event-queue with all future intersection points of two non-linear curves, if available, as soon as they become adjacent in the status-line for the first time. Note that in the final Dcel all of them pop up anyhow.

A naive solution for the reordering of ℓ curves passing an event (at point p) is a comparison-based sorting. It consists of pair-wisely computing the order of y-coordinates of two such curves slightly to the right of the common intersection. However, this results in an algorithm with $O(\ell \log \ell)$ running time, while each comparison is also a task of non-trivial cost.

The reordering can be improved if one knows the multiplicity of intersection in the point for two such curves. This is, for example, the case for input that is supported by algebraic curves (see §2.1.4), if the intersection does not takes place at a singularity (which can be excluded). The precise definition of this value is given in [MPS$^+$]. Intuitively, the two curves change their relative vertical alignment when passing p, if the multiplicity is *odd*, while their order is preserved if the multiplicity is *even*. This leads to an easy combinatorial decision on how to update L. Based on these multiplicities there exists an $O(M\ell)$ algorithm that reorders ℓ (algebraic) curves passing through p, where M is the maximal multiplicity of intersection that occurs for two curves passing the point; see [BEH$^+$02] and [FHK$^+$, Chapter 1] for a more detailed proof. Even better, it is possible to remove M by constructing a *multiplicity tree*. The algorithm presented in [BK07] only requires time $O(\ell)$ relying on pair-wise multiplicities of intersections.

Abstracting from the curve-specific details, we can state a generic version of the sweep line algorithm.

Algorithm 2.13. Sweeping (weakly) x-monotone curves

Input: Set of curves $\mathcal{C}$ in $\mathbb{R}^2$

Output: Lexicographic processing of events and how they are connected with sub-curves

- Replace each curve $c \in \mathcal{C}$ by curves that represent a decomposition of c into (weakly) x-monotone curves
- Insert lexicographical minimal and maximal point of each (weakly) x-monotone curve in $\mathcal{C}$ into Q
- While the Q is not empty
  - Extract minimal ev event from Q
  - Remove all curves from L that end at ev
  - Reorder all curves passing through ev
  - Insert all curves into L that begin at ev, compute intersections for newly adjacent curves and insert them into Q

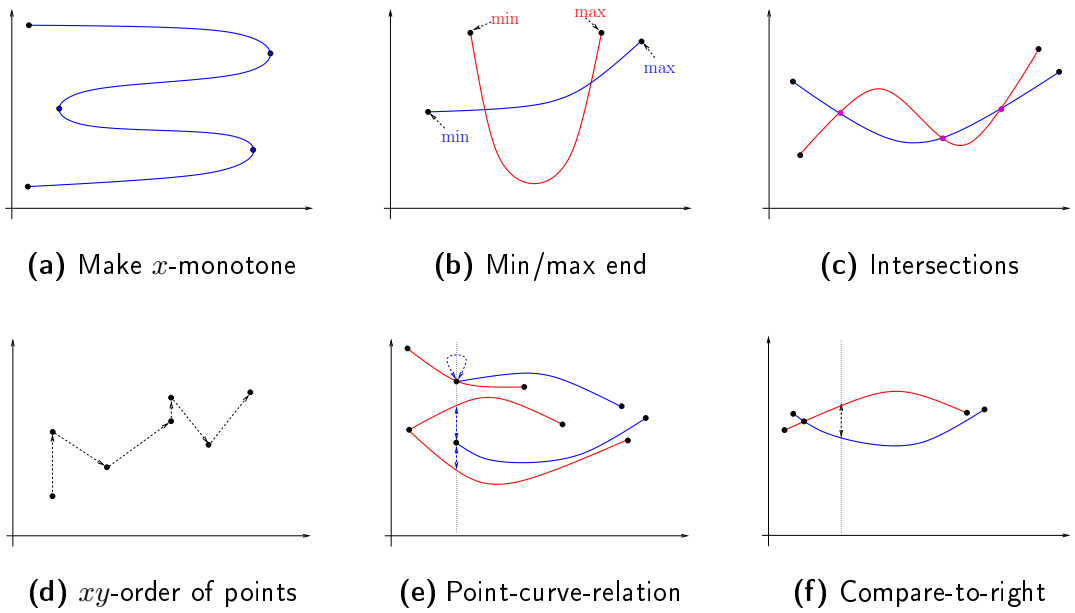
Having this generic sweep line algorithm, we next concentrate on the individual tasks

in each step, that is, we break down the approach into subtasks consisting of geometric predicates and constructions. As already mention, we require to decompose arbitrary one-dimensional input into (weakly) x-monotone pieces.

Make x-monotone Given a one-dimensional input object c, decompose it into weakly x-monotone curves. If other predicates expect stronger conditions than just weak x-monotonicity, it is the responsibility of this geometric construction to ensure them as well. We refer to such a split curve γ as a *sweepable curve*.

We next describe the predicates that are required to maintain the event-queue and to update the status-line when sweeping over an event.

Figure 2.8. Geometric constructions (a),(b),(c) and predicates (d),(e),(f) required for the sweep line algorithm



(a) Make x-monotone **(b)** Min/max end **(c)** Intersections

(d) xy-order of points **(e)** Point-curve-relation **(f)** Compare-to-right

Minimal/maximal-end Given a weakly x-monotone curve c, the predicates returns its lexicographical smallest (largest) point. They are used during initialization, to check whether a curve starts or ends at an event, and to determine the location of a starting curve in the status-line.

Compare-xy Given two points p_1, p_2, compare them lexicographically. We require this predicate to keep the event-queue sorted, and to check whether a curve starts or ends at an event.

Point-curve-relation Given an x-monotone curve c and a point p in the x-range of c, this predicate determines the relative vertical alignment of p and c, that is, whether p lies below, on, or above c at p's x-coordinate. In case of a vertical c, it returns whether p is below the minimal point of c, on c, or above the maximal point of c. In the sweep line algorithm, this predicate is used to locate the position of a curve that

starts at an event in the status-line. To do so, the curve's minimal end is compared with the curves already stored in the status-line. Of course, no such comparison is required if other curves end or pass the current event, as the algorithm remembers their position in L. That is, it knows where to insert new curves. If there are passing curves, the next predicate is required upon a starting curve.

Compare-to-right Given two weakly x-monotone curves c_1, c_2 that intersect at p. This predicate determines the relative vertical alignment of c_1 and c_2 after passing p, that is immediately to the right of p. The predicate is called to determine the location of a curve that starts at an event with passing curves, that is, we determine the position of the new curve (whose minimal end lies on a curve in the status-line) in the sequence of curves to the right of an event.

Intersections Given two weakly x-monotone curves c_1, c_2 compute their intersections. Usually the set of intersection is zero-dimensional, that is, it consists of a finite number of points. It might be helpful to also obtain the corresponding multiplicities of intersection (or at least their parities). In degenerate situations, the two curves may overlap. In such a case the construction is requested to compute all overlapping parts. Of course, the processing of an event resulting in proper updates of the dynamic data structures, also has to deal with overlapping curves. We omit these technicalities, as they are previously discussed elsewhere; see [MN00, §10.7].

The incremental approach The aggregated construction using the sweep line approach is very efficient, in particular when the number of intersection is relatively small, that is $k < O(\frac{n^2}{\log n})$. A drawback of the approach is that all curves must be known in advance, which some application do not provide, as new curves can arrive in an on-line fashion. For such cases, an incremental (and local) update should be privileged. Algorithm 2.14 gives a method that inserts a weakly x-monotone curve c into an existing (not necessarily empty) arrangement $\mathcal{A}$. Non-weakly-x-monotone curves are decomposed beforehand as in the sweep line approach.

Algorithm 2.14. Incrementally inserting a weakly x-monotone curve c

INPUT: (non-empty) arrangement $\mathcal{A}$; curve c
OUTPUT: refined $\mathcal{A}$ with inserted c

1. Split c into (weakly) x-monotone curves. For the next steps we assume that c has this property.
2. Locate the minimal end of c and either update the found vertex (locate position of curve in its circular list of incident curves), or split the found halfedge-pair, or insert a new vertex in the interior of the found face.
3. Traverse the *zone* of c, that is, all DCEL-items intersected by c. Whenever we detect an intersection of c with some vertex or some halfedge-pair, we split c into two subcurves c_{left} and c_{right}, update the vertex or the halfedge-pair accordingly, process c_{left}, and proceed with c_{right} until we reach c's maximal end.
4. Locate c's maximal end and proceed similar to what we did for c's minimal end.

Special care is needed when c overlaps with an existing curve in $\mathcal{A}$, or c completely lies in a face of $\mathcal{A}$. In the latter case, c must be inserted as a new hole in that face.

The subtlety for incremental insertion is that it requires point location, that is, given

a point p determine the DCEL-item to which it belongs. We shortly discuss point location in §2.4.3. The running time for incrementally inserting n weakly x-monotone segments is $O(n^2)$. Thus, for dense arrangements, $k \geq \omega(\frac{n^2}{\log n})$, the incremental approach theoretically (and practically) beats the sweep line approach which requires $O((n+k)\log n)$ time. However, the running time of the sweep line method is output sensitive. The proofs and more details on the incremental construction of arrangements can be found in [dBvKOS00, Chapter 8].

Needless to say, that both approaches can be combined. For example, the sweep line method is used to construct the DCEL for an initial set of curves, while it is augmented by curves arriving in an on-line fashion by applying the incremental algorithm. Or an initial dense arrangement is constructed by the incremental method, while later a set of curves that imply only a few new intersection are "swept" into the arrangement. Note that sweeping an arrangement of non-intersecting curves is a much easier task, theoretically and practically, as no intersection has to be computed and the event-queue is not altered at any time. All this flexibility on two-dimensional arrangements is offered by CGAL's `Arrangement_2` package that we present next. In its presentation, we also cater for how to delete curves in an existing arrangement.

2.4.3. Arrangements in CGAL

We next introduce CGAL's `Arrangement_2` package with various details. It is developed and maintained at Tel-Aviv University in the lab of Dan Halperin. During the package's lifetime, it always has been improved, while for CGAL version 3.2 a major redesign has been applied, that was mainly driven by Dan Halperin's students Ron Wein, Efi Fogel, and Baruch Zukerman. The "changelog" is reported in a sequence of publications: [FHH+00], [FWH04], [WFZH05], and [WFZH07b].

In this section, we present the `Arrangement_2` package of CGAL 3.2. that only supports bounded curves in the plane: It maintains a single unbounded face that contains all input objects, that themselves fit in the interior of a finite rectangular area. We show in Chapter 4 how newer extensions (CGAL 3.3) already enable unbounded curves, and how the restriction of the embedding surface to be a plane is removed (upcoming version of CGAL).

The `Arrangement_2` package implements the generic programming paradigm as explained in §2.2.1. This technique allows to separate the combinatorial and topological algorithms and data structures from whatever geometric objects are at hand. Central to the package are only a few classes. The main class-template is intended to represent a planar embedding of weakly x-monotone curves that are pairwise disjoint in their interior. It is instantiated with two parameters:

```
Arrangement_2< GeometryTraits_2, Dcel >
```

`GeometryTraits_2` This is the main parameter for the package, as it defines the type of geometric curves (and points) that induce an arrangement. It also implements basic operations on the types to support the arrangement's construction and maintenance. As a positive side-effect of this distinction, a developer with less experience in computational geometry, and arrangements in particular, can engage in the package with all its functionality for its own curves, as long as he provides a proper geometric-traits class for them. The list of required operations has been reduced over time

and finalized in the *ArrangementTraits_2* concept of CGAL version 3.2.[18] We present details of the concept and available models below.

`Dcel` This parameter determines the type (and specialties) of the underlying topological structure used to represent the planar subdivision. A default implementation is contained in the package and if it should be used, one even can omit to specify the argument to define the arrangement type. On the other side, a more experienced user is able to replace it, for example, to attach user-specific data to the DCEL-records.

A valid two-dimensional arrangement (of bounded curves) has one unbounded face. Each face, except the unbounded one, has an outer CCB (connected component of the boundary). The non zero-dimensional holes within a face are represented by a number of inner CCBs. The zero-dimensional holes (also known as isolated vertices) are stored explicitly. The latter two entities are not required to exist. The hierarchical order of holes and isolated vertices in a face is distinguished by graph- and edge-based structures.

The arrangement class-template provides all necessary capabilities to construct and maintain the DCEL that is extended with geometric data. Basic functions are available to access, to modify, or to traverse an arrangement. For example, all vertices, edges, and faces can be visited by iterators, or the halfedges of a CCB and the incident edges of a vertex can be circulated. The central modifiers are the *basic insertion* and *deletion methods*. It is possible to insert points or weakly x-monotone curves. For a new point, either a vertex for it already exists, then nothing happens, or it lies on an existing halfedge-pair, that is going to split, or it will be added as an isolated point in a face's interior. When adding a new weakly x-monotone curve, we distinguish four cases: Either it is inserted in a face's interior, its minimal/maximal point hits a non-face, or both ends hit a non-face (two possibilities). In every case the DCEL has to receive some modifications, for example, when short-cutting an inner CCB, a new face is constructed, and some CCBs must be adapted. Figure 2.9 explains the various cases. Similar modifications are required when removing an edge. The arrangement takes care of the correct order of modifications to transform the DCEL from one valid state to a new valid state that represent the new situation. The user who adds or removes the object does not even notice about all the details, at least not directly. Note that these operations implicitly modify the nesting graph of the DCEL.

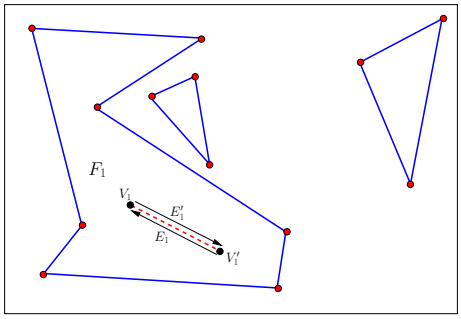
For the user's information on changes of the arrangement's structure, the package implements the *observer pattern* [GHJV99]. An observer receives notifications from a given arrangement-instance, for example, when a vertex is added or deleted, or a new edge is inserted. A default observer class-template with empty implementations contained in the package can serve as source class to derive models that execute special code on such changes. An example application is a point location that relies on auxiliary data (*landmarks*) which should be kept up-to-date upon structural changes of the arrangement it is connected to. The numbers of observes attached to an arrangement is not limited.

The package is also equipped with a number of free[19] `insert` functions, that allow to insert curves into a given empty or non-empty arrangement. Depending on the case a single

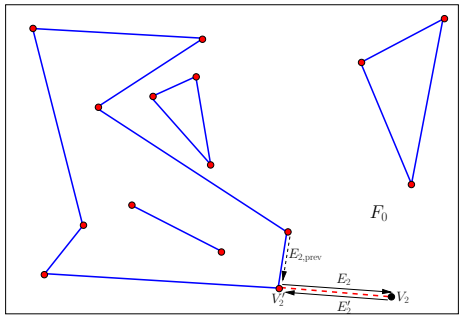
[18] One may wonder why the parameter is called `GeometryTraits_2` and not `ArrangementTraits_2`. The reason is that *ArrangementTraits_2* is the most refined concept, but one can also use `Arrangement_2` with weaker concepts (e. g., the input curves do not intersect). Thus, the more generic name. We usually refer to the most refined version.

[19] Floating in namespace `CGAL::` without coupling to a class.

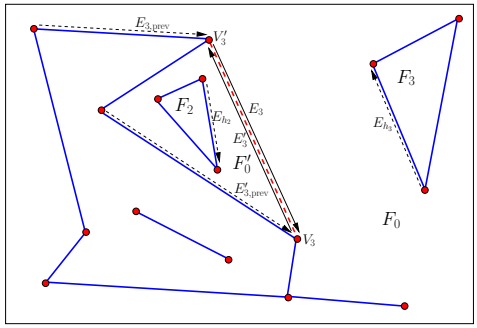
Figure 2.9. Basic insertions into a planar arrangement of line segments



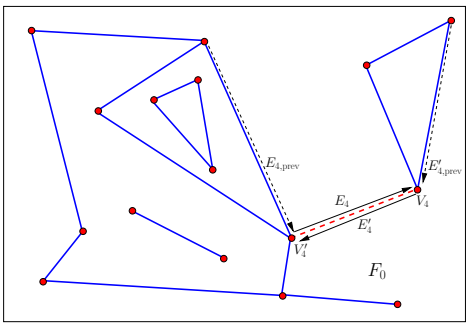
(a) Insert in face interior: The arrangement creates two new vertices V_1 and V_1' and connects them with the halfedge-twins E_1 and E_1'. Here: The cycle (E_1, E_1') forms a new inner CCB of F_1.



(b) Insert from vertex: V_2' exists, while V_2 must be created, as well as the halfedges E_2 and E_2'. Both extend in form of an *antenna* the CCB to which the given $E_{2,\text{prev}}$ belongs. Here: E_2 and E_2' extend an inner CCB.



(c) Inserting from two vertices: A new pair of halfedges E_3 and E_3' close a new face F_0'. Holes and isolated vertices of the old F_0 (e. g., for E_{h_2} and E_{h_3}) must be checked whether to move to F_0'. Here: H_2 moves to F_0'.



(d) Inserting from two vertices: The new pair of halfedges E_4 and E_4' connects two components, that is, it merges two CCBs. Here: two inner CCBs are merged into one.

curve is incrementally added or a group of curves is inserted with the sweep line approach. The free `overlay` function efficiently *overlays* two given arrangements. If a group of curves is known not to intersect in their interior, special `insert_non_intersecting_curves` functions are also available, that have impact on the efficiency and the required operations: These methods are faster and only demand a reduced set of geometric operations. In particular, the construction of intersections is avoided, which usually results in geometric objects with an increased complexity. For example, bit-lengths for intersections of curves are usually larger than for the originating objects.

The free insertion functions are internally implemented in terms of the *visitor design pattern*. That is, the aggregated construction is based on the

```
Sweep_line_2< GeometryTraits_2, Visitor,...>
```

class-template that implements a generic sweep line algorithm as described in §2.4.2. In particular it can deal with any degeneracy that is possible, for example, vertical curves, covertical events, more than two curves intersecting in a point, intersection at endpoints, or overlapping curves. The `GeometryTraits_2` parameter again refers to the geometry model that should be used, while the given instance of type `Visitor` receives notifications about the status of sweep line algorithm and can act with respect to these changes. With this strategy the actual sweep line code is centralized, reusable, and easy to maintain. The implementation of a sweep-based algorithm boils down to write the visitor that constructs the desired output from the notifications. The package provides a number of visitor classes for various purposes:

- construct the intersections of curves
- construct the arrangement as DCEL induced by curves
- insert a set of curves into an existing arrangement
- overlay two arrangements
- perform batched point locations

Other sweep-based algorithms can be realized by writing own visitors.

In the same spirit, the incremental insertion is realized by a model of the *ZoneVisitor* concept that inserts the curve, while the generic zone computation, implemented by the class-template

```
Arrangement_zone_2< Arrangement, ZoneVisitor, ...>
```

is executed. There is another visitor for the zone algorithm that just report intersections along the zone. As before, writing own visitors allows to easily develop computations that are based on the zoning.

Functionality

We want to highlight three capabilities provided with CGAL's `Arrangement_2` package that are constantly applied through the main chapters of this thesis. It only represents a small subset of the full functionality provided by the package.

Decorating The `Arrangement_2` package provides several methods to attach additional data to the geometry. For example, input curves can be enhanced with a user-specific type

(e. g., a color) that is even preserved when applying computations (e. g., a sweep line) on them. The most sophisticated method to attach data consists of the class-template

```
Arr_extended_dcel< GeometryTraits_2, VertexData, HalfedgeData,  FaceData,...>
```

that can be used when instantiating the `Arrangement_2` template. Each type of DCEL-records is then equipped with the corresponding type and the data of a given DCEL-record (be it a vertex, a halfedge, or a face) can be accessed by a `data()` member.

Overlay As already mentioned there exists a free function

```
CGAL::overlay(arr1, arr2, arr_ovl, ovl_traits)
```

that computes the overlay of `arr1` and `arr2` and stores the result in `arr_ovl`. Its correctness and efficiency is ensured by instantiating the sweep line implementation with the

```
Arr_overlay_sl_visitor< OverlayTraits >
```

that is instantiated with a model of the *OverlayTraits* concept. If no user-specific data is attached, the default `Arr_default_overlay_traits` suffices as argument for `ovl_traits`, otherwise the necessary merging of attached data must be implemented by a case-specific model. Such a class determines, for example, how to combine the data attached when overlaying a face of `arr1` with a face of `arr2`, and all other possible combinations. We refer to the manual [WFZH07a] for further details, and mention only the simple example, where a `bool` is attached to each DCEL-record, and the model of the *OverlayTraits* concept implements a boolean operation (like `and`) on the attached boolean values.

Point location Having an arrangement instance at hand, a very common query consist in the question where a query point q is located, that is, to identify the DCEL-record to which q belongs. For random points, the found object is usually a face, while for degenerated queries the point can be located on an edge or even coincide with a vertex. Again, the `Arrangement_2` package relies on the generic programming paradigm capabilities to implement various kinds of point-location strategies. In particular, a developer is invited to write its own method, while a basic set of strategies comes out-of-the-box:

- The *naive* strategy exhaustively scans each DCEL-record until it successes.
- The *simple* approach uses some geometric filtering.
- A more sophisticated method walks along a *vertical ray* emanating from q until it hits an edge or vertex, or extends to infinity. Depending on this the corresponding DCEL can be obtained.
- There is also a point location that relies on a set of *landmarks* stored for the arrangement. The positions of landmarks are known. The query consists in an efficient detection of the nearest landmark to q and the traversal of the line that connects q with this landmark. This method requires auxiliary data.
- Auxiliary data is also required by the point location strategy that utilizes a partial vertical decomposition of the arrangement.

See CGAL's manual pages for full details [WFZH07a].

Remember that the point-location strategy might not be a game of its own, as for example, the incremental insertion of a curve using Algorithm 2.14 has to locate the

curve's endpoint before starting the zone computation. This fact should be adhered, when choosing the point-location strategy.

We skip further details on the impressive functionality of CGAL's `Arrangement_2` package, and refer to [WFZH07b] and [WFZH07a] for further reading. In addition, CGAL's manual pages also cover details to use the package for envelopes of curves [Wei07a], Minkowski sums in two dimensions [Wei07b], or as basic support in regularized boolean set operations [FWZH07].

The *ArrangementTraits_2* concept

The basis interface between the `Arrangement_2` package and the geometric object is the `GeometryTraits_2` parameter, that fulfills the *ArrangementTraits_2* concept or one of the weaker versions: The concept is actually described hierarchically, as some algorithms and maintenance operations only require very basic types and operations on them, while others are expecting a larger set (of types or operations, or both). We omit to present the full distinction of layers that enables a fine adjustment of available traits model and the desired application. In fact, all models we know are implementing the full set of requirements. A model of CGAL's *ArrangementTraits_2* concept is expected to provide three main types:

`Curve_2` This type is used to store a general curve, howsoever it is represented. Its topology might be very complex, for example, it can have self-intersection, or comprises several components that even may be zero-dimensional. No further specific requirements are demanded from this type, except from the fact that it can be decomposed. We refer to `Make_x_monotone_2` for further details.

`X_monotone_curve_2` This type is used to represent a (weakly) x-monotone curve. All geometric algorithms of the `Arrangement_2` package are designed to rely on weakly x-monotone curves.

`Point_2` Objects of this type are used to represent (finite) ends of weakly x-monotone curves, and their (finite) intersections.

Any model of the *ArrangementTraits_2* concept is also expected to provide geometric predicates and constructions as functors. For `Curve_2` only one construction is expected.

`Make_x_monotone_2` Decomposes a general `Curve_2` into a finite number of (weakly) x-monotone curves and (maybe) a finite number of isolated points. If the remaining operations require more conditions on the curves, this functor also has to take care to construct the sub-curves respecting these prerequisites.

All other operations involve only (weakly) x-monotone curves and points, and it is no surprise that the following predicates and constructions fit the tasks that we already identified as required for the sweep line algorithm. It turns out that the mentioned ones are the most important, while the list collecting the missing ones after the following detailed descriptions gives operations that are of more technical nature.

`Compare_x_2`, `Compare_xy_2` Compare the x-coordinates of two points or, respectively, compare the coordinates of two points lexicographically, that is first by x-coordinate, then by y-coordinate.

`Construct_min_vertex_2`, `Construct_max_vertex_2` Extracts the lexicographical smallest (largest, respectively) endpoint of a weakly x-monotone curve.

`Compare_y_at_x_2` Determines the relative vertical alignment of a point with respect to a weakly x-monotone curve.

`Compare_y_at_x_right_2`, (`Compare_y_at_x_left_2`) Determines the relative vertical alignment of two weakly x-monotone curve, immediately to the right of one of their intersections.

Remark. `Compare_y_at_x_left_2` is only expected when the tag `Has_left_category` has been set to `CGAL::Tag_true`, otherwise its expected outcome can also be deduced from converting the problem into a "right"-case. We omit the technical details. Anyhow, only some algorithms really require this predicate.

`Intersect_2` Computes the intersection of two weakly x-monotone curves, sorted in increasing lexicographic order. If a `Multiplicity` of intersection is known, it is attached to each intersection point. In case (parts of the) curves overlap, the overlapping portions are returned as (weakly) x-monotone curves as well.

The following self-explanatory operations that are expected for weakly x-monotone curves are of more technical nature: `Equal_2`, `Is_vertical_2`, `Split_2`, `Are_mergeable_2`, `Merge_2`. The exact signatures for each construction and predicate is listed in CGAL's manual [WFZH07a].

Remark (Asymmetry). The asymmetry of the expected functors (for example, there is no `Compare_yx_2`) is intended and results from the fact that we split curves into x-monotone pieces and also assume that we *sweep with a line* from *left to right*. Any model fulfilling is allowed to over-achieve the concept's demands by further functors.

Available models CGAL's `Arrangement_2` package already contains several models of the *ArrangementTraits_2* concept, among them classes for line segments (with different caching strategies), and one for polylines. Both require only exact rational arithmetic. There are also classes for non-linear curves which are computationally more complex and require algebraic numbers of higher degree. The simplest is the one that handles segments and circular arcs [WZ06]. Circles are special algebraic curves of degree 2. The `Arr_conic_traits_2` class handles arcs of arbitrary degree 2 curves [Wei02], so-called conics. A model for arbitrary algebraic curves of any degree is not part of the package. However, there are two specializations for any degree. The simpler one allows to compute and maintain arrangements defined by rational functions [FHK+, §1.4.2], that is, an arc is defined by an interval $I := [\ell, r]$ and by the graph of a function $y = f(x) = \frac{p(x)}{q(x)}$ over I, with $p, q \in \mathbb{Q}[x]$. The most sophisticated model contained in the package deals with Bézier curves of arbitrary degree [HW07]. The efficiency of the later implementation results from a consistent application of geometric filters, that is, most computations can be derived from the geometric properties of Bézier arcs, namely their bounding polygons. Only in a few (near-)degenerate cases, exact algebraic methods cannot be avoided.

There are also some "external" contributions of *ArrangementTraits_2* models, that is, they are not shipped with the `Arrangement_2` package. CGAL's `Circular_kernel_2` extends a linear kernel with circles and a basic set of predicates and constructions. It also

provides a model of the *ArrangementTraits_2* concept [PT07]. The kernel has been used to compute aggregated unions of circular polygons that occur in VLSI design [dCPT07]. Outside CGAL, Lazard et al. have developed a model that also realizes arcs of rational functions [LPT08]. It internally uses RS for real root solving of the occurring univariate polynomials.

The EXACUS-team also participated in the challenging task to provide models. In contrast to the previous classes, the project does not have specialized models for different curves, but maintains a generic implementation. The central idea is that all required operations can be expressed in terms of the analysis of single curves and pairs of them. This layer of abstraction has been implemented in EXACUS' SWEEPX library. Its name used to be *generic algebraic points and segments* (GAPS). As mentioned, the EXACUS libraries are moving into CGAL. Thus, we desist from discussing the original implementation, and refer to §2.4.4 where we present CGAL's new `Curved_kernel_via_analysis_2` package that emerged from GAPS and even improved it. We only mention, that this way it is possible to compute arrangements of conics [BEH+02], cubics [EKSW06] (theoretically improved by [CGV08]), projected silhouettes and intersections of quadrics [BHK+05], and algebraic curves of arbitrary degree [EK08a]. Caravantes and González-Vega filled the gap with arbitrary quartic curves [CGV07], however, an implementation is missing. Using a specialized algebraic kernel, it is also possible to compute arrangement of conics rotated by angles whose sin and cos are (nested) one-root numbers; see [BCW07]. In an internal version of CGAL, the same idea has already been applied to algebraic curves of arbitrary degree.

We also mention that the `Arrangement_2` package provides a set of wrapping traits models, that is, a given model can be enhanced with additional properties. An example is the `Arr_counting_traits_2` that counts how often each geometric operation has been called, for example, when inserting curves with a sweep into an empty arrangement. The outcome can help to improve an implementation. Another wrapper is the `Arr_tracing_traits_2` class, that prints the input and output for each traits operation during an execution. This is very helpful for debugging purposes.

Remark (Boundedness). We remember the fact that all presented algorithms are designed to work for curves γ with $I = [0, 1]$, that is, all curves are bounded. The `Arrangement_2` class-template of CGAL version 3.2 only allows to have *one* unbounded face, and, as carefully denoted, the types and operations expected from a model of the *ArrangementTraits_2* concept also expect finite ends of curves.

However, Chapter 4 describes how the package has been extended to remove such restrictions. We antedate that all presented models for curves in the plane have been adapted towards unboundedness, that is, their current version is already primed and outfitted with the extended set of operations that we discuss in §4.2.1.

We conclude this introduction on arrangements by presenting a generic model of the *ArrangementTraits_2* concept that relies on analyses provided by a model similar to the *AlgebraicKernelWithAnalysis_d_2* concept. It plays an important role throughout the thesis.

2.4.4. Curved_kernel_via_analysis_2

In this section we present CGAL's new `Curved_kernel_via_analysis_2` package that provides a generic kernel for curves than can be analyzed. The kernel is one of the main achievements in terms of community service that we present. Its history goes back to the *Generic Algebraic Points and Segments* (GAPS) module that used to be part of EXACUS's SWEEPX-library. That module has been initiated in [EKSW06] to support points and arcs of cubic curves. While this first version had some restriction with respect to the generic position assumption, we removed them, and completed the implementation for [BHK+05]. We skip further details on GAPS and present next what emerged from that code, namely the `Curved_kernel_via_analysis_2` class and its dependent classes. The current, improved, design and the implementation results from joint work of the author with Pavel Emeliyanenko. More details and the reference documentation can be found in [BE08].

The `Curved_kernel_via_analysis_2` package is a layer between curves that can be analyzed on one side and objects supported by such curves along with geometric predicates and constructions on the other side. We already mentioned analyses of curves and pairs of such in §2.3.3. The `Curved_kernel_via_analysis_2` package heavily relies on exactly such analyses. In contrast to the GAPS module, it does not assume curves to be algebraic. Thus, the main `Curved_kernel_via_analysis_2`-class is templated in a more generic parameter

```
Curved_kernel_via_analysis_2< CurveKernel_2 >
```

We omit to discuss the more generic *CurveKernel_2* concept in detail, as the differences to the *AlgebraicKernelWithAnalysis_2* are mainly names avoiding algebraic terminology. Thus, we can assume, for simplicity of presentation in this thesis, that we instantiate the `Curved_kernel_via_analysis_2` class-template with a bivariate algebraic kernel with analysis, for example `ACK_2`:

```
typedef Curved_kernel_via_analysis_2< ACK_2 > CKvA_2;
```

An important subtlety in this simplification step should be mentioned: We identify the `Xy_coordinate_2` type defined in the *CurveKernel_2* concept with the `Algebraic_real_2` defined in the `ACK_2`. This means, that we also assume a special internal representation and constructor for an `Algebraic_real_2`, that is, its internal representation relies on a curve-analysis; see Definition 2.45 that gives the details. This choice enables an integrated usage of the analyses, in both `ACK_2` and `CKvA_2`, and additional computational effort is avoided from the beginning. The strategy mainly supports the overall goal of the `Curved_kernel_via_analysis_2` to derive all geometric operations without the explicit knowledge of y-coordinates, as this can be a costly task.

Definition 2.45 (Implicit y-coordinate). Each point $p = (p_x, p_y)$ on a curve c, that can be analyzed, can be uniquely represented as a triple (p_x, c, a), where a denotes the index that identifies p among the sorted distinct intersections of c with the vertical line at p_x, where counting starts at 0.

Thus, the integrated handling of curve analyses is ensured by representing an instance of type `Xy_coordinate_2` (and thus, by assumption, an `Algebraic_real_2`) by such a triple (x, c, a). Of course, it is still possible to extract the exact y-coordinate. However, it is not

expected by the `Curved_kernel_via_analysis_2`. In any case, this choice has implications on how to compare two instances of type `Xy_coordinate_2` lexicographically.

Algorithm 2.15. Lexicographical comparison of two `Xy_coordinate_2`

INPUT: $xy_1 := (x_1, c_1, a_1)$; $xy_2 := (x_2, c_2, a_2)$
OUTPUT: Lexicographic order of xy_1 and xy_2

- If $x_1 \neq x_2$, return their order.
- Else, if $c_1 = c_2$, return the order of a_1 and a_2.
- Else, analyze pair of curves defined by c_1 and c_2, and compute their status line at $x_1 (= x_2)$. Locate a_1-th "arc" of c_1 as index i_1, and a_2-th "arc" of c_2 as index i_2 in sequence of merged curves along the status line at x_1. Return the order of i_1 and i_2.

An illustration of this algorithm is given in Figure 2.10.

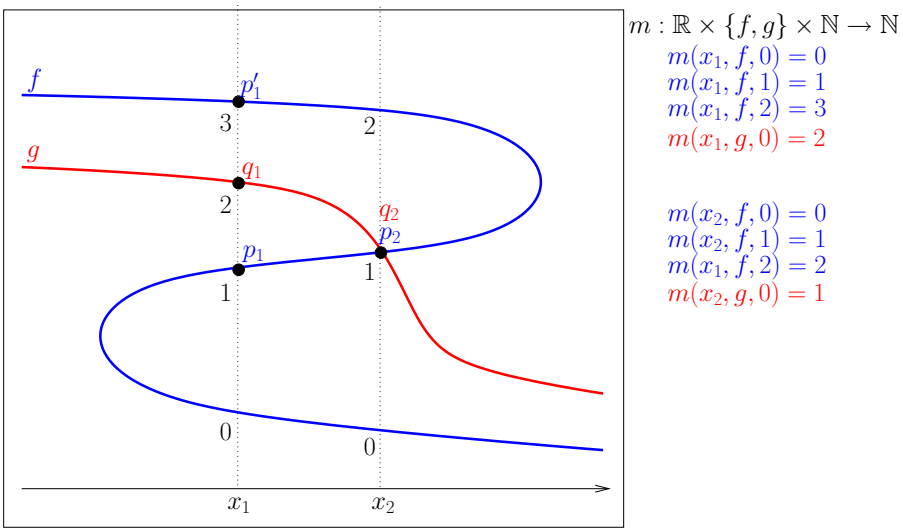


Figure 2.10. Compare-xy via analyses of curvess: Given $p_1 = (x_1, f, 1)$, $p_1' = (x_1, f, 2)$, $q_1 = (g, x_1, 0)$ and $p_2 = (f, x_2, 1)$, $q_2 = (g, x_2, 1)$. All points with $x = x_1$ are lexicographically smaller than points with $x = x_2$. Then, $p_1 <_{\text{lex}} p_1'$ as both lie on f and $a_{p_1} < a_{p_1'}$. It also holds that $p_1 <_{\text{lex}} q_1$, as $m(x_1, f, a_{p_1}) < m(x_1, g, a_{q_1})$ and $q_1 <_{\text{lex}} p_1'$, as $m(x_1, g, a_{q_1}) < m(x_1, f, a_{p_1'})$. Finally, $p_2 =_{\text{lex}} q_2$, as $m(x_2, f, a_{p_2}) = m(x_2, g, a_{q_2})$.

The techniques used in Algorithm 2.15 can be seen as blueprints for other geometric operations implemented in the `Curved_kernel_via_analysis_2`; see below.

Basic types

In fact, the combinatorial information obtained from curve analyses is a central source of knowledge within the `Curved_kernel_via_analysis_2`. While the `Curve_analysis_2` and `Curve_pair_analysis_2` types are given through instantiation, three new types to represent geometric objects are defined by the `Curved_kernel_via_analysis_2` class.

`Point_2` This is the simplest one among the three. A standard point is constructed from a triple (x, c, a). Internally it holds a pointer to an `Xy_coordinate_2` instance. In the algebraic case, the x-coordinate can be a real algebraic number of any degree.

Although not handled until Chapter 4, we already remark that there are special points to represent ends of non-bounded arcs. Such points, however, are not explicitly constructible by the user.

`Arc_2` Represents a one-dimensional connected and weakly x-monotone subset of a curve. An `Arc_2` *arc* is either vertical, or it has the property, that the arc number for all points in its interior is constant.

Internally, it stores besides minimal and maximal endpoint $p_{\min}$, $p_{\max}$, its supporting curve c, and three arc number $a_{\min}$, a, and $a_{\max}$. Note that the supporting curves of $p_{\min}$ and $p_{\max}$ do not have to match c, and similar their arc numbers do not have to match $a_{\min}$ and $a_{\max}$. However, $a_{\min}$, a, and $a_{\max}$ must be chosen such, that the represented arc is a connected subset of c.

`Poly_arc_2` This type is only for the user's convenience, as it allows to represent a non-x-monotone connected subset of a curve c by a chain of connected `Arc_2` instances. There are preconditions, that all these arcs must be supported by the same curve, and all arcs are either vertical or non-vertical. There is the plan to provide a one-dimensional object composed of arcs supported by different curves.

In order to simplify the subsequent discussion we assume that the considered supporting curves have a finite number of (self-)intersections. Of course, the implementation takes care of such special cases, and simplifies in an on-line fashion (i. e., interactively during execution of an operation) the internal representations of the `Point_2` and `Arc_2` respectively. Simplification means to choose curves that only have a finite number of (self-)intersections, and to adapt affected arc numbers, respectively.

Operations

In this part, we present the central operations of the kernel. Much more are implemented and currently documented in [BE08].

`Make_x_monotone_2` The main operator of this functor decomposes a given curve c with the help of a left-to-right traversal of c's analysis into a finite number of `Arc_2` instances and isolated points of type `Point_2`.

Another operator unchains the linking of a `Poly_arc_2`.

There are two trivial operators for `Arc_2` and `Point_2` that just return the given objects itself as it is already (weakly) x-monotone.

A final operator takes a `CGAL::Object` that is allowed to encapsulate any of the `Curved_kernel_via_analysis_2`'s geometric types. Depending on the type, one of the previous four operators is applied and the proper decomposition is returned.

`Compare_xy_2` For two instances of type `Xy_coordinate_2` stored for the two given points, the functor executes Algorithm 2.15 to compare them lexicographically.

`Compare_y_at_x_2` The functor compares the relative vertical alignment of a point p and an arc *arc*. As precondition the point must lie in the x-range of *arc*. The result is obtained from constructing a point p_{arc} on the arc at p's x-coordinate, and then to (lexicographically) compare p with p_{arc}. Note that we can simply skip the comparison of x-coordinates in this case.

`Compare_y_at_x_right_2` Given two arcs and one of their intersection points. If the supporting curves of the arcs are equal, we can just compare the two interior arc numbers. Otherwise, we compute a status line of the corresponding pair of curves slightly to the right of the intersection (e. g., at a representative and rational r within the open interval to the right of the intersection's x-coordinate), and compare the relative vertical alignment of the arcs in the spirit of the y-comparisons of points in Algorithm 2.15.

`Intersect_2` Given two arcs, compute all their zero- and one-dimensional intersections. Note that the supporting curves are not equal and have a finite number of intersections. We first compute the common x-range of the two arcs. Then, we traverse the analysis of the corresponding pair of supporting curves from the left end of the common range to the right end, detect in each status line of an event the intersections of the two curves. This information, suffices to construct the intersection points. An overlap is detected priorly, and requires a matching between the common supporting curve(s) and the two curves supporting the input arcs.

Each of these operations also has some subtleties, for example with respect to the handling of vertical arcs. We do not want to discuss the technical details in this overview.

The `Curved_kernel_via_analysis_2` as *ArrangementTraits_2* model

We aim to use an instantiated `Curved_kernel_via_analysis_2` as the `GeometryTraits_2` for CGAL's `Arrangement_2` package. Thus, it has to fulfill the *ArrangementTraits_2* concept. All necessary functors are already in place. It remains to define the required types. Remember that the *ArrangementTraits_2* concept expects three types. For the `Point_2` we do not have a choice, and as the `X_monotone_curve_2` only `Arc_2` is sufficient. Some flexibility is afforded with respect to the input type `Curve_2`. As the `Make_x_monotone_2` functor can deal with all internal types, it is the user's choice to typedef `Curve_2` either to `Curve_analysis_2`, `Poly_arc_2`, `Arc_2`, `Point_2` or even `CGAL::Object` that is most flexible as it can encapsulate each of the former types. We recommend to choose among `Curve_analysis_2`, `Poly_arc_2`, or `CGAL::Object`, as for the others no `Make_x_monotone_2` is required.

To summarize, we obtain a valid model of CGAL's *ArrangementTraits_2* concept for algebraic curves to be used as `GeometryTraits_2` in the `Arrangement_2` package by instantiating the `Curved_kernel_via_analysis_2` with a bivariate algebraic kernel (e. g., direct or wrapping version of `Algebraic_curve_kernel_2`). We henceforth use the shorter term `CKvA_2` when referring to such an instantiated instance.

In Chapter 3 we use `Algebraic_curve_kernel_2` wrapping QUADRIX's `P_curve_pair_2`, while in Chapter 4 and 5 we mainly rely on the self-contained `Algebraic_curve_kernel_2` in combination with the `Curved_kernel_via_analysis_2`. Typically, we only make use of the `Curved_kernel_via_analysis_2` as a mediating layer. However, in §4.3 we show how it

must be extended to support unbounded curves, while in §4.6 we even modify it noticeable in order to compute arrangements on parametric surfaces. These modifications are possible due to the chosen software design.

Software design The design of the `Curved_kernel_via_analysis_2` is held flexible. An intelligent combination of derivation and template meta programming allows to replace the two basic types `Point_2` and `Arc_2`. This way, the original class can be substituted by derived versions that are enhanced with additional functionality, such as a construction history. But not only the basic types can be exchanged, it is also possible to replace individual functors, for example, with a filtered version. Contained in the package we already provide a derived `Filtered_curved_kernel_via_analysis_2` whose functors are equipped with bounding box filtering in order to avoid analyses of pairs of curves; see [Ker08]. The current version is preliminary, that is, further improvements should be accomplishable.

Other derivations replace point and arc classes and some functors. Examples are the `Quadrical_kernel_via_analysis_2` for curves on a quadric (see §4.6.1) and the new `Arr_surfaces_intersecting_dupin_cyclide_traits_2` class that enables curves on a ring Dupin cyclide (see §4.6.2).

The kernel is also equipped with a robust visualization by Pavel Emeliyanenko for points and arcs following the ideas of [Eme07]. The package and its visualization can also be experienced in the web when computing arrangements of algebraic curves of arbitrary degree in an interactive demo; see [7] and [EK08c]. We also rely on the planar visualization when drawing an arrangement induced on a ring Dupin cyclide.

3 Lower Envelopes of Quadrics

Our journey between the three- and two-dimensional world starts with an important structure in computational geometry — lower envelopes. We present the computation of envelopes of a set of quadratic algebraic surfaces defined in $\mathbb{R}^3$ using CGAL's `Envelope_3` package. This package provides a generic and robust implementation of a divide-and-conquer algorithm. In this chapter, we concentrate on the algebraic and combinatorial tasks that occur for quadratic surfaces and their implementation. As the package follows the generic programming paradigm, we have to provide a quadric-specific model of a certain concept. Both, the package and the model are exact and robust, thus the obtained implementation follows the exact geometric computing paradigm. As we see at the end of this chapter, the efficiency depends on three criteria.

Parts of this chapter also appear in [Mey06a], as we describe a joint work with Michal Meyerovitch from Tel-Aviv University, Tel-Aviv, Israel. A short version of our results has been presented 2007 [BM07].

3.1. Envelopes

Lower envelopes are fundamental structures in computational geometry, which have many applications like computing general Voronoi diagrams, or performing hidden surface removal. Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ be a set of n (hyper)surface patches in $\mathbb{R}^d$. We denote with $x_1, \ldots, x_d$ the axes of $\mathbb{R}^d$, and assume (for now) that each S_i is monotone in $(x_1, \ldots, x_{d-1})$, namely every line parallel to the x_d-axis intersects S_i in at most one real point (without counting multiple intersections). If we now consider each patch S_i as a partially defined $(d-1)$-variate function $\mathbb{R}^{d-1} \to \mathbb{R}$, with $x_d = S_i(x_1, \ldots, x_{d-1})$, we can define the lower envelope.

Definition 3.1 (Envelope). The *lower envelope* $\mathcal{E}_\mathcal{S}$ of $\mathcal{S}$ is the point-wise minimum of these functions: $\mathcal{E}_\mathcal{S}(x_1, \ldots, x_{d-1}) := \min S_i(x_1, \ldots, x_{d-1})$, where the minimum is taken over all functions defined at $(x_1, \ldots, x_{d-1})$.

Instead of saying that a function S_i is not defined at some point $(\overline{x}_1, \ldots, \overline{x}_{d-1})$, we can also assume that $S_i(\overline{x}_1, \ldots, \overline{x}_{d-1}) = \infty$.

Definition 3.2 (Minimization Diagram). The *minimization diagram* $\mathcal{M}_\mathcal{S}$ of $\mathcal{S}$ is the subdivision of $\mathbb{R}^{d-1}$ into maximal connected cells such that $\mathcal{E}_\mathcal{S}$ is attained by a fixed (possibly empty) subset of functions over the interior of each cell.

Similarly, the *upper envelope* is defined as the point-wise maximum of the functions S_i which leads to their *maximization diagram*. However, until the end of the chapter we refer for the sake of simplicity to lower envelopes only.

The complexity of an envelope is defined by the complexity of its minimization diagram. Several analyses exists [HS94], [Sha94], [SA95]. Constructing an envelope for a set of (hyper)surfaces is also well-studied. Observe that the minimization diagram of algebraic (hyper)surfaces can be easily extracted from the proper cylindrical algebraic decomposition [Col75] (see also §2.1.6). The cad only needs to be clustered with respect to the minimization. However, the construction of a cad computes much more than needed, in particular, it always adheres hidden features. Hidden means that it considers boundaries or intersections of surfaces that finally do not show up in the minimization diagram. Several more efficient algorithms have been developed for low-dimensional envelopes, especially for $d = 3$. There exist output-sensitive algorithms for special cases [dBHO$^+$94], [KOS92], [Mul89]. A randomized incremental algorithm is due to Boissonnat and Dobrindt [BD96]. It runs in time $O(n^{2+\varepsilon})$, with $\varepsilon > 0$. The same time is needed by the divide-and-conquer approach presented by Agarwal et al. [ASS96].

Meyerovitch presented the generic and exact implementation of a divide-and-conquer algorithm for the three-dimensional case that decouples the combinatorial part from the geometric predicates using the generic programming paradigm [Mey06b]. The implementation is contained in CGAL's `Envelope_3` package, that has been released with CGAL version 3.3. It is based on and strongly coupled with CGAL's `Arrangement_2` package, which is a well-taken choice, since the problem actually is two-and-a-half-dimensional: The input $\mathcal{S}$ consists of objects in $\mathbb{R}^3$, while their minimization diagram is represented by an augmented planar arrangement in $\mathbb{R}^2$, that is, each cell of the arrangement (vertex, edge, and face) is labeled with the set of surfaces that attain the minimum over the cell. We typically distinguish between an empty set, a singleton, or more than one surface. Algorithm 3.1 describes how the labels are assigned using a divide-and-conquer approach.

Remarks (on Algorithm 3.1).

- We observe that its output is with respect to the xy-monotone pieces $g_1, \ldots, g_k$ of the S_i. This actually poses no real problem, as each g_j can store from which S_i it originates. In §3.3 we see an implicit storage strategy for quadrics.
- The splitting into $\mathcal{G}_1$ and $\mathcal{G}_2$ is not specified. However, in practice, a randomized partition obtains the best results. This has also been shown in theory by an analysis of the expected running time [HSS08].
- The descriptions of the algorithm contained in [Mey06a] and [Mey06b] cover more details. In particular, they discuss subtleties that we skipped for the sake of simplicity, they explain how to use CGAL's `Arrangement_2` package for the actual implementation, and they also presents how to propagate continuity and discontinuity information of the surfaces in order to significantly reduce the amount of geometric constructions and comparisons by combinatorial deductions. Such operations

Algorithm 3.1. Lower envelope with divide-and-conquer

INPUT: Set of surfaces $\mathcal{S} = \{S_1, \ldots, S_n\}$

OUTPUT: Minimization diagram $\mathcal{M}_{\mathcal{S}}$ representing the lower envelope $\mathcal{E}_{\mathcal{S}}$ of $\mathcal{S}$

- Extract (weakly) xy-monotone pieces of each S_i (each line parallel to the z-axis intersects such a piece at most once, or S_i is completely vertical). Let $\mathcal{G}$ be the set collecting them.
- If $\mathcal{G} = \{g\}$, compute $\mathcal{M}_{\mathcal{G}}$. This is done by first projecting the boundary of g into the xy-plane which induces faces. For each face it is decided whether it represents a projection of g. There can be more than one such *active* face. In case of g being vertical, no face is active. The decision is lead by a flag attached to each x-monotone projected curve of the boundary indicating whether the projection of g is *above*, *below*, or *none of them*; for an exact specification of these terms see Definition 3.4.
- If $|\mathcal{G}| > 1$, we split $\mathcal{G}$ into two non-empty sets $\mathcal{G}_1$ and $\mathcal{G}_2$ (of roughly the same size), recursively construct $\mathcal{M}_{\mathcal{G}_1}$ and $\mathcal{M}_{\mathcal{G}_2}$, and finally merge them into $M_{\mathcal{G}}$ with the following steps (simplified):
  1. Overlay the planar arrangements representing $\mathcal{M}_{\mathcal{G}_1}$ and $\mathcal{M}_{\mathcal{G}_2}$ resulting in O. Store for each cell Γ of O two pointers to Γ's originating cells $\Gamma_1 \in \mathcal{M}_{\mathcal{G}_1}$ and $\Gamma_2 \in \mathcal{M}_{\mathcal{G}_2}$.
  2. Update the labeled set $\ell_c \subseteq \mathcal{G}$ for each cell Γ of O: Let $\ell_1 \subseteq \mathcal{G}_1$ and $\ell_2 \subseteq \mathcal{G}_2$ be the labeled set of surfaces attached to Γ_1 and Γ_2. We skip the trivial cases, where at least one $\ell_1 = \emptyset$ or $\ell_2 = \emptyset$ holds. In the remaining non-trivial case the envelope over Γ is the envelope of $\ell_1 \cup \ell_2$. Reduce the sets ℓ_1, ℓ_2 to representative singletons $\ell_1' = \{g_1\}$ and $\ell_2' = \{g_2\}$. Split Γ (if not a vertex) with respect to the projected intersection of g_1 and g_2. For each resulting cell $\Gamma_1', \ldots, \Gamma_k'$ (k can become large) determine whether either ℓ_1', ℓ_2', or $\ell_1' \cup \ell_2'$ forms its envelope. Flush with re-replacing the representatives ℓ_1', ℓ_2' with ℓ_1, ℓ_2 in the labels of each Γ_i'.
  3. Clean up by removing edges whose two incident faces carry the same labeling as the edge. Also delete vertices of degree 2 whose two incident edges carry the the same labeling as the vertex and that can be merged geometrically (i. e., the edges and the vertex originate from a single projected curve).

are usually expensive, especially when following the exact geometric computation paradigm.

The outline of Algorithm 3.1 already defines the tasks that must be provided in order to support a certain class of surfaces. In particular, we detect (a) the extraction of (weakly) xy-monotone pieces, (b) to construct the projected boundary (with side information) for a single xy-monotone surface, (c) to construct the projected intersection of two xy-monotone surfaces, (d) to overlay arrangements composed of such constructed curves, and finally, (e) to determine the relative z-order of xy-monotone surfaces over a cell of a planar arrangement.

CGAL's `Envelope_3` package implements the generic parts, as the maintenance of the planar arrangement, or the overlay using the sweep-line algorithm. However, in order to compute the lower envelope for a certain family of surfaces, the surface-specific geometric types and operations must be provided. As usual for generic programming, this is done in form of a traits class fulfilling a certain concept. The `Envelope_3` package already contains such traits classes for triangles, planes, and spheres. In §3.3 we present the details of the concept, and show how to implement a proper model for quadrics.

3.2. Quadrics

Definition 3.3 (Quadric). A *quadric* is a real algebraic surface for whose defining polynomial $f \in \mathbb{Z}[x, y, z]$ it holds $\deg_{\text{total}}(f) = 2$.

As collected in §1.2, basically three approaches to computationally study quadrics exist. Namely, (a) the sweep of a plane perpendicular to the x-axis, while keeping track of topological changes, (b) the parametric approach, where intersection curves are represented in the parameter space of the quadrics, and (c) the projection approach, which projects curves of interest onto the xy-plane, analyzes them, and lifts them back to the third dimension. We notice that especially the projection method turns out to be a fundamental basis when computing envelopes. Let us briefly review the results of [BHK+05], that is basically motivated by the cylindrical algebraic decomposition method, see §2.1.6.

Let $\mathcal{Q} := \{q_0, \ldots, q_n\}$ be a set of n quadrics, among which we select one *reference quadric*, w. l. o. g. q_0. Abusing notation we identify with q_i also the vanishing set of the polynomial, that is, the surface itself. By resultant computations and Proposition 2.8 the intersection curves are projected onto the xy-plane. The resulting real algebraic plane curves have degree at most 4 and are Zariski closed. We call them *projected intersection*. The *silhouette* of q_0, defined by the intersection of q_0 and $\frac{\partial q_0}{\partial z}$, partitions q_0 into a *lower* and an *upper* part. We also project the silhouette onto the xy-plane. The corresponding curve is also Zariski closed, has degree at most 2, and is called the *projected silhouette*. We can combine a proper model of the *AlgebraicKernelWithAnalysis_2* concept with CGAL's `Curved_kernel_via_analysis_2` (`CKvA_2`) to compute the induced planar arrangements of the projected curves as explained in §2.4.4. Two such models exist. One instantiates the `Algebraic_kernel_2` (in wrapping mode, see §2.3.3) with quadric-specific analyses of planar curves of degree 4. These analysis are taken from EXACUS's QUADRIX library, and presented in [BHK+05]. The other is CGAL's new `Algebraic_curve_kernel_2` that comprises the analysis of algebraic curves of arbitrary degree,[20] that also suffices for our

[20] Formerly known as EXACUS' ALCIX library.

purpose. The ingredients are published in [EKW07] and [EK08a]. These days, we prefer the second approach, as its analyses keep shearing internally, while the analyses of the quadric-specific analyses have preconditions on the choice of the three-dimensional coordinate system.

For the plane sweep, curves get decomposed into maximal arcs with constant arc number in their interior; see §2.1.4. However, in the projection of the three-dimensional curves onto the xy-plane the spatial information is lost. In order to recover it [BHK+05] uses a stronger decomposition of projected intersection curves, such that each (maximal) sub-curve can be uniquely assigned to the lower or upper part of q_0. As before, the projection of q_0's intersection with some q_i is split at its critical points, but also at its intersection points with the projected silhouette of q_0; see, in particular, Figure 3.1 (c).

Note that this decomposition is conservative in the sense that the curve may be split at projected points of $q_0 \cap q_i$ where the spatial counterpart only touches the silhouette of q_0, but does not cross it.

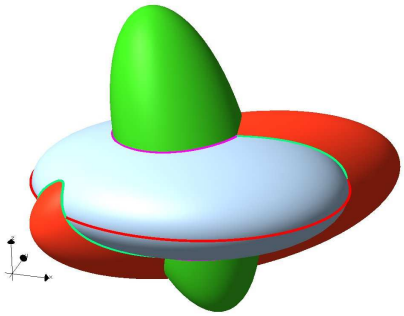
In the next step, each such sub-curve (and each existing isolated point) is checked whether it belongs to the lower part of q_0 or the upper part of q_0 (or even both, which is also possible) by finding the common intersection(s) of q_0 and q_i with a z-vertical line. In the generic case, the flip of intersections along two related lines with rational x- and y-coordinate is detected to decide whether a curves lies on the lower or upper part of q_0. Figure 3.2 illustrates this case. In the other cases, we have to directly compare z-coordinates of quadrics' intersections with a vertical line. We discuss these intersections below. For further details on the decomposition and the assignment we refer to [BHK+05] and [Ber04].

We next concentrate on the intersections of a quadric with a vertical line ℓ_p at some point p, which is important for the previous assignment. It is essential to compute the relative z-order of two quadrics, expected by the concept we have to model to compute a lower envelope of quadrics; see §3.3.

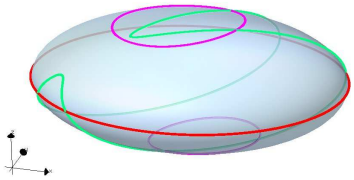
Let q_i be a quadric and consider a point $p = (p_x, p_y) \in \mathbb{R}^2$ with $\mathcal{R}_i(p) := \{z \in \mathbb{R} \mid 0 = q_i(p_x, p_y, z) \in \mathbb{R}[z]\}$. As $\deg_z(q_i) \leq 2$, it holds that $|\mathcal{R}_i(p)| \leq 2$. That is, if any is existing, q_i has either one or two intersections with ℓ_p and $\mathcal{R}_i(p)$ exactly defines their z-coordinates; see also Lemma 5.64. Let us have a closer look at the algebraic degrees of $\mathcal{R}_i(p)$'s elements.

- If p is a rational point, then $r \in \mathcal{R}_i(p)$ is an algebraic number of degree at most 2. Such a number can be represented in the form $r = a + b\sqrt{c}$, with $a, b, c \in \mathbb{Q}$, also referred to as a *one-root number*. CGAL's number type `Sqrt_extension` is able to represent such one-root numbers, allows to compare them, and provides arithmetic operators on them.
- Next, think of p lying on a projected silhouette of a quadric, with p_x being rational. Then, by $\deg_y(\mathrm{Res}_z(q_i, \frac{\partial q_i}{\partial z})) \leq 2$, p_y is not worse than a one-root number. We assume the worst, and thus conclude, that although $r \in \mathcal{R}_i(p)$ having algebraic degree 4, it can be represented by a *nested one-root number of depth* 1: We can write $r = a' + b'\sqrt{c'}$ where a, b, c are simple one-root numbers itself. CGAL's `Sqrt_extension` type allows such a nesting.
- Let now p be a singular point of a projected intersection curve of two quadrics. As shown in [Wol02] (and used in [BHK+05]), p's x- and y-coordinates can be represented as nested-one-root numbers of depth 1. Applying the previous idea again, $r \in \mathcal{R}_i(p)$ is representable as nested one-root number of depth 2. Alternatively, we can switch to numbers types representing algebraic expressions involving the $\diamond$-

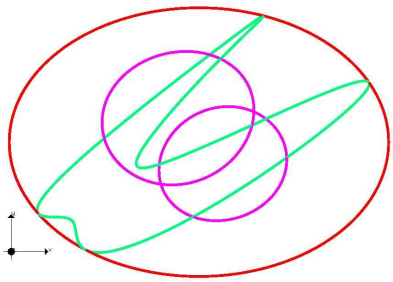
Figure 3.1. Developing the two arrangements on a reference quadric



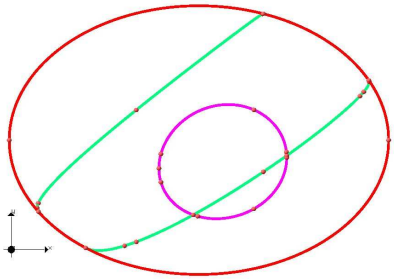
(a) Red and green quadric are intersecting the gray reference quadric q_0



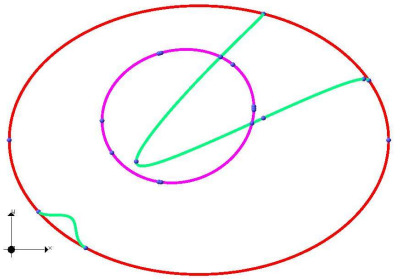
(b) The same situation on q_0



(c) The projection of the reference's silhouette and the two intersection curves onto the xy-plane. The projected intersection curves must be split and assigned to the lower and upper part of q_0.



(d) Arrangement on lower part of q_0



(e) Arrangement on upper part of q_0

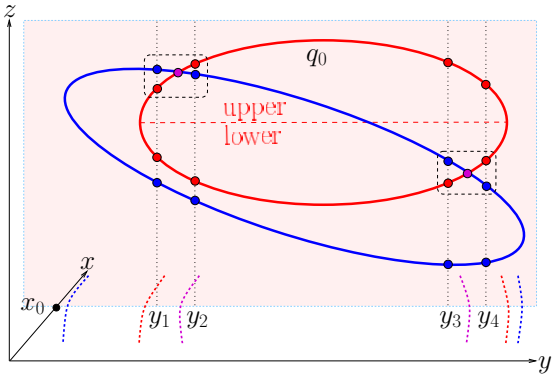


Figure 3.2. Lifting the intersections of the blue quadric with the red reference quadric q_0 to q_0's lower and upper part. In this generic case, it suffices to locate the flip (dashed rectangles) along pairs of z-axis parallel lines with rational x_0 and rational y_i. The picture takes place in the plane $x = x_0$. In degenerate cases, comparisons of one-root numbers give the answer.

operator. Examples are `leda::real` or `CORE::Expr` as stated in §2.3.1.

- Such algebraic expressions constitute the default representation for z-coordinates of $\ell_p \cap q_i$, for all other p. In particular, if p is an intersection of a projected silhouette and a projected intersection. Its x-coordinate has algebraic degree up to 8, which implies for its y-coordinate a degree of up to 16. Thus, $r \in \mathcal{R}_i(p)$ already has degree 32.

Of course, it is possible that algebraic expressions could also be used for all cases replacing all (nested) one-root numbers. However, detecting the equality of two such numbers r_1 and r_2 is more costly for algebraic expression, as $|r_2 - r_1|$ must be approximated below the separation bound to derive a certified answer. On the other hand, checking $r_2 - r_1 = 0$ using (nested) one-root numbers reduces to repeated squaring of the expression $r_2 - r_1$ until no square-root remains. This is usually the cheaper approach; see [Meh01].

Remark. In Chapters 4 and 5 quadrics also play a fundamental role. Observe that the intersection curves $q_0 \cap q_i, 1 \leq i \leq n$ actually induce a two-dimensional arrangement on the surface of q_0. The software presented in [BHK+05] is only able to compute two projected arrangements, that is, one for the lower part of q_0 and one for its upper part. Their connections are missing. Chapter 4 describes a framework that can be used to directly compute a sole two-dimensional arrangement for an elliptic q_0. In Chapter 5 we redesign the analysis of surfaces. The explicit representation for z-coordinates is replaced by an approximated version relying on the output of the bitstream Descartes method. We incorporate the idea for quadrics again, but also generalize to algebraic surfaces of arbitrary degree.

3.3. *EnvelopeTraits_3* concept and the model for quadrics

CGAL's `Envelope_3` package implements the generic programming paradigm, that is, in order to compute lower envelopes for a certain family of surfaces, the algorithm template must be instantiated with a *traits* class (see §2.2.1) that encapsulates basic geometric objects and operations on them. The requirements are also referred to as the concept that must

be fulfilled. The `Envelope_3` package expects a class that implements the *EnvelopeTraits_3* concept. In this section, we present the details of the concept and how we provide an implementation for quadrics. For the reason of readability, we simplify syntactical issues. The interested reader is encouraged to read the reference documentation in [MWZ07].

As the computation of lower envelopes is based on two-dimensional arrangements and also employs their overlays, the *EnvelopeTraits_3* concept is a direct refinement of CGAL's *ArrangementTraits_2* concept. Thus, we automatically inherit types for planar points (`Point_2`), planar curves (`X_monotone_curve_2`) and basic operations on them; see §2.4.3.

For quadrics: Thus, we derive the new model from the `CKvA_2` that is instantiated with one of the two possible algebraic kernels as written in §3.2.

The concept also expects spatial types and operations related to them. Two types are expected, namely `Surface_3` and `Xy_monotone_surface_3`.

For quadrics: We map both types to QUADRIX's `Quadric_3` class. This may be surprising at first, since a quadric, in general, is not xy-monotone. However, it is only an implementation detail to simplify matters. All subsequent operations that are expected to work on an xy-monotone surface g consider only the lower part of the appropriate quadric. If $f \in \mathbb{Z}[x, y, z]$ defines a quadric, its lower part is separated from its upper part by its intersection with the plane defined by $\frac{\partial f}{\partial z}$, the silhouette.

Due to this choice of types, the first expected operation[21] is simple:

- *Extract xy-monotone surfaces*

  Task: The function object `Make_xy_monotone_3` is expected to decompose a given surface S into its xy-monotone subsurfaces.

  For quadrics: As both basic surface types use the same representation, we simply return the given quadric itself as the sole output object.

For the other spatial functors, we first introduce some notation on planar curves.

Definition 3.4 (Points below and above a curve). Let c be a planar non-vertical x-monotone curve.

- We say that a point $p = (p_x, p_y) \notin c$ is *below* c if it lies in the x-range of c and if $p_y < p'_y$, where $p' = (p'_x, p'_y) \in c$ with $p'_x = p_x$. The analog case of *above* c is met if $p_y > p'_y$.
- The vertical half-open line segment defined by $C_{p,c} := \{(x, y) \in \mathbb{R}^2 \mid x = p_x \wedge p'_y < y \leq p_y\}$ is called the *critical segment between p and c*. The critical segment for a point above c is defined analogously.
- The set of all points p below c define a half-stripe called *area below c*, while the set of all points p above c define its counterpart, called the *area above c*.

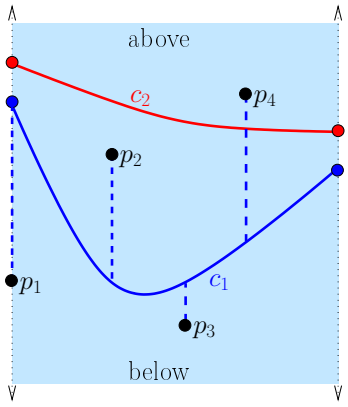
The notation of *below* and *above* is even used for a vertical c; but with exchanged coordinates. Figure 3.3 illustrates this definition.

Remark. Mind that we carefully distinguish notation here. The terms *below* and *above* classify planar points related to a projected *planar* curve and a critical segment is located in the xy-plane as well; see Definition 3.4. In contrast, *over* deals with intersections of a surface (or two surfaces) with a line parallel to the z-axis going through a planar point p

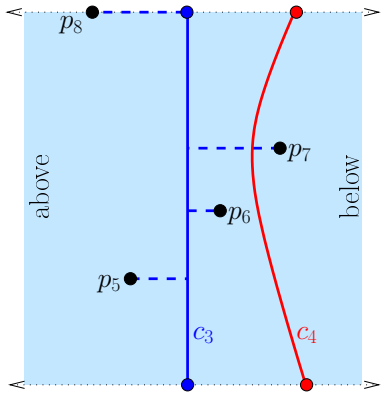
[21] Each operation is interfaced as function object (also known as functor).

(or a representative point p_c on a given projected curve c). In the latter case we mainly compute a set $\mathcal{R}_i(p)$ or compare entries of sets $\mathcal{R}_i(p)$ and $\mathcal{R}_j(p)$; see §3.2 for more details.

Figure 3.3. Points below and above curves



(a) Points p_2 and p_4 are above the non-vertical c_1; points p_1 and p_3 are below it. The critical segment for p_4 is intersected by the red curve c_2.



(b) Points p_5 and p_8 are above the vertical c_3; points p_6 and p_7 are below it. The critical segment for p_7 is intersected by the red curve c_4.

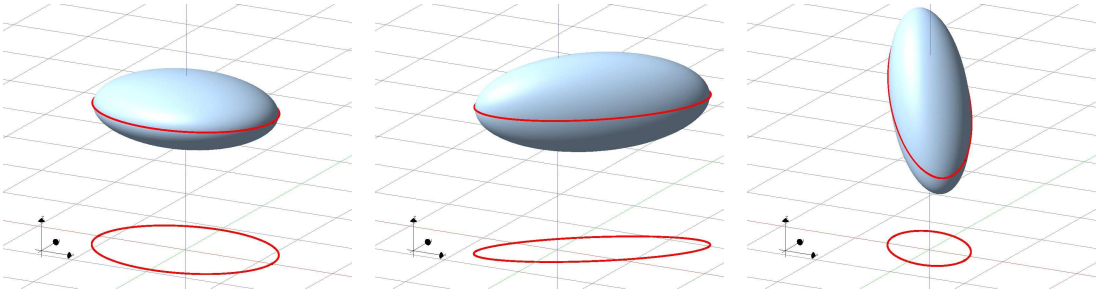
The next two expected operations perform the projection of boundaries or intersections into the plane of the minimization diagram. Their implementation for quadrics, benefits from prior work that we repeated in §3.2.

- *Construct projected boundary*

  Task: The function object `Construct_projected_boundary_2` computes for a given (xy-monotone) surface g all planar (weakly) x-monotone curves (and possibly isolated planar points) that form the projection of g's boundary into the xy-plane. Note that these objects are at most one-dimensional, that is, it is required to label induced open two-dimensional sets (faces), whether the surface exists *over* them (i. e., covertical to the planar face). For that purpose each reported (weakly) x-monotone curve c is enhanced with a flag whether the projection of g is (locally) below or above c. The flag can actually also encode the third, degenerate, case, namely that g is vertical over the corresponding curve c (i. e., g contains every line parallel to the z-axis, that run through points on c). The flag is used to properly tag all faces. Observe that the objective of this function object is to support the computation of $\mathcal{M}_{\{g\}}$, that is, the minimization diagram for a single surface as expected in Algorithm 3.1.

  For quadrics: The projected silhouette of a quadric q is easy decomposable into (weakly) x-monotone curves and isolated points, using `Make_x_monotone_2` supplied by `CKvA_2`. The assignment to which side of some c the non-vertical lower part of q is projected is decided in two steps: First, we choose a rational point $p = (p_x, p_y)$ below c, but close enough. This means that the critical segment between p and c is not intersected by any another projected boundary of q. Second, we compute the

Figure 3.4. Constructing the projected boundary for a quadric, three examples



(a) Quadric 1 **(b)** Quadric 2 **(c)** Quadric 3

cardinality $m := |\mathcal{R}(p)|$. This value gives the number of real roots of $q(p, z) \in \mathbb{R}[z]$, or more geometrically, the number of distinct real intersection of q with ℓ_p. If $m > 0$, the projected quadric is below c (by choice of point), otherwise it is above c. This simple implication (i. e., the else-case) is allowed as (1) quadrics that only show a single intersection over a non-boundary have no boundary at all and (2) if q is vertical this information is stored with q itself. Thus a single cardinality suffices. We simply save to check the cardinality over a second point above c to decide verticality.

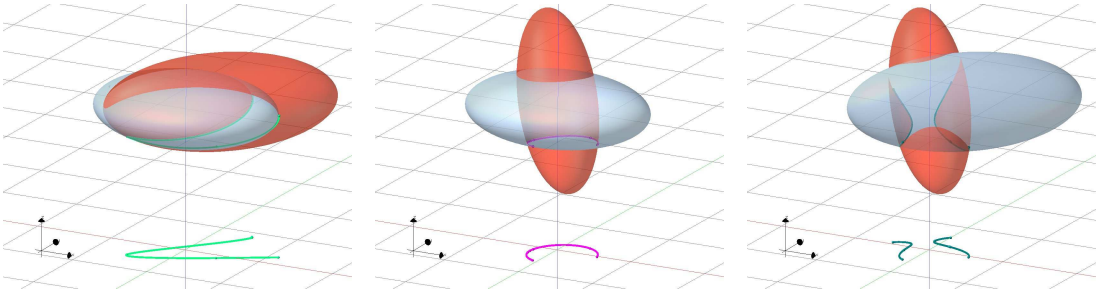
- *Construct projected intersection*

  Task: The function object `Construct_projected_intersection_2` computes the objects of the projected intersections of two xy-monotone surfaces g_1 and g_2. If such an object is an isolated point (`Point_2`) it is either the projected image of a degenerate (isolated) intersection, or the projection of a vertical intersection curve. Otherwise, an object can also be a one-dimensional (weakly) x-monotone curve c, which is equipped with an optional integral *multiplicity*. If this multiplicity is an odd value, we know that the two surfaces intersect transversely over c, that is, they change their relative z-order on either side of the spatial counterpart of c. An even multiplicity indicates that the surfaces maintain their relative z-order. The divide-and-conquer algorithm can derive the relative z-order of two surfaces on one side from their known relative z-order on the other side. This avoids explicit tests incorporating one of the remaining functors below, and thus, improves the overall performance of the algorithm. If the multiplicity is set to 0, additional comparisons are unavoidable.

  For quadrics: We mainly consider the projected intersection curve as presented in §3.2. Remember that we decomposed it with respect to its critical points and its intersections with the projected silhouette of a reference quadric. This time, we partition it with respect to the projected silhouettes of *both* given quadrics. This decomposition paves the way to assign (the interior of) sub-curves (and isolated points) uniquely to the lower part of both involved quadrics: For each point and each curve we check, using ray-shooting as in §3.2, to which part of the first quadric it belongs, and to which part of the second quadric it belongs. We finally return all sub-curves and points that have been assigned to the lower parts of both surfaces.

The remaining expected function objects compute the relative z-order of two xy-monotone surfaces g_1 and g_2 over projected cells of a planar arrangement. We distinguish

Figure 3.5. Constructing the projected intersection for pairs of quadrics



(a) Quadric 1+2 **(b)** Quadric 1+3 **(c)** Quadric 2+3

five cases, collected in three functors. For quadrics, all of them rely on computing and comparing the minimal intersections of $q_1(= g_1)$ and $q_2(= g_2)$ with ℓ_p at some suitable point $p = (p_x, p_y)$. It is easy to see that the relative z-order of the lower part of q_1 and the lower part of q_2 over p is given by the order of $r_1 := g_1(p) = \min \mathcal{R}_1(p)$ and $r_2 := g_2(p) = \min \mathcal{R}_2(p)$. That is, we mainly explain how to find a suitable point for each desired comparison. Depending on the representation (algebraic degree) of the point, an actual z-comparison is simply carried out by the comparisons of the corresponding number types: either (nested) one-root numbers or algebraic expressions. We refer to §3.2 where we discussed the different possibilities.

- *Compare z over xy*

  Task: The function object `Compare_z_at_xy_3` provides three operators. Each considers as input two given non-vertical xy-monotone surfaces g_1 and g_2 and a planar geometric object.

  1. The first determines the relative z-order of g_1, g_2 at a given planar point $p = (p_x, p_y)$. Both surfaces must be defined over p. The returned information is the comparison result of $g_1(p)$ and $g_2(p)$.
  2. The second determines the relative z-order of g_1, g_2 over the interior of a given (weakly) x-monotone curve c. It has the precondition that c is fully contained in the xy-definition range of both surfaces, and that c is not part of the projected intersection of g_1 and g_2. The functor is expected to return the comparison result of $g_1(p')$ and $g_2(p')$ for some point p' in the interior of c.
  3. The last operator is only required if unbounded surfaces occur. Actually, the surfaces must be defined over the entire xy-plane having no boundary and no intersection at all. A simple example consists two planes parallel to the xy-plane. The operator determines the relative z-order by (technically) choosing some planar point $p' \in \mathbb{R}^2$ and returning the comparison result of $g_1(p')$ and $g_2(p')$.

  For quadrics: We discuss the three operators in reversed order, as this reflects 1how complicated each is.

  For the third, we simply choose $p = (0, 0)$, compute one-root numbers r_1 and r_2 as defined, and compare them.

  For the second operator, we distinguish two cases, namely either

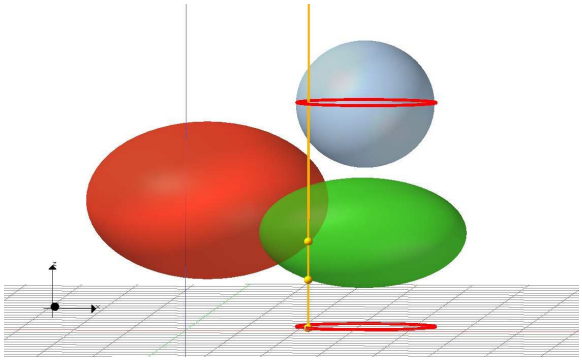


Figure 3.6. Compare relative z-order of (lower parts of) of two quadrics over a projected boundary curve (Example)

- c is part of a projected boundary of a quadric's lower part or
- c is part of a projected intersection of surfaces g_1' and g_2', with $\{g_1', g_2'\} \neq \{g_1, g_2\}$.

We see that the algebraic degrees of the corresponding p''s coordinates can be kept quite small in both cases, which starts with choosing a rational number for p_x' in c's x-range.[22] If c is part of a projected boundary, p_y' is a one-root number, and thus r_1 and r_2 can be computed and compared as nested one-root numbers of depth 1. Figure 3.6 shows an example for such a comparison. In the other case, we benefit from the fact that the projected intersection is not of the queried surfaces g_1 and g_2, which implies that there is a two-dimensional connected (maybe open) subarea below c whose points' critical segments are not intersected by the projected intersection of g_1 and g_2. Thus, a point from this subarea is a good candidate. However, we also need to ensure that the topology of g_1 and g_2 over such a point is identical to the topology of the surface over points of c. We choose a *rational* point p' in the subarea below c whose critical segment between p' and c is not intersected by any of the following curves: (1) The projected intersection of g_1 and g_2, (2) the curve that supports c (the projected intersection of some g_1' and g_2'), and (3) the projected boundaries of g_1 and of g_2 . The corresponding values r_1 and r_2 are one-root numbers.

It turns out that the comparison of the lower parts of two quadrics over a point p is the most expensive one as p's coordinates are often algebraic numbers of high degree. In addition, there is no guarantee to find a nice point p' (i. e., best with rational coordinates) nearby where q_1 and q_2 have the same order. In fact, most of the time such a nice point will just not exist, as the majority of usages of this method by the lower envelope algorithm only occur in degenerate situations. An implication is, that we are really forced to exactly compare the surfaces' relative z-order over a point with coordinates of higher algebraic degree. However, the list of possible cases is not arbitrary. In fact, the generic divide-and-conquer implementation exploits continuity and discontinuity information of the envelopes to carry a decision over between incident cells. Bringing this into consideration, the comparison of surfaces over a point can occur only in two special situations. They remain by checking all the possible cases where a point is created in the merge step, and keeping only those where the comparison method over a point is invoked: Either p is an isolated point

[22] If c is vertical, think of swapped coordinates for the whole procedure.

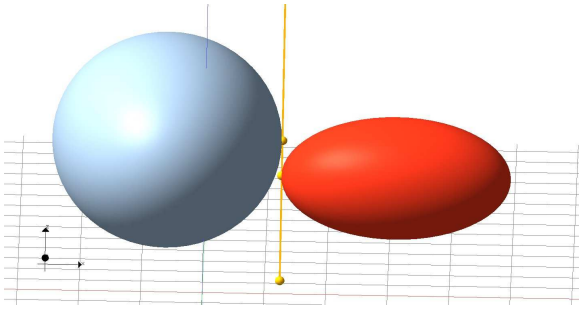


Figure 3.7. Compare relative z-order of (lower parts of) of two quadrics over a planar point (Example)

of a curve, or p lies on a projected boundary of an xy-monotone surface. In both cases, the algebraic complexity is not the highest possible.

In most cases, we can compute and compare r_1 and r_2 using (nested) one-root numbers. In particular, this holds for p being isolated, as an isolated point is singular and the coordinates of singular points of projected intersections can be represented as (nested) one-root numbers [BHK+05]; the singularity (if existing) of a projected boundary is, due to the degree, even rational; see Figure 3.7. If p lies on a projected boundary, we distinguish by the algebraic degree of p_x. If it is at most 2, we still can cope with nested one-root numbers. If if exceeds 2, we have no choice and switch to algebraic expressions to represent r_1 and r_2. This implies the costly usage of the $\diamond$-operator. However, the algebraic degree of p_x is bounded by 8. Note that this case forms the most expensive comparison in the algorithm, especially, if it eventually holds that $r_1 = r_2$.

Summarizing, it is possible in all cases to compute the relative z-order of the lower parts of two quadrics over a point or a curve. One can see, that due to the algebraic degree of quadrics, we can always use (nested) one-root numbers as long as the algebraic degree of the x-coordinate does not exceed two. Otherwise we have to switch to the expensive $\diamond$-operation. Note that, in general, the comparison over an *arbitrary* algebraic point p is possible using the same techniques, in particular when relying on algebraic expressions. However, this can be arbitrary costly (depending on the degrees) and it is not expected during the execution of the divide-and-conquer algorithm, because of the special care taken in designing the algorithm [Mey06b].

- *Compare z over area below (or above) curve c*

  Task: The function object `Compare_z_at_xy_below_3` computes the relative z-order of the two given xy-monotone surfaces g_1 and g_2 immediately over a point that is below one of their projected intersection curves c. It has the precondition, that both surfaces are defined below c, and their relative z-order is kept unchanged in some small enough neighborhood of points below c.

  For quadrics: To compute this information for quadrics, the strategy is similar as the comparison over a projected intersection curve. We choose a rational point below c whose critical segment is not intersected by any of the following curves: (1) the projected boundaries of q_1 and q_2 and (2) the projected intersection curve of

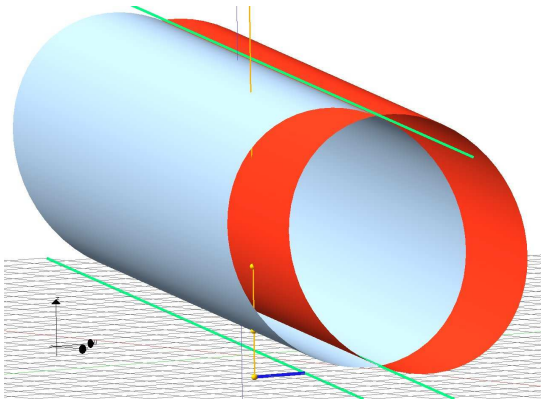


Figure 3.8. Compare relative z-order of (lower parts of) of two cylinders over a point in the area below their projected intersection (Example)

q_1 and q_2. This ensures that both surfaces are defined over p and, by a continuity argument, that the z-order over p is the desired order.

We skip the symmetric discussion of the also required above-version.

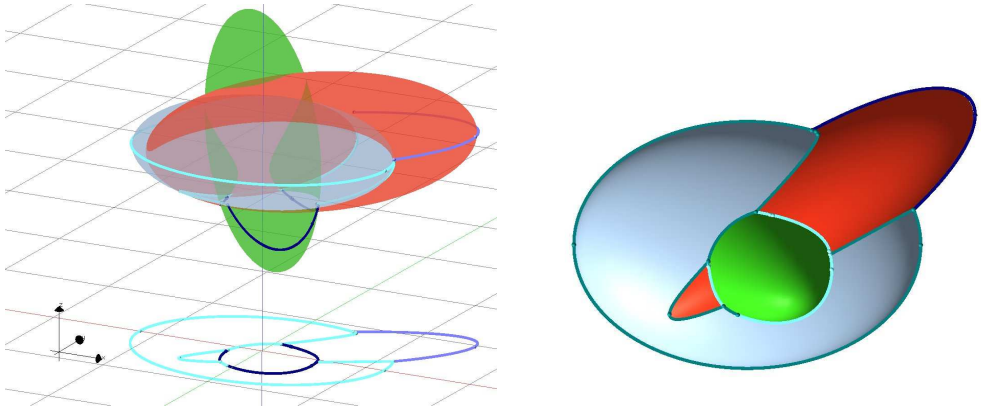
Remark (Unbounded surfaces). CGAL's `Arrangement_2` package in version 3.2 can only deal with curves having finite ends, which does not allow to store the projection of an unbounded xy-monotone surface. Its projection is simply not a compact set. In addition, if all curve-ends are finite, the arrangement only has to deal with a single unbounded face. This constitutes another problem, as it is insufficient to store the minimization diagram for a set of unbounded surfaces. There are simple examples (e. g., $\mathcal{S}$ contains a single infinite cylinder) where $\mathcal{M}_\mathcal{S}$ may comprises more than one unbounded face, and each such face stores an individual labeling. Both problems have been attacked by CGAL's `Arrangement_on_surface_2` package, which generalizes two-dimensional arrangements. The unbounded plane dealing with more than one unbounded face is the first surface that has been tackled. We present full details on the generic `Arrangement_on_surface_2` framework in Chapter 4.

3.4. Results

Using the model presented in §3.3, we can successfully construct lower envelopes (minimization diagrams) of quadrics with Algorithm 3.1 by calling `CGAL::lower_envelope` for a set of input surfaces. Figure 3.9 shows the final lower envelope of the surfaces introduced in Figure 3.1. The actual implementation of the traits class for quadrics is still in EXACUS' QUADRIX library. The whole library is going to move soon as a package of its own into CGAL. Thus, a future public release of CGAL will not only contain two main strategies to analyze quadrics and their intersections, but also comprise the computation of lower envelopes of quadrics. In addition, some variants are available as well. We present them at the end of this chapter; see §3.5.

The performance of our traits class for quadrics used in CGAL's divide-and-conquer algorithm to compute lower envelopes has also been checked experimentally. For increasing n we created five sets of random quadrics whose coefficients are ten-bit integers. We

Figure 3.9. Lower envelope of quadrics



(a) Final lower envelope **(b)** Look from $z = -\infty$

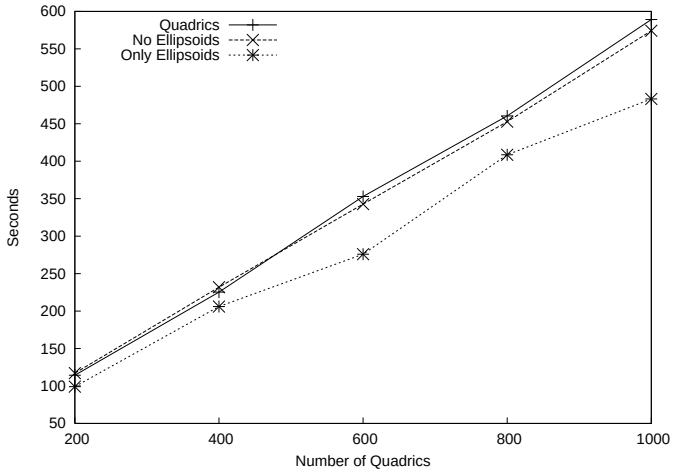


Figure 3.10. The running time required to compute the lower envelope of sets of quadrics as a function of the number of input quadrics.

n of	200	400	600	800	1000
quadrics	114.4	225.3	353.0	460.5	589.2
non-Ellipsoids	117.1	231.8	342.7	452.8	574.2
ellipsoids	99.1	206.0	275.9	408.6	483.2

Table 3.1. Averaged running times (in seconds) required for computing the lower envelope of instances of quadrics.

1000 of	Properties of $\mathcal{M}_S$			
	$\#S$	#V	#E	#F (unb.)
quadrics	8	15	22	8 (5)
non-ellipsoids	7	16	22	7 (5)
ellipsoids	67	249	324	77 (1)

Table 3.2. The number of attained surfaces and the size of minimization diagrams for a selected instance of 1000 quadrics of different kind.

used a version of CGAL's `Arrangement_2` package that is able to maintain several unbounded faces (see Chapter 4, where we discuss this extension in detail). This allowed us to consider bounded and unbounded quadrics. Actually, we distinguish between *ellipsoids*, *non-ellipsoidal quadrics*, and *mixed sets*. All experiments were executed on a 3 GHz Pentium IV machine with 2 MB of cache. For exact arithmetic we used LEDA's number types, and relied for the analyses of projected (boundary and intersection) curves on QUADRIX's specialized approach [BHK+05]. The resulting times in Figure 3.10 and Table 3.1 were averaged over several runs on the instances of same size. The obtained running times seem to (nearly) linear depend on the number of input surfaces. We emphasize that an exact result for 1000 arbitrary quadrics is computable in less than 10 minutes. As we can see from Table 3.1, computing lower envelopes of bounded quadrics (ellipsoids) is even remarkably faster. A reason is that ellipsoids are bounded and thus influence only a restricted compact planar set. In contrast the area of possible intersections of an unbounded quadric is larger, and thus modifications of the minimization diagram are more probable. In particular, when computing the lower envelope, an unbounded quadric can simplify the minimization diagram drastically. A single unbounded face can remain, while all previous (recursively computed) diagrams become obsolete. For an ellipsoid this probability is smaller. This fact is also reflected in the complexities of the final minimization diagrams. The number of surfaces attained in the envelope ($\#S$) and the number of faces ($\#F$), and thus for vertices and edges, is smaller for unbounded surfaces as for bounded ones; see Table 3.2 for examples.

It is easy to see that the performance of the computation is mainly influenced by three parameters. The first is the choice of the partitioning into subsets, which is beyond the scope of this work and we refer to [HSS08] that discusses a randomized choice. The second is the performance in two-dimensions itself, that is, how efficient are analyses of projected curves and pairs of them. Our implementation relies on a planar algebraic kernel for this task. The last factor is the amount of time spent to compute the relative z-orders of surfaces. The model presented in this chapter relies on (nested) square-root numbers provided by CGAL or algebraic expression from LEDA or CORE. In Chapter 5 we present another technique to compute the intersection pattern of surface along a vertical line.

Besides these elementary factors, it is also the combinatorial deduction employed by the algorithm itself that improves the general performance of the lower envelope computation. As explained the algorithms propagates continuity and discontinuity information to decide the relative z-order of incident planar cells. To quantify this improvement, we counted for example sets of 1000 surfaces the number of such savings. Table 3.3 shows the amount of finally executed comparisons compared with the number of actual comparisons (in parentheses) when not using combinatorial deduction. As one can see, the computation

1000 of	Number of comparisons over Point	Curve	above/below Curve
quadrics	0 (18315)	2804 (31373)	1273 (4638)
non-ellipsoids	0 (18087)	2386 (30777)	1273 (4640)
ellipsoids	0 (22747)	1292 (38172)	1282 (3798)

Table 3.3. Amount of required calls to compute the relative z-order of two surfaces during invocation of lower envelope algorithm for a set of 1000 arbitrary quadrics, 1000 non-ellipsoids, and 1000 ellipsoids. The number of operations when not propagating information to neighbored cells is shown in parenthesis.

of the envelope significantly benefits from this propagation of continuity and discontinuity information about the relative z-order of quadrics.

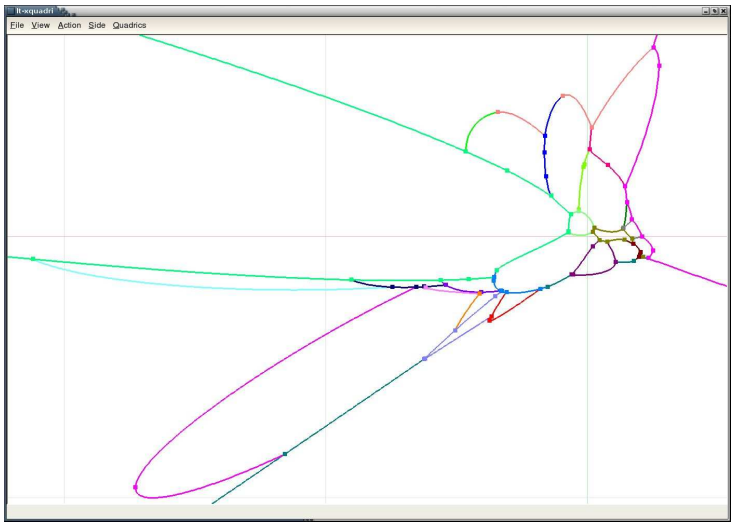


Figure 3.11. Cutout of the lower envelope of 400 quadrics, hyperboloids and ellipsoids. It consists of 30 faces, 4 of which are unbounded, 101 edges, and 76 vertices.

3.5. Variants

At the end of this chapter, we shortly want to mention some variants that can be extracted by slight modifications of the model that we presented for quadrics in §3.3.

Upper envelope Computing the upper envelope of a set of quadrics, requires only two small adaptions of the traits class. The first change affects the computation of the projected intersection of two xy-monotone parts of quadrics. Instead of returning the (weakly) x-monotone curves (and isolated points) that can be assigned to the lower parts of the two input quadrics, we only return the ones that can be assigned to the proper upper parts of the quadrics; this task is directly supported by the work in [BHK+05] on which we rely throughout this chapter. The other modification concerns the relative z-order over

different projected geometric objects. Remember that we first compute a suitable point p, then determine $r_1 := \min \mathcal{R}_1(p)$ and $r_2 := \min \mathcal{R}_2(p)$, and finally compare r_1 and r_2 to obtain the correct z-order. Once we store r_1 and r_2 in a proper number type (which is the second actual problem, besides computing p), we just generically call the comparison operator on this number type. When now computing upper envelopes, it suffices to consider $r_1' := \max \mathcal{R}_1(p)$ and $r_2' := \max \mathcal{R}_2(p)$ instead of r_1 and r_2. Our analysis of the involved algebraic degrees also holds for these values, and thus the same number types can be used. It remains to call `CGAL::upper_envelope`, that switches the algorithm to a status that takes the topmost surface in the labeling step, instead of the bottommost one.

It is easy to see that the computational effort for lower and upper envelopes following this strategy is identical, such that we abstain from reporting additional experiments for upper envelopes.

Arbitrary directions Lower and upper envelopes are with respect to the z-parallel projection onto the xy-plane. However, the traits class can also be used to compute lower and upper envelopes in arbitrary directions. To do so, if suffices to apply a rigid change of coordinates $R(x, y, z)$ that models a rotation. One can even think of more sophisticated linear mappings.

Instead of thinking that a point has moved in space by a map, it is possible to change the defining polynomials of input quadrics. That is, for a quadric q we define $R(q(x, y, z)) := (q \circ R^{-1})(x, y, z)$. It follows that $q(p) = 0 \Leftrightarrow R(q(R(p))) = 0$. Thus, in order to compute the envelope in the direction of the rotated xy-plane (defined by R), we consider as input the quadrics $R(q_1), \ldots, R(q_n)$. A simple example is the upper envelope where $R(x, y, z) = (x, y, -z)$.

Again, the combinatorial effort keeps unchanged, while the way we handle the rotation mainly influences the bit-lengths of the quadric's coefficients and the denseness of their defining polynomials. Thus, additional experiments could only reflect the efficiency of the quadrics' analyses with respect to these parameters. However, these consideration are already discussed elsewhere; see [BHK+05].

Voronoi Diagrams

Definition 3.5 (Voronoi Diagram). Let $\mathcal{O} := \{o_1, \ldots, o_n\}$ be a set of n pairwise disjoint convex objects in $\mathbb{R}^d$ and δ be a metric on $\mathbb{R}^d$. The *Voronoi Diagram of* $\mathcal{O}$ *with respect to* δ is a partition of $\mathbb{R}^d$ into maximal connected cells, each of which consists of the points that is closer to one particular object than to any other. A *Voronoi cell of object* o_i is the set $\{p \in \mathbb{R}^d \mid \delta(p, o_i) < \delta(p, o_j)\ \forall j \neq i\}$. The set of points $B_{i,j} := \{p \in \mathbb{R}^d \mid \delta(p, o_i) = \delta(p, o_j)\}$ is called the *bisector* of o_i and o_j.

As observed by Edelsbrunner and Seidel [ES86], every Voronoi diagram is exactly the minimization diagram of a set of surfaces in $\mathbb{R}^{d+1}$, that is, the projection of their lower envelope, where the surfaces are given by the graphs of functions $f_i : \mathbb{R}^d \to \mathbb{R}$ defined by $f_i(x) = \delta(x, S_i)$. More details on this duality can also be found in [dBvKOS00, §11.5].

We restrict in the following to $d = 2$, which implies that every two-dimensional Voronoi diagram can be computed by CGAL's `Envelope_3` package, provided that a proper traits class is supplied. Its two-dimensional objects and operations are responsible to build the planar subdivision of the diagram, the three-dimensional objects are supposed to model

the graph of the distance function for an object. Actually, the explicit storage of such a surface is superfluous in an efficient model, as it suffices to represent them by the objects o_i themselves. Three facts justify this simplification.

- Observe that in unbounded domains and metrics (as $\mathbb{R}^2$), the graph of the distance function has no projected boundary.
- The projected intersection of two sites is directly given by the bisector of the two planar objects. If possible, as usual, there is no need to construct the bisector by intersecting the distance surfaces.
- The desired relative z-orders of two sites can be directly encoded by comparing the distances of a point p to the two involved objects.

Actually, there is work by Halperin, Setter, and Sharir, that discuss this idea more detailed [HSS08]. It presents a framework to apply the divide-and-conquer approach for envelopes to compute various kinds of Voronoi diagrams and shows that through randomization the expected running time is near-optional (in a worst-case sense). The work also comprises a collection of robust and efficient traits classes to compute Voronoi diagrams, power diagrams, Apollonius diagrams in the plane. For some they rely on CGAL's new algebraic kernel and also the `Curved_kernel_via_analysis_2` as modelling the planar *ArrangementTraits_2* concept. Some of the diagrams can even be established on the sphere using CGAL's new `Arrangement_on_surface_2` package whose details we present in Chapter 4; see also [FHS08].

As mentioned, the explicit storage is not needed, however, we want to conclude this chapter with another modification of the *EnvelopeTraits_3* model for quadrics. Our goal is to use the modified version in CGAL's divide-and-conquer algorithm to compute the Apollonius diagram in two dimensions; see also CGAL's `Apollonius_graph_2` package [KY07].

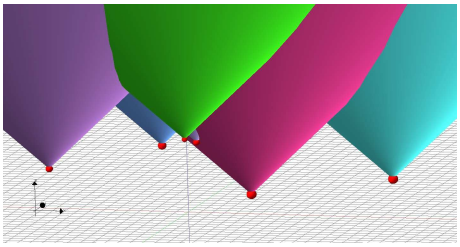
Definition 3.6 (Apollonius diagram). Let $A_i = (p_i, w_i)$, $1 \leq i \leq n$ be a set of sites, where $p_i \in \mathbb{R}^2$ and w_i is the weight of A_i. The *Apollonius diagram* of the A_i is the Voronoi diagram of the p_i with $\delta(x, p_i) := ||x - p_i|| - w_i$, where $||\cdot||$ denotes the Euclidean norm. The Apollonius diagram is also known as *additively weighted Voronoi diagram*.

If all w_i are equal, the Apollonius diagram is identical to the standard Voronoi diagram. Following Edelsbrunner and Seidel's relation, the Voronoi diagram of $\{p_1, \ldots, p_n\}$ is the vertical projection onto the xy-plane of the lower envelopes of a set of cones in $\mathbb{R}^3$. For each p_i we define a cone C_i whose apex is p_i itself. The cone's axis is a line parallel to the z-axis passing through p_i, its angle is 45°, and p_i is the cone's point with minimal z-coordinate.

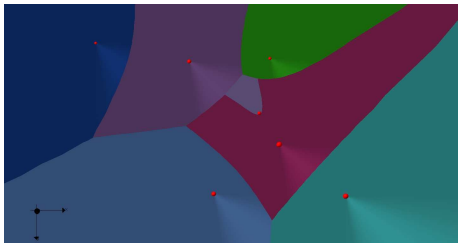
For the Apollonius diagram, we have to consider the weights in this geometric setting. For that reason the apex of C_i is shifted in z-direction by a quantity equal to the weight w_i of A_i. A site with positive weight corresponds to a cone whose apex is in the positive z-halfspace, the apex of a site with negative weight is in the negative z-halfspace. Figure 3.12 shows an example. The Apollonius diagram is attained by computing the vertical projection onto the xy-plane of the lower envelope of the shifted cones, that is, the cones' minimization diagram.

Remark (Shifted cones). First, observe that the Apollonius cell of a site A_i can be empty, which happens to be in the case, where A_i's shifted cone C_i is hidden in some other cone C_j for site S_j, $j \neq i$, that is, $C_i \cap C_j = \emptyset$; see Figure 3.13 for an example.

Figure 3.12. Cones that define the Apollonius diagram

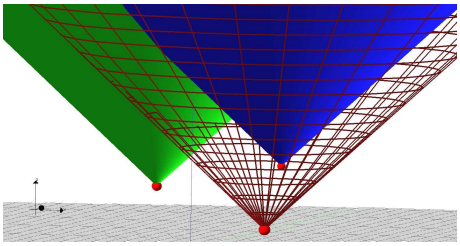


(a) Weighted points are transformed into cones whose apices' z-coordinates corresponded to weights.

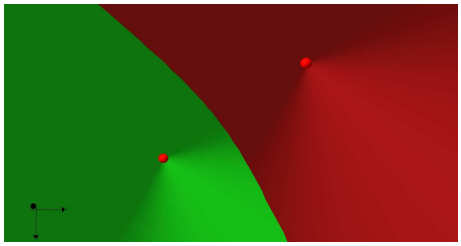


(b) The cones seen from $z = -\infty$. The lower envelope representing the Apollonius diagram can be guessed.

Figure 3.13. A hidden cone



(a) The input consists of three weighted points, but one of the corresponding cones is *hidden* ...



(b) ...and thus, it does not influence their lower envelope.

Second, as the projection of the lower envelope is z-axis parallel, the Apollonius diagram keeps unchanged, if we translate all cones by the same amount in z-direction. Without changing the algebraic complexity, we can move the apices of all cones into the positive z-halfspace. Thus, w. l. o. g., we assume that all $w_i > 0$, as it is the case in Figure 3.12. An implication of this fact is a geometric denotation: A site $A_i = (p_i, w_i)$ can be seen as a circle centered at p_i with radius w_i. For more details on this, we refer to [KY07].

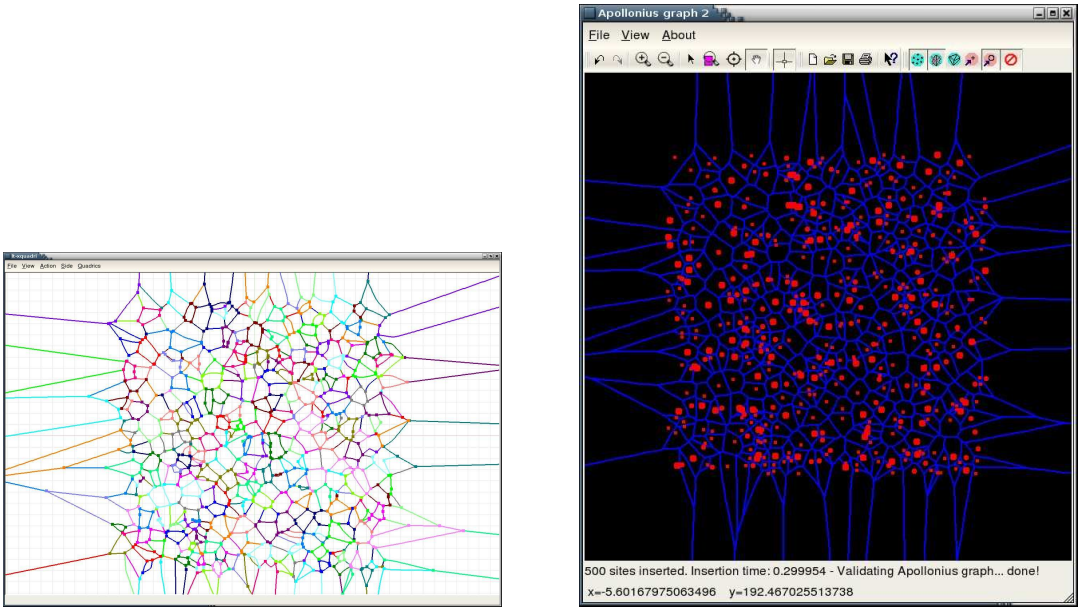
We finally explain which steps are required to compute the Apollonius diagram using our quadric traits. The key step is to construct the input surface for a site $A = (p, w)$. We refer to p_x as p's x-coordinate, and to p_y as p's y-coordinate. In contrast to the work in [HSS08], we use an explicit representation of the surface in $\mathbb{R}^3$ modelling A_i's distance function. In our case, we have to model a cone whose apex is at (p, w) and opening with 45° in positive z-direction. Unfortunately, there is no polynomial $q \in \mathbb{Z}[x, z, y]$ whose vanishing set $V(q)$ defines such a cone. However, if we mirror and copy the cone at the horizontal plane through its apex, we obtain a double-cone which can be defined algebraically, namely by $q = x^2 + y^2 - z^2 - 2p_x x - 2p_y y + 2wz + (p_x^2 + p_y^2 - w^2)$. Observe that $\deg_{\text{total}}(q) = 2$.

Thus, we actually could directly run CGAL's divide-and-conquer algorithm with our traits class to compute the lower envelope of these quadrics. However, this would not result in the minimization diagram denoting the Apollonius diagram of weighted points, for which we are looking for. The problem is that the input consist of double cones, but we actually want to compute the *lower envelope* of the cones' *upper parts*. In order to achieve this goal, we modify the implementation of our quadrical traits class at some positions.

- First of all, we return no projected boundary for a quadric. Actually, the projected boundary of a double cone is an isolated point, that is, the projected version of the double-cone's (singular!) apex. We just skip it. This is fine, as the distance function of a site is not bounded.
- When computing the projected intersections of quadrics, we change the code to only return the projected x-monotone curves that can be assigned to the upper parts of both double-cones. Observe that neither isolated points nor vertical curves occur in the projected intersections of two double-cones. This is actually true for all Voronoi diagrams, and should be incorporated when using a lower envelope algorithm for Voronoi diagrams.
- Finally, we adapt the computation of the relative z-order of two quadrics (here double-cones q_1 and q_2) in the obvious way. Instead of comparing $r_1 := \min R_1(p)$ with $r_2 := \min R_2(p)$, we now compare $r_1' := \max R_1(p)$ with $r_2' := \max R_2(p)$. In fact, we neither have to consider the comparison over boundaries nor over isolated points. This way, we are luckily left with the comparisons that can be determined with a rational p.

As a result, we can successfully compute Apollonius diagrams of (weighted) points using our modified traits. We tested various examples taken from CGAL's repository [KY07]. While all of them produced correct output, the performance numbers seen for these tests are bad, which somehow is an intrinsic problem. There are mainly three reasons.

- We consider explicit representations of the surfaces modelling the distance function (the shifted cones).
- The surfaces are more complicated than required, that is, we consider a double-cone instead of a single cone. This also has implications on the computation of the "bisector", which is here given by parts of the projected intersection of two double-



(a) Output with `xquadri`

(b) Output with demo from CGAL's `Apollonius_graph_2` package

Figure 3.14. Apollonius diagram of 500 weighted points

cones. The algebraic degree of such a projected intersection is 4, while all bisectors of weighted points in the plane are actually conics (i. e., curves of degree 2).

- Comparing the distances to two sites by checking the real relative z-order of two cones using pure exact arithmetic is far too complicated. In most cases, we should be able to derive the order of $\delta(x, p_i)$ and $\delta(x, p_j)$ by certified numerical approximations, for example using interval arithmetic.

For these reasons, we abstain from reporting extensive experiments on this naive approach, and refer to [HSS08] for a more sophisticated implementation of the problem using CGAL's divide-and-conquer algorithm for lower envelopes. However, it must be acknowledged for our toy example that in terms of coding it is simple to modify the traits in order to achieve results beyond pure envelopes.

In this chapter, we have seen how to come up with a model of CGAL's *EnvelopeTraits_3* concept for arbitrary quadrics. The model is based on a planar algebraic kernel which provides analyses of curves and pairs of them. In addition, we have shown how tiny modifications of the model (in collaboration with constructing proper input) render possible variants of envelopes, or even (naively) support another geometric problem, that is, the computation of the Apollonius diagram for a set of weighted points.

4 Two-Dimensional Arrangements on Surfaces

In this chapter we present a framework to compute arrangements of curves embedded on a two-dimensional parametric surface. Its development is driven by maximizing code-reuse. In particular, we generalize the sweep line algorithm and the zone algorithm to construct arrangements on the desired surfaces. The main tool in this direction is again the generic programming paradigm which allows to decouple the combinatorial representation from the actual supporting surface and the curves embedded on it. The framework originally extended CGAL's `Arrangement_2` package. For the upcoming CGAL 3.4 release it has been renamed to `Arrangement_on_surface_2`.

The outline of the chapter is as follows. We first present the framework, and how we have extended CGAL's `Arrangement_2` package to support various parametric surfaces as the unbounded plane, spheres, cylinders, tori, and more. This part of the chapter is based on results obtained in collaboration with Efi Fogel, Dan Halperin, and Ron Wein from Tel-Aviv University, Tel-Aviv, Israel, and Kurt Mehlhorn from the Max-Planck-Institut für Informatik, Saarbrücken, Germany. A short version previously appeared in [BFH+07]. Support for several surfaces with different kinds of curves embedded on each already exists. In the second part of the chapter we exemplary discuss two particular settings. As our initial example, we present what is needed to use the framework to construct, maintain, and overlay arrangements on an elliptic quadric. The curves embedded on such a surface are defined by its intersections with arbitrary quadrics. As final example we consider the case of a ring Dupin cyclide as the reference surface that is intersected by arbitrary algebraic surfaces. This implementation is joint work with Michael Kerber from the Max-Planck-Institut für Informatik, Saarbrücken, Germany. It has been presented in [BK08].

4.1. Setting and related work

We are given a parametric surface S in $\mathbb{R}^3$ and a set of curves $\mathcal{C}$ embedded on S. The curves $\mathcal{C}$ subdivide S into a finite number of cells of dimension 0 (vertices), 1 (edges), and

2 (faces). We refer to this subdivision as the *arrangement induced by* $\mathcal{C}$ *on* S and name it $\mathcal{A}_S(\mathcal{C})$. Until recently, CGAL's `Arrangement_2` package was only capable of constructing and maintaining arrangements induced by bounded planar curves; see §2.4.3. There is not even native support for unbounded curves. Handling such curves requires to cheat on the software with typically one of two options. The first solution is to clip the curves at some rectangle (or some other shape that is homeomorphic to a circle). However, it is the user's responsibility to choose the rectangle such that no essential information is lost, for example, a finite intersection point. In addition, the size of the rectangle also matters when trying to overlay two such arrangements. It must be ensured that both are clipped with respect to the same box, and if not, at least one must be recomputed, which is costly, non-trivial, and annoying. Alternatively, one can introduce a symbolic representation for a *point at an unbounded end* of a curve. Such a strategy has been formerly applied in EXACUS's old GAPS module. Its generic model of CGAL's *ArrangementTraits_2* concept was capable of dealing with unbounded curves. However, both solutions are somehow faking, generically inconvenient, and still insufficient for some applications.

The reason is that both still maintain only *one* unbounded face. But remember that CGAL uses planar arrangements to represent the minimization diagram $\mathcal{M}_\mathcal{S}$ for a set of surfaces $\mathcal{S}$, where each cell is labeled with the subset of surfaces that induce the lower envelope over that cell; see Chapter 3 and [Mey06a] for details. As remarked, if considering unbounded surfaces for lower envelopes, generally more than one unbounded face is expected in the representation of $\mathcal{M}_\mathcal{S}$. Actually, Mehlhorn and Seel [MS03] already proposed the *infimaximal frame* for extending the sweep line algorithm to handle unbounded curves. However, the design was intended for lines in the plane and it is unclear how it extends to arbitrary curves, be they algebraic or not. More problematic is, that their technique does not extend to parametric surfaces — a case that we especially want to include.

There already exist results that deal with arrangements on non-planar surfaces, for example, Hachenberger and Kettner compute two-dimensional boolean operations of geodesic arcs on a sphere [HK07a]. Such arrangements represent sphere maps around vertices in a three-dimensional Nef-like data structure [HKM07]. The sphere is also covered by Andrade and Stolfi [AS01], Halperin and Shelton [HS98], and recently by Cazals and Loriot [CL07]. Cazals and Loriot provide a software package that can sweep over a sphere constructing exact arrangements of arbitrary circles on it. They also show applications in computational biology that frequently employ spherical arrangements in molecular modeling: each sphere represents an atom of a molecule and the arrangement on the sphere represents the intersection pattern with neighboring atoms. Their extension, so-called anisotropic interactions of atoms, can be modeled using ellipsoids as primitive objects. The work by Berberich et al. [BHK+05] constructs arrangements on quadrics, which include ellipsoids. However, it considers two planar arrangements of projected intersection and silhouette curves, one for the lower part of a quadric, and one for its upper part; see §3.2 for an introduction. The approach requires as post-processing step the stitching of the two planar arrangement; this part is unfortunately not available. Stitching of sub-arrangements is also a key tool in work by Fogel and Halperin [FH07]. They model the single arrangement of arcs of great circles on a sphere with six arrangements of linear segments in the plane that correspond to the six faces of a cube circumscribing the sphere.

None of the previous solutions tackles in a generic fashion all problems that can occur, such as clipping, stitching, or the support for various curves. This justifies our goal to

develop a framework that effectively and generically deals with all of them. We start with the unbounded plane because this is a special case of a bijectively parametric surface. In a second step, we generalize, and allow non-injectivity on the boundary of the parameter space, which lead to the current implementation of the package.

Remember CGAL's `Arrangement_2` class-template as presented in §2.4. It is parameterized in two arguments. First, it takes a `GeometryTraits_2`[23] that basically defines the curves to consider and operations on them. Once curves have been split into (weakly) x-monotone curves in a *pre-processing step*, these operations are used to feed the internal algorithms and data structures to construct, maintain, and overlay arrangements of them. The second parameter is the `Dcel` type. Each arrangement internally maintains an instance of this type. Its vertices are enhanced with geometric points and its edges carry (weakly) x-monotone geometric curves. Each of the beforehand mentioned maintenance and construction operations modify the combinatorial structure of the internal DCEL-instance, in sync with updating the stored geometric objects. We listed in §2.4.3 basic insertions (and respective deletions) that consistently modify the DCEL. Such consistent modifications are called by the constructing visitor class for the two main algorithms that construct (or overlay) arrangements, namely the sweep line algorithm, and the incremental insertion using the zone algorithm. Actually, these algorithms only produce a *canonical output* and it is the visitor that defines which basic insertions must be called. The canonical output of an algorithm is defined by the execution path of the algorithm, which itself is controlled by the outcome of geometric predicates and constructions provided by the given geometric-traits class. We shortly repeat the internal flow of each algorithm in order to understand for what we provide geometric operations.

- The **sweep** (of weakly x-monotone curves) involves the handling of events and the maintenance of the status-line. Handling events comprises to maintain the sorted event-queue, that is, new events must be inserted, while the minimal event at a time is removed. In the planar case, events are endpoints of curves, or their (zero-dimensional) intersections, while their order is given by the lexicographical comparison of points stored with events. The status-line is updated, whenever a curve reaches its endpoints, a new curve starts, or the order of curves changes. In either case, curves that become adjacent in the status-line are checked for intersections to the right of the sweep line and any such intersection is inserted into the event-queue.
- The central operations for the **zone algorithm** are to locate the ends of a new curve, and to compute the curve's intersections with existing curves. That is, we require geometric operations to locate points, and to intersect curves.

We want to generalize this existing work to two-dimensional parametric surfaces. The geometry of S is captured by a parameterization as in Definition 2.30, that is, there is a function $\varphi_S : \Phi = U \times V \to \mathbb{R}^3$ whose image defines S. We allow intervals $U = [u_{\min}, u_{\max}]$, $U = [u_{\min}, +\infty)$, $U = (-\infty, u_{\max}]$, or $U = (-\infty, +\infty)$, and similarly for V. Intervals that are open at finite endpoints bring no additional power and we therefore do not discuss them here. Curves on parametric surfaces are defined as in Definition 2.40; here we have $D = \Phi = U \times V$. What used to be x-monotonicity for bounded planar curves, is now naturally extended: A curve γ is called *sweepable* if it is (weakly) u-monotone, that is, if

[23] Expects basically a model fulfilling the *ArrangementTraits_2* concept, however, there is a hierarchy of concepts, and each refinement level of the hierarchy is valid (for a certain goal).

$t_1 < t_2$ then $\gamma(t_1) <_{\text{lex}} \gamma(t_2)$, where $<_{\text{lex}}$ now denotes lexicographic uv-ordering. For the internal arrangement tasks, we consider all curves to be sweepable. If an input curve does not fulfill this property we apply, as before, a pre-processing step. The standard planar sweep, for example, corresponds to $U = V = (-\infty, +\infty)$, and $\varphi_S(u, v) = (u, v, 0)$, but none of the input curves extends to infinity. Other instances are given in Example 2.31, or appear in the remainder of this chapter.

While the input processing turns to out to be relatively simple, we have to work harder for the internal structures of the arrangement package. In particular, we expect answers to the following raised questions:

1. How do we keep the general flow of the constructing algorithms mainly unchanged?
2. How do we ensure to properly construct and update a DCEL with respect to given surface?

We do give the answers for both questions in several steps. In §4.2 we first discuss how to ensure a canonical output for the sweep line and the zone algorithm. The discussion starts with bijective parameterizations and then we remove injectivity on the boundary of the parameter space. The actual construction of the DCEL is presented afterwards in §4.4. As for some surface there often exist several valid encodings of an induced arrangement as a DCEL. Our solution aims for this flexibility.

4.2. Sweeping and zoning on a surface

In this section we explain how to modify the two main algorithms such that they can be executed for a parametric surface, to be prepared for the second task: An attached visitor class should be able to correctly interpret the visiting pattern of the algorithm for its purposes, namely to construct the DCEL. We mainly consider the sweep algorithm in this section, and refer to the simpler zone algorithm shortly at the respective places.

Sweeping a parametric surface, in terms of the standard two-dimensional sweep algorithm, should be correctly seen as taking place in the parameter space, that is, we sweep with a vertical line $u = u_s$ from $u_{\min}$ to $u_{\max}$. However, for reasons of intuition, it can be more convenient, to see it from a different angle, namely to *sweep over* S with the curve on S defined by the moving image of the vertical line $u = u_s$ under φ_S. Both views are valid. The considerations of this section assume that the sweep (zone) takes place in the rectangle defined by U and V. We switch to the *surface-view* in §4.4, when actually constructing the DCEL on S.

Let us state an important remark with respect to the chosen parametric view.

Remark (Parameterization). We do not expect surfaces and curves to be given in parametric form, but consider this tool for the definition of the problem, and for its realization of the adapted algorithms. In §4.3 we learn that the algorithms still learn about surface, curves, and points only through a well-defined set of geometric predicates provided by an extended geometric-traits class. It is the choice of the traits' implementer how to compute these pieces of information. While the example of §4.6.2 does really deploy the parameterization, the example exercised in §4.6.1 cleverly combines planar counterparts to deduce the expected answers for the parameter space.

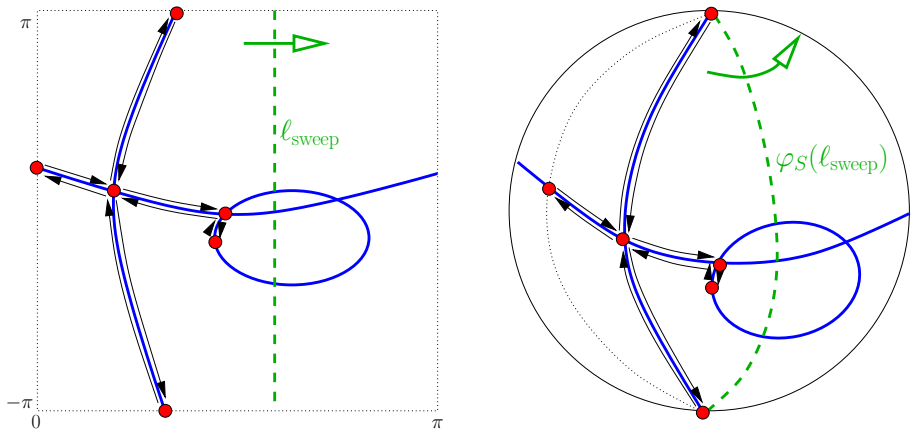


Figure 4.1. Sweeping a sphere: sweeping a line in parameter space from $u = 0$ to $u = \pi$ corresponds under φ_S to sweep a meridian from $0°$ to $360°$ around the sphere.

4.2.1. Bijective parameterizations

Our first generalization discusses surfaces whose parameterization is bijective. At first sight, there seems to be simple solution to just incorporate the parameterization into the geometric-traits class. This strategy is even fine and no further considerations must be made — if only bounded curves occur. The true differentiation from the standard sweep line algorithm emerges in the case, where curves are allowed to extend to infinity. Or in other words, we can neither restrict U nor V to an interval $[-M, M]$, for sufficiently large $M \in \mathbb{R}$, such that the event queue (which contains ends of (weakly) u-monotone curves and their intersections) only has to deal with finite points as event. If unbounded curves are allowed, we face the problem that these curves do not have such finite endpoints.

Our solution to this problem is to extend the definition of an event. We basically distinguish two kinds of event. The first kind, an *interior event* encapsulates (as before) a finite point. For the second kind, we introduce the term of a *curve-end*. Each (weakly) u-monotone curve $\gamma : D \to (-\infty, \infty) \times (-\infty, \infty)$ has two curve-ends, the lexicographical minimal one, and the lexicographical maximal one (in uv-ordering). A curve-end may either be a finite endpoint or represent an unbounded entity in case that the sequence of points attained by γ towards the specified end approaches the boundary of the parameter space: More precisely, we say that the curve-end $\langle \gamma, 0 \rangle$ *approaches the left (right) boundary* if $\lim_{t \to 0+} \gamma(t) = (-\infty, v_0)$ ($(+\infty, v_0)$, respectively), for some $v_0 \in \mathbb{R} \cup \{-\infty, +\infty\}$, and that it *approaches the bottom (top) boundary* if $\lim_{t \to 0+} \gamma(t) = (u_0, \pm\infty)$ for some $u_0 \in \mathbb{R}$.

Remark (Asymmetry). Observe the slight asymmetry in the definition. If a curve-end actually approaches one of the four cases $(\pm\infty, \pm\infty)$, we subsume them belonging to the left or right boundary. This simplifies the later discussion, and also reflects the asymmetry we already noticed for predicates required for the (bounded planar) sweep line algorithm.

Following this notation, we can use the pre-processing step to associate an event with each end of a (weakly) u-monotone curve: An *interior event*, associated with the finite endpoint, is assigned if $0 \in D$ ($1 \in D$). A *near-boundary event*, associated with the unbounded curve-end $\langle \gamma, 0 \rangle$ ($\langle \gamma, 1 \rangle$), is assigned if $0 \notin D$ ($1 \notin D$).

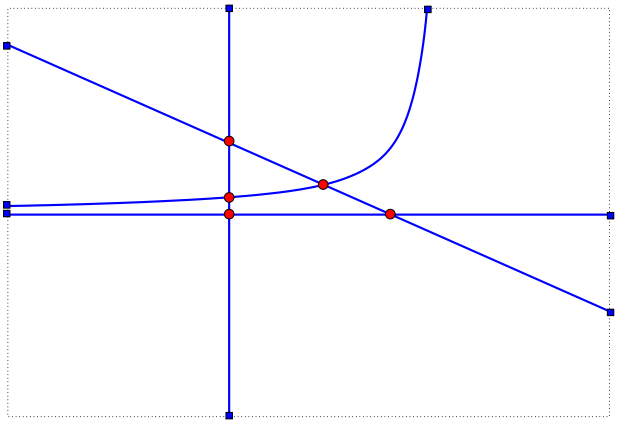


Figure 4.2. Arrangement of four infinite curves that intersect in 5 finite points. To sweep the curves, we have to define a lexicographic order that can handle with finite points, but also with the 8 infinite curve-ends.

The order of events in the event queue of the standard sweep line procedure is simply defined by the uv-lexicographic order of finite events. For our extended definition of events we augment their comparison procedure. It is required to also handle those events associated with unbounded curve-ends as well. This is done by subdividing the procedure into separate cases.

First of all, two finite events are still ordered purely uv-lexicographically. It remains to define the order of two events where at least one is an unbounded curve-end. Most of which can be handled in a straightforward manner. For example, it is clear that an event *on the left boundary* is smaller than any event associated with a finite point, which is smaller than any event *on the right boundary*. To compare two curve-ends approaching the left (right) boundary, we consider the intersection of relevant curves with a vertical line $u = u_0$ for small (large) enough u_0 and return the v-order of these points. "Small enough" ("large enough") means that the result does not depend on the choice of u_0 (or v_0), which is well-defined as curves are allowed to intersect only at finitely many points. That is, we are interested in the relative vertical order of two curves immediately to the right of the left (to the left of the right) boundary. There is the exception of overlapping curves, which constitutes a special case on its own: The comparison of events representing unbounded curve-ends of overlapping curves are allowed to return equal. Two cases are left, namely to compute the relative horizontal (in u-direction) order of an interior event with a near-boundary event for the bottom- or top-boundary, and to compute the same order for two near-boundary events where both attached curve-ends approach the bottom- or top-boundary. Note again that it suffices to only consider a situation close enough to the boundary, that is, intuitively one can choose finite points close enough to the boundary that reflect the correct order and return their relative horizontal order. An illustration of the conceptual description of the required comparisons is given in Figure 4.3. Technically, they are collected in an extended version of CGAL's *ArrangementTraits_2* concept that we present in §4.3. As for each concept, it is not specified how to finally implement it.

Observe that we only enhance events and their order for the sweep line. The actual sweep process remains mostly unchanged. In fact, some maintenance operations for the status-line can even be established *without executing any geometric comparison*. Note

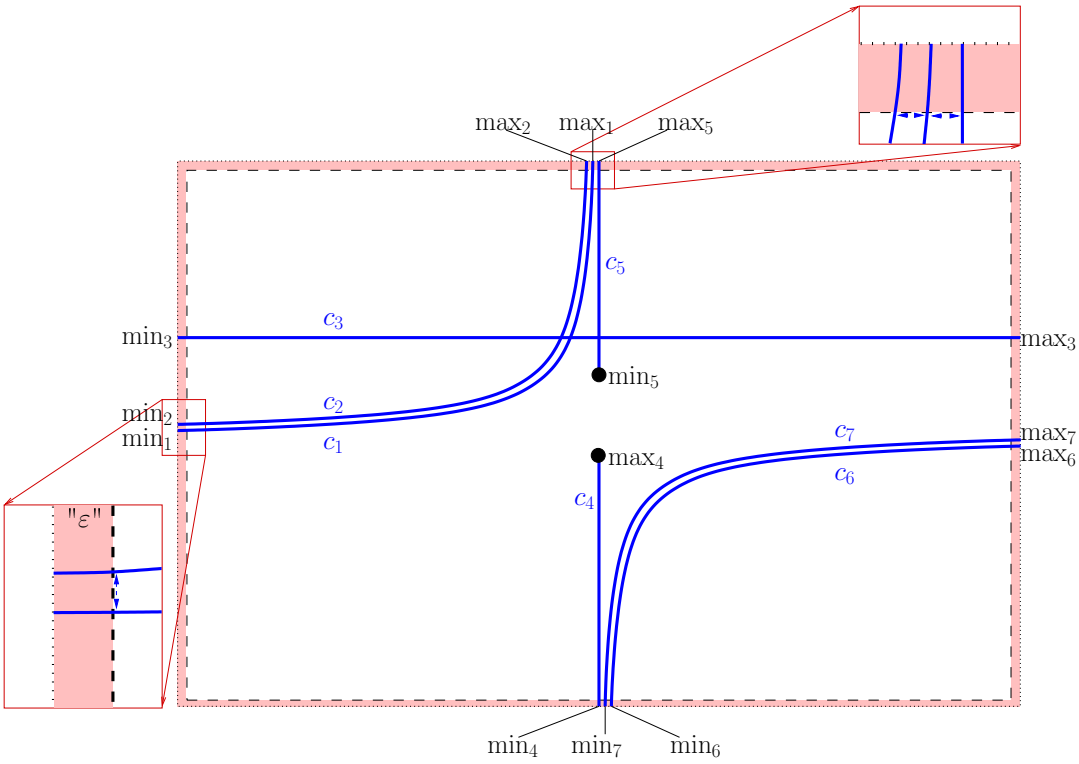


Figure 4.3. Compare curve-ends near boundary: The view is in parameter space. Left boundary: $\min_1 <_y \min_2 <_y \min_3$. Right boundary: $\max_6 <_y \max_7 <_y \max_3$. Bottom and top boundary: $\max_2 <_x \max_1 <_x \min_4 =_x \max_4 =_x \min_5 =_x \max_5 <_x \min_7 <_x \min_6$. The comparison functors that we present in §4.3.1 are responsible to ensure this order. But: The actual computation is not expected to elaborate the parameterization.

that the first events in the event-queue tagged with *on left boundary* are already sorted in correct increasing v-order. Thus, as long as the sweep extracts such near-boundary events, we can simply put the corresponding curve(s) at the top of the status-line. Similarly, when we proceed with the sweep line algorithm and handle a minimal event whose curve-end approaches the bottom (top) boundary, we know that the curve must lie below (above) all other curves currently maintained in the status-line. Thus, we can simply insert the curve at the bottom (top) of the status-line without any additional geometric operation. As all intersection points do not take place on the boundary of the parameter space, there is also no need to modify the sweep in its intersection handling.

As a result, we centralize the handling of events near the boundary in the sweep line algorithm itself, while keeping the geometric interface small. In addition, we obtain a way to avoid some geometric comparisons in the maintenance of the status-line, which are usually costly, especially if a model implements the exact geometric computation paradigm. The output of the sweep still consists of a unique visitor pattern. By now, it is open how to transform it into a DCEL-representation that stores the induced arrangement. We describe this step in §4.4.

The zone algorithm for a given curve γ consists of two main steps, namely the localization of γ's ends and to compute γ's intersections with existing curves in the arrangement. Again, the intersections do not take place on the (unbounded) boundary of the parameter space, and thus, no modifications are needed. In contrast to the localizations. For them it is expected to return the cell of the existing arrangement to which the given end of γ belongs. That is, we either obtain a face, an edge, or a vertex. Note that we have tagged γ's ends with information whether each lies in the interior of the parameter space or which boundary it approaches. This information is needed, but not sufficient. In fact, we do need knowledge on how the arrangement (containing curves approaching the boundary of the parameter space) is represented as DCEL. Only this allows to return the correct DCEL-record. In §4.4 we generalize the DCEL-representations for arrangements, and a part of the task is the localization of curve-ends on the boundary.

4.2.2. Allowing non-injectivity on the boundaries

We just introduced events near the boundary of the parameter space, which we by now only use to represent end of curves that extend to infinity. But what we describe also enables an elegant generalization of the sweep line procedure for curves embedded on a parametric surface in $\mathbb{R}^3$; see Definition 2.30 and Example 2.31 for such surfaces. A parameterization φ_S is allowed to be non-bijective, that is, some points in S may have multiple pre-images in Φ. In fact, we allow one-dimensional sets to do so, as it is the case for rational surfaces.

Let us exemplary remember the unit sphere, where we have $\varphi_S(-\pi, v) = \varphi_S(+\pi, v)$ for all v, while $\varphi_S(u, -\frac{\pi}{2}) = (0, 0, -1)$ and $\varphi_S(u, \frac{\pi}{2}) = (0, 0, 1)$ for all u. The curve $v \mapsto \varphi_S(-\pi, v)$ is a meridian on the sphere, analogous to the *international date line*, and the points $(0, 0, \pm 1)$ correspond to south and north pole, respectively. The non-injectivity of φ_S induces the *date line*, which implies that a closed curve on the sphere, for example the equator, may be the image of a non-closed curve in parameter space. The poles also pose another problem: They always lie on the *sweep curve* (i. e., the image of φ_S for $u = u_0$, for u_0 from $u_{\min}$ to $u_{\max}$) during the sweep.

The example of the unit sphere introduces two cases where we relax the requirements for surface parameterization, in order to model a wider range of surfaces, as cylinders,

paraboloids, tori, and their homeomorphic counterparts. Central is that we require bijectivity of φ_S only in the *interior* of Φ, while non-injectivity is allowed on the boundary of Φ, denoted by $\partial\Phi$. More precisely, we demand that $\varphi_S(u_1, v_1) = \varphi_S(u_2, v_2)$ with $(u_1, v_1) \neq (u_2, v_2)$ implies $(u_1, v_1) \in \partial\Phi$ and $(u_2, v_2) \in \partial\Phi$.

Before allowing non-injectivity in a controlled way, we precise the weak definition from §4.2.1 for the location of a point in parameter space.

Definition 4.1 (Locations). Let $p = (u, v) \in \Phi = U \times V$. We say that p *lies on the left boundary* if $u = u_{\min}$, or that p *lies on the right boundary* if $u = u_{\max}$. If p does neither lie on the left nor on the right boundary, we say that p *lies on the bottom boundary* if $v = v_{\min}$, or that p *lies on the top boundary* if $v = v_{\max}$. If no such condition holds, we say that p *lies in the interior of* Φ.

This disjoint partitioning of Φ implies four boundary sides $\partial_l\Phi, \partial_r\Phi, \partial_b\Phi, \partial_t\Phi$ of the parameter space, and its relative interior $\mathring{\Phi}$. Observe again, that the left and right side are defined (for the known reason) asymmetric to the bottom and top side.

For the four sides of $\partial\Phi$ we allow two kinds of relaxations, given in Definitions 4.2 and 4.3.

Definition 4.2 (Contraction). A closed side $\partial_s\Phi$ is called *contracted* if the image of $\partial_s\Phi$ is a single point $p_s \in S$, that is, $\forall(u, v) \in \partial_s\Phi$ it holds $\varphi_S(u, v) = p_s$. We call p_s a *contraction point*.

In the running example of the sphere, we have that the bottom and the top boundary, inducing the south and north pole, are contracted. That is, $\forall u \in \mathring{U}$ we have $\varphi_S(u, v_{\min}) = (0, 0, -1)$ and $\varphi_S(u, v_{\max}) = (0, 0, 1)$.

Definition 4.3 (Identification). Two opposite closed sides of $\partial\Phi$, that is, either $\partial_l\Phi$ and $\partial_r\Phi$ or $\partial_b\Phi$ and $\partial_t\Phi$, are called *identified* if they define the same curve γ_I on S. We call γ_I the *curve of identification*. More precisely, identifying the left and right boundary means that $\forall v \in V, \varphi_S(u_{\min}, v) = \varphi_S(u_{\max}, v)$, while identifying the bottom and top boundary implies $\forall u \in \mathring{U}, \varphi_S(u, v_{\min}) = \varphi_S(u, v_{\max})$.

We detect an identification of the left and right boundary for the parameterized unit sphere. Its curve of identification induces the *international date line*. Let us see what other surfaces we can model using identification and contraction.

- A *triangle* with corners $(a_1, b_1), (a_2, b_2)$, and (a_3, b_3) is parameterizable via $\Phi = [0, 1]\times[0, 1]$ with $\varphi_S(u, v) = (a_1+u(a_2-a_1)+uv(a_3-a_2), b_1+u(b_2-b_1)+uv(b_3-b_2), 0)$. We observe that $\partial_l\Phi$ is contracted.
- An open or closed *cylinder* is modelled by identifying, for example, $\partial_l\Phi$ and $\partial_r\Phi$, while V is an open or closed interval.
- A *torus* is modelled by identifying both opposite pairs of $\partial\Phi$; see also §4.6.2.
- A *paraboloid* or *cone* is modelled by identifying $\partial_l\Phi$ and $\partial_r\Phi$, and contracting $\partial_b\Phi$. If the surface opens to infinity, $\partial_t\Phi$ should be tagged as unbounded.

For each of them, there exist other equivalent combinations with exchanged sides. However, we explicitly forbid to combine contraction and identification on one boundary side. This would allow to model a genus-one surface with a single pinch point by identifying both opposite pairs, while one pair is also contracted. Although this surface would be sweepable with our framework, we exclude it, as an embedded arrangement might not be representable

using a typical DCEL-structure. The reason is that the DCEL-vertex for the pinch point can become incident to two different faces, which is not covered by DCEL-representations; see Figure 4.4 for such a surface.

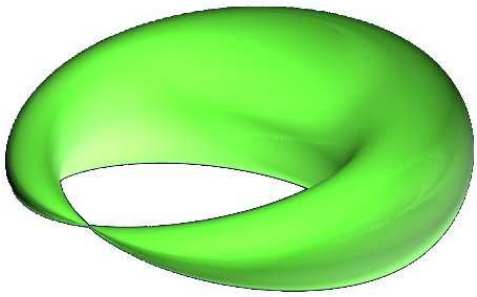


Figure 4.4. The "croissant": a surface with one pinch point and whose parameterization would contain two identifications. One of these identifications must actually be contracted as well. © Herwig Hauser, www.freigeist.cc/gallery.html

At this point we switch to a rather generic sweep, that is, we are given a surface S and (notationally) its parameterization φ_S. We know for each side of the parameter space an explicit tag annotating its type, that is, either *bordered*, *unbounded*, *contracted*, or *identified*. Bordered constitutes a finite curve of delimitation, as for the example in the case of a triangle. An identification tag on one side of the boundary implies the same tag for its opposite side. As input, we are also given a set of curves embedded in S. Conceptually, we aim to sweep over the parameter space of S, that is, the rectangle defined by $U \times V$ with special properties at its boundaries.

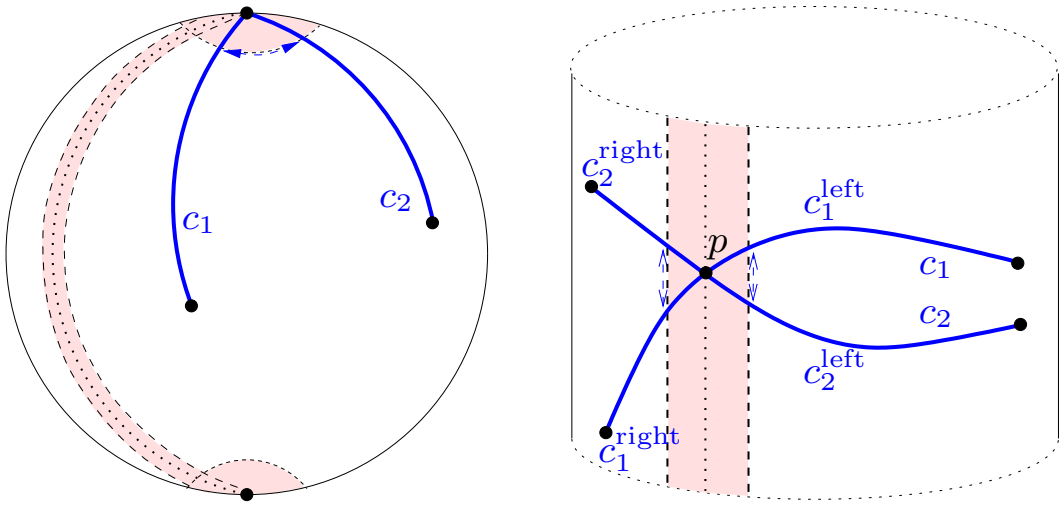
We come to the phases of the sweep, and start with pre-processing of input curves (in parameter space) to feed the actual sweep. Sweepable curves are expected to meet two criteria: First, as for the standard sweep, curves are expected to be (weakly) monotone in the direction the sweep line moves. In our case, we split input curves into their (weakly) u-monotone components. This splitting already partially fulfills the other criterion: A curve that is not fully contained in $\partial\Phi$ is expected to touch $\partial\Phi$ only at its ends. This condition implies that we split curves whose interior intersects with a contracted or identified side. Note that due to achieved u-monotonicity, it only remains to check the bottom and top boundary for this purpose. After this partitioning, the curves with their curve-ends can be characterized. A first observation is that only non-closed curves in Φ exist. The *interior* of each such curve is either completely contained in some $\partial_s\Phi$ (maybe in its identified counterpart, too), or it completely lies in $\mathring{\Phi}$. In the latter case, the two ends are allowed to meet (not necessarily[24]) distinct boundary sides. As in §4.2.1 each such curve-end can be uniquely annotated with one out of five locations: $\partial_l\Phi$, $\partial_b\Phi$, $\mathring{\Phi}$, $\partial_t\Phi$, and $\partial_r\Phi$. Note that in case of identification, actually two choices exists, but the connection to the interior of the curve gives the desired one; see, for example, Figure 4.5 (b). The two curves c_1 and c_2 cross the identification in p. However, we split them to be u-monotone in the parameter space, and obtain c_1^{left}, c_1^{right} and c_2^{left}, c_2^{right}. The minimal ends of c_i^{left} lie on the left boundary, while the maximal ends of c_i^{right} lie on the right boundary. All other ends exist in the

[24]Note that a u-monotone curve cannot start and end on $\partial_l\Phi$. The same holds for $\partial_r\Phi$. There is no such restriction for the bottom and top boundary.

interior of parameter space. Input fully embedded on a boundary is discussed below.

We next study how to sort the event-queue of the sweep. We can assume to execute a sweep over the open surface attained by $\varphi_S(\mathring{\Phi})$, while handling ends of curves meeting $\varphi_S(\partial\Phi)$ are handled following the strategy from §4.2.1. We are able to derive the correct order of events using the explicit distinction between *interior events* that are associated with points in $\mathring{\Phi}$ and *near-boundary events* that occur for curve-ends approaching $\partial\Phi$. Again, most comparisons of such events are straightforward, while all remaining can be answered using exactly the same set of additional geometric predicates as introduced for unbounded curves — assuming they take place in parameter space; see Figure 4.3. We compare curve-ends in an ε-distance away from boundary (in the direction of $\mathring{\Phi}$) to obtain a unique order of different near-boundary events that do not have a trivial order. Note that the ε-environment is *conceptual* only, that is, how the actual comparison is achieved is not determined, in particular, it is not enforced to compute in parameter space. Figure 4.5 presents two examples on surfaces. In §4.6 we explain how to implement the comparisons for elliptic quadrics and ring Dupin cyclides.

Figure 4.5. Two examples of comparisons near non-unbounded boundaries



(a) Compare u near top contraction: $c_1 <_u c_2$

(b) Compare v near left-right identification: $c_2^{\text{left}} <_v c_1^{\text{left}}$ and $c_1^{\text{right}} <_v c_2^{\text{right}}$

With this strategy each curve-end finally meeting a boundary side gets its own event for the sweep, that is, if we have k sweepable curves incident to a point on $\varphi_S(\partial\Phi)$ (namely a contraction point or a point on the curve of identification), we handle k separate events that relate to this point. An example is a set of longitudes on the sphere. The maximal end of each longitude results in its own event, although eventually all longitudes meet in the north pole; see the example depicted in Figure 4.6. Our current goal is only to obtain a unique order for the sweep events. The sweep itself proceeds then exactly as the standard sweep does; see Algorithm 2.13. In §4.4 we explain how we *tie all the loose ends* left out by the sweep procedure and construct a well-defined DCEL that represents an arrangement of curves on S. Or more exemplary, how we obtain a single DCEL-vertex for the sphere's

north pole.

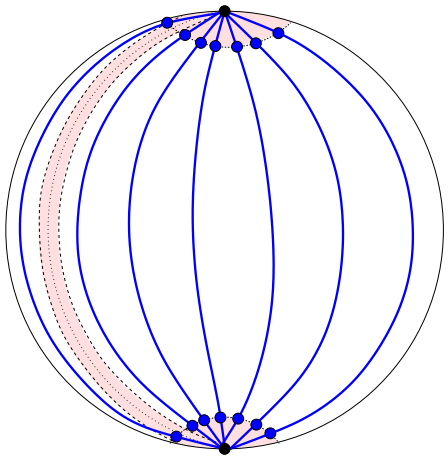


Figure 4.6. Not each sweep line event (here blue nodes near contractions, i. e., poles) are supposed to model a DCEL-vertex. In §4.5 we discuss how to unify different but related events, and how the vertices representing such contractions (north and south pole) are created.

We are left with the completion of the sorting of events, that finally should also comprise input that is fully contained in the image of some $\partial_s\Phi$ For such points and ends of such curves, we introduce *boundary events*. In the following we explain how such events are ordered among each other, and in comparison to interior and near-boundary events.

We start with the simple case of a contracted side. Note that the only boundary event that can occur relates to a single isolated point. We need to check whether a point lies on such a contraction, and if so, we create the special event without any incident curve. The handling of an isolated event during the sweep can be kept unchanged, however, we need to determine the position of this isolated boundary event in the event queue in relation to other events. The solution is to define that this special event is always the smallest event that belongs to the corresponding side of the boundary. This choice already defines the order with respect to every other near-boundary event, but also to interior events. See event be_6 in the example depicted in Figure 4.7.

Bounded sides and identified sides are left. We again expect a possibility to check whether a point or a curve is contained in such a side of the boundary; see §4.3 for the technical details. If an object is detected to lie on a left-right identification, we consider its left pre-image, while we handle an object detected to lie on a bottom-top identification as solely belonging to the bottom boundary. This handling is only internal, that is, in case of an identification the user has not to care about these details; see our respective interface in §4.3.7.

We create a boundary event for each such isolated point (no incident curves), for each minimal point, and for each maximal point of such a curve. We remark that the minimal or maximal end of such a curve can be unbounded; for example in the case of an infinite cylinder. Considering this fact, the order of boundary events on a single side of the boundary is given by comparing their u- or v-coordinates, depending on the side in focus. But this order is not sufficient if we also have to compare a boundary event with

near-boundary and interior events. For the sweep we define the following order among different kinds of events at the same coordinate:

- There are some straightforward relations:

$$B_l <_u N_l <_u I <_u N_r <_u B_r$$

with

$$\begin{aligned} B_l &:= \{be | be \text{ is a boundary event on } \partial_l\Phi\} \\ N_l &:= \{ne | ne \text{ is near-boundary event related to } \partial_l\Phi\} \\ I &:= \{ie | ie \text{ is an interior event}\} \\ N_r &:= \{ne | ne \text{ is a near-boundary event related to } \partial_r\Phi\} \\ B_r &:= \{be | be \text{ is a boundary event on } \partial_r\Phi\} \end{aligned}$$

Note that within each set the v-order must still be determined to know "$<_{\text{lex}}$". We expect corresponding comparisons; see §4.3.

- We are left with interior events and those related to $\partial_b\Phi$ and $\partial_t\Phi$. We first order them by u-coordinate.[25]
- If two of them share the same u-coordinate, the order of two events is given by the following symbolic perturbation.
  - The boundary event of an ending curve is smaller than a near-boundary event of an ending curve.
  - The near-boundary event of an ending curve is smaller than an isolated boundary event or an interior event.
  - An isolated[26] bottom boundary event is smaller than an interior event which is smaller than an isolated top boundary event.
  - An isolated boundary or an interior event is smaller than a near-boundary event of a starting curve.
  - A near-boundary event of a starting curve is smaller than a boundary event of a starting curve.

  The order of near-boundary events again requires an external geometric predicate. All other members of a set of equivalent events can be assumed to be equal.

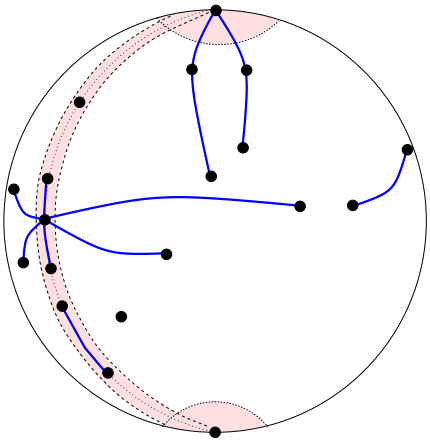
We remark that most of this case-distinction is internal and thus serves code reuse. The geometric-traits class is only expected to provide the mentioned, specialized, comparisons. Among them, it is expected to compare u- or v-coordinates of (always finite) points on $\partial\Phi$. In §4.4 we see another usage of comparisons of coordinates on a boundary.

Let us summarize what has been done in order to keep the sweep generic for a parameterized surface S. Instead of a single event type for finite points, we rely on three kinds of events, namely interior events that correspond to points in $\mathring{\Phi}$, near-boundary events that encode ends of curves on $\partial\Phi$ whose interior is still contained in $\mathring{\Phi}$, and boundary events for isolated points on $\partial\Phi$ and ends of curves that are fully contained in $\partial\Phi$. *We define a unique uv-lexicographic order of all events, described by large, but internal, case distinction, that*

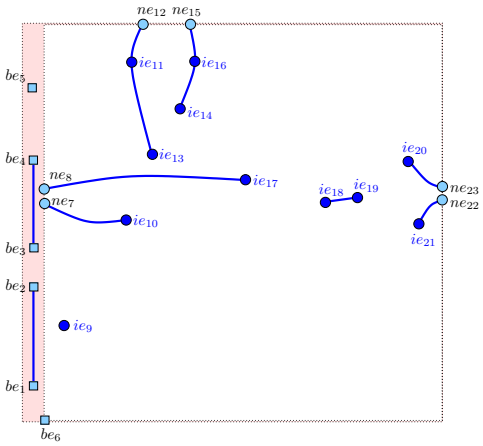
[25] Observe that u-coordinates of points and curve-ends on bordered and identified bottom- and top-boundaries are available.

[26] We consider near-boundary events of *vertical* curves as "isolated" as well.

Figure 4.7. Events on the sphere for input that also comprises curves and points on $\partial\Phi$



(a) Input: Two curves and one isolated point on the identification, an isolated point at the south pole, 4 curves meeting the identification, 2 curves incident to the north pole, one interior curve, and one isolated vertex in the interior.



(b) Events: 6 boundary events (be_i), 6 near-boundary events (ne_j), 11 interior events (ie_k). The indices indicate the uv-lexicographical order, derived using the locations of curve-ends (and points) in Φ and by on-boundary-, near-boundary-, or interior-comparisons.

relies on a small set of simple geometric comparisons. We give the full list in §4.3. Why do we exert ourselves with this distinction? The reason is simple: We do not want the user to do it. Most of these comparisons are straightforward and would appear repeated times for each family of curves that are supported on a specific parametric surface. With the chosen approach, we maximize code reuse. By splitting the annoying task into easy-to-solve subtasks, we also reduce the expected level of expertise for someone who plans to provide new curves. There is another reason: Theoretically, it is possible to already unify events, for example, combining boundary and near-boundary events belonging to the same point p on S. However, this reduces the flexibility to choose a certain Dcel-representation for some parameterization. We learn in §4.4 that is it beneficial to give the responsibility of such a unification to another entity.

For the zone algorithm the situation is similar as for unbounded curves. We again have to locate the Dcel-feature that is met by a curve's minimal or maximal end. But to provide this information, knowledge how Dcel-records encode bordered, unbounded, contracted, or identified sides is expected. Thus, we postpone this problem to §4.4.

4.3. Extending the *ArrangementTraits_2* concept

As explained in §2.4.3 the `Arrangement_2` package is instantiated with a model of Cgal's *ArrangementTraits_2* concept that provides types and geometric constructions and predicates in order to support the arrangement construction and maintenance. The version of the concept until Cgal 3.2 supports bounded curves, while implicitly assuming that the embedding surface is the xy-plane. We refer to this version as the *NoBoundaryTraits* refinement. This version also constitutes the root of a hierarchy of refined concepts that we uncover in this section. The new `Arrangement_on_surface_2` package, that replaces the former `Arrangement_2` package in an upcoming version of Cgal, is able to deal with this hierarchy of geometric-traits concepts. An illustration of the hierarchy is given in Figure 4.8.

For each refinement we present which additional functors are expected, or said in other words, we give the technical details of the various predicates that we only contoured in §4.2. We distinguish abstract and concrete refinements. A *concrete* refinement defines all specifications that are required in order to support some *specific kind on a boundary side* of the parameter space. In contrast, an *abstract* refinement constitutes a common ancestor for various concrete refinements. For a specific family of surfaces, it is possible, and often required, to combine concrete refinements to support different kinds of boundary sides; see Example 4.4 at the end of this section. Such a combined concept constitutes the minimal requirements imposed by geometric algorithms in the package that operate on arrangements for the desired family of surfaces. The hierarchical structure alleviates the production of models (for curves on such a surface) and increases the usability of the algorithms. Each refinement features a set of new expected functors. We mainly distinguish functors that give location information and functors that compute a relative order of two geometric objects.

Remark. We decided to stick with the traditional naming of variables chosen for Cgal's arrangement concepts, that is, in contrast to u and v for variables in the parameter space, we refer to x and y. In addition, we simplify the structure to be exposed next: Some refinements actually distinguish whether a certain kind of boundary appears for the x- or

y-coordinate. In such cases, we here only discuss the x-case. The analogue y-version is always supposable and should be commemorated. In addition, we simplify the technical presentation that not exactly meets the expected syntax of C++ (e.g., omitting `const`-declarations, passing parameters by reference, et cetera).

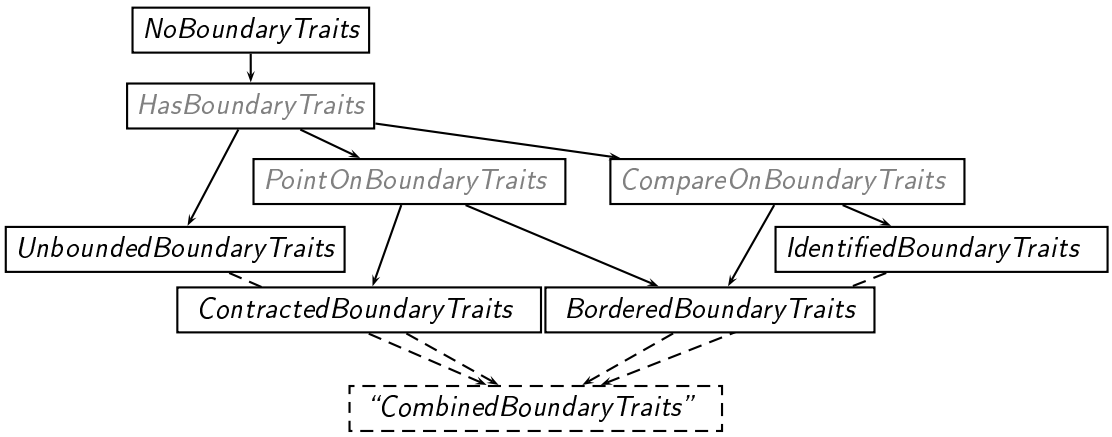


Figure 4.8. Refinement hierarchy of CGAL's *ArrangementTraits_2* concepts for surfaces. The gray concepts are abstract, that is, they only collect functors required by more than one concrete refinement. The *"CombinedBoundaryTraits"* is a placeholder for various combinations, for example, a paraboloid refines all but *BorderedBoundaryTraits*. We remark that the drawing is simplified, as we are missing actual coordinate-specific distinctions.

4.3.1. *HasBoundaryTraits*

Following Figure 4.8, the *NoBoundaryTraits* concept is refined by a single abstract concept: *HasBoundaryTraits*. It lists additional predicates required to support any curves that approach or even reach $\partial\Phi$. Before we give the expected functors, we need to generally introduce some enumerations used in the interface in addition to CGAL's `Comparison_result` which distinguishes between `SMALLER`, `EQUAL`, and `LARGER`.

```
enum Arr_curve_end
{
  ARR_MIN_END,
  ARR_MAX_END
};
```

Allows to select the minimal or maximal end of a curve.

```
enum Arr_parameter_space
{
  ARR_LEFT_BOUNDARY = 0,
  ARR_RIGHT_BOUNDARY,
  ARR_BOTTOM_BOUNDARY,
  ARR_TOP_BOUNDARY,
  ARR_INTERIOR
};
```

This enumeration categorizes the location of a curve-end or an isolated point in Φ.

The first additional functor is very basic.

- `Parameter_space_in_x_2`

  The functor is expected to provide the operator

```
Arr_parameter_space operator()(
  X_monotone_curve_2 xcv,
  Arr_curve_end ce
)
```

that returns the location of `xcv`'s curve-end defined by `ce` in parameter space in x-direction. It can return `ARR_LEFT_BOUNDARY`, `ARR_INTERIOR`, or `ARR_RIGHT_BOUNDARY`. Note that `xcv` is a (weakly) x-monotone curve whose interior lies in $\mathring{\Phi}$.

As mentioned, the similar version `Parameter_space_in_y_2` also exists.

Remark. The concept does neither mention nor specify how the locations of curve-ends are computed. However, it is encouraged to adapt `Make_x_monotone_2` such that each constructed (weakly) x-monotone curve is enhanced with these pieces of information. In fact, `Make_x_monotone_2` already has to do parts of this job, as it ensures to split curves such that there are no zero-dimensional intersections of the interior of a curve with the boundary of the parameter space. For that reason, a model of this refinement also needs knowledge about the geometry of the surface.

The next two functors provide comparisons of curve-ends near the boundary. We explicitly mention x- and y-case, as they are expected to provide operators with different signatures.

- `Compare_x_near_boundary_2`

  An instance of this functor is expected to provide two operators:

  ```
  Comparison_result operator()(
    Point_2 p,
    X_monotone_curve_2 xcv, Arr_curve_end ce
  )
  ```

  which should return the relative x-order of p's x-coordinate (in parameter space) and `xcv`'s curve-end defined by `ce` that approaches the bottom or top boundary.

  ```
  Comparison_result operator()(
    X_monotone_curve_2 xcv1, Arr_curve_end ce1,
    X_monotone_curve_2 xcv2, Arr_curve_end ce2
  )
  ```

  returns for two curve-ends approaching the bottom or top boundary the relative order of their x-coordinates (in parameter space) near the boundary.

- `Compare_y_near_boundary_2`

  The instance of this functor must provide a single operator, namely

  ```
  Comparison_result operator()
    X_monotone_curve_2 xcv1,
    X_monotone_curve_2 xcv2,
    Arr_curve_end ce
  )
  ```

  The expected output of this member is the relative y-alignment of the two curve-ends slightly to the right of the left boundary if `ce` determines their minimal ends.

Otherwise, we compare slightly to the left of the right boundary. Both curves are expected to approach (or reach) the referred boundary side, respectively.

Remark. We again mention that the comparisons functors are expected as to work in parameter space. However, a concrete implementation is not forced to compute the answer this way. There might be other (more efficient) methods to obtain the same result. We see a model that does not rely on the parameterization to give these answers in §4.6.1. The same remark propagates to other comparisons functors presented in this section.

As said, the concept is abstract, that is, a model of it does not suffice to compute an arrangement on some surface. It remains to explicitly introduce functors for different kinds of boundary side. We do so by concrete refinements.

4.3.2. *UnboundedBoundaryTraits*

The simplest next refinement is expected if a boundary side of the parameter space is tagged as unbounded. In order to fulfill the concept, the following functor is required.

- `Is_bounded_2`

  An instance of this functors should provide

  ```
  bool operator()(
     X_monotone_curve_2 xcv, Arr_curve_end ce
  )
  ```

  which returns `true` if the intended curve-end is finite, and `false` otherwise. If a curve-end is finite it is allowed to access the according point by `Construct_min_vertex_2` or `Construct_max_vertex_2`, respectively.

A model of this refinement allows to compute and maintain arrangements of curves which can be unbounded, as explained in §4.2.1.

4.3.3. *PointOnBoundaryTraits*

This abstract refinement is rather tiny, as we only expect one additional operator for an existing functor.

- `Parameter_space_in_x_2`

  must additionally provide

  ```
  Arr_parameter_space operator()(
     Point_2 p
  )
  ```

  that is, we expect to locate a (finite) point in Φ. In other words, it is possible that an isolated point exists on $\partial\Phi$, which is detected by this functor. Again, the y-version can also appear.

4.3.4. *CompareOnBoundaryTraits*

If a certain side is not labeled as unbounded, all points on that side are finite and can be accessed by the mentioned constructions. This abstract refinement introduces a functor to explicitly compare their relative order within the side. We exemplary mention

- `Compare_x_on_boundary_2`

  An instance of this functor must provide

  ```
  Comparison_result operator()(
    Point_2 p1,
    Point_2 p2
  )
  ```

  that computes the relative x-order of two points. As a precondition each point must lie on either the bottom or the top boundary.

The analogue y-version is also supposable.

4.3.5. *ContractedBoundaryTraits*

This concrete concept does not add further requirements to *PointOnBoundaryTraits*. However, we introduce it in order to explicitly distinguish the contraction case from the *BorderedBoundaryTraits*.

4.3.6. *BorderedBoundaryTraits*

As for the previous refinement, this one is artificial, that is, though concrete it is not a true refinement, as no new requirements are lists. Its intention is to constitute a concrete concept for the case that a surface comprises a bordered boundary. It refines from two abstract concepts, namely *PointOnBoundaryTraits* and *CompareOnBoundaryTraits*. We introduce it, in order to distinguish from other concrete concepts.

4.3.7. *IdentifiedBoundaryTraits*

This concept is almost similar to the previous one, but there is a significant difference: *It is not a refinement of the PointOnBoundaryTraits* In contrast to the `Parameter_space_in_x_2` for a point, it expects an additional functor whose utilization is more specific for an identification:

- `Is_on_x_identification_2`

  An instance of this functor must provide two members, namely

  ```
  bool operator()(
    Point_2 p
  )
  ```

  and

  ```
  bool operator()(
    X_monotone_curve_2 xcv
  )
  ```

Each checks whether the designated geometric object is fully contained in the left-right identification (i. e., in "x"-direction), or not. For a bottom-top identification, the y-version is also conceivable.

Note that by this design of the interface, the model is not obliged to decide whether a point or a curve lying on an identification is attained by the left or the right (bottom or top) pre-image. It just returns that the point or the curve lies on the boundary. We previously decided, how to deal with such objects internally; see §4.2.2 for more details.

4.3.8. *CombinedBoundaryTraits*

Internally, a clever dispatching of tags (we omit the technical details) allows to combine the previous concrete concepts. This enables to deduce a concept that fits for a certain family of surfaces. That is, a model for a certain surface contains a set of tags that reports which concepts it implements. The `Arrangement_on_surface_2` package uses this information to internally and automatically provide dummy implementations for the non-expected functors. This simplifies the development of a concrete model for a certain family of surfaces, as one only has to implement the functors that are really executed. The compilation is ensured by the non-called dummy implementations. In fact a quite a large number of combinations are possible; see Table 4.1.

Example 4.4 (Paraboloid). A geometric-traits class for curves embedded on the paraboloid is expected to be a model of almost all concepts that we introduced in this section. One side, for example $\partial_l\Phi$, is contracted to model the paraboloid's *apex*. Then, $\partial_r\Phi$ must be unbounded in case the paraboloid opens to infinity, or bordered, in case the paraboloid is finite. The remaining pair of opposite sides ($\partial_b\Phi$ and $\partial_t\Phi$) are identified. In the language of the herein introduced concepts, we expect the model to implement the *ContractedBoundaryTraits* for $\partial_l\Phi$, the *UnboundedBoundaryTraits* for $\partial_r\Phi$, and the *IdentifiedBoundaryTraits* for $\partial_b\Phi$ and $\partial_t\Phi$.

4.4. Maintaining a DCEL on a surface

As already mentioned in §2.4.3, CGAL uses visitors to process the topological information gathered in the course of the sweep (or the zoning) in order to construct (or modify) the DCEL that represents an arrangement of curves. That is, the *canonical output* of the sweep consists in processing events, while maintaining the event-queue and the status-line. On each combinatorial change a visitor is notified on the progress of the sweep process, for example, which event is currently handled, and which sub-curves are emerging to its left. Similar for the zone algorithm. It is the visitor's implementation that decides the actual and final output of the procedure. It varies from just reporting intersection points, or may comprise a more sophisticated task such as to construct the arrangement of the processed curves. Another variant inserts new curves into an existing arrangement, or overlays two such. More information is given in §2.4.3 and [WFZH07b]. In what comes next we mainly concentrate on the construction of an arrangement. The other applications are similar or straightforward; where needed we give additional details. A visitor that constructs the

#	Left	Right
1	Bordered	Bordered
2	Bordered	Contraction
3	Bordered	PlusInfinity
4	Identification	Identification
5	Contraction	Bordered
6	Contraction	Contraction
7	Contraction	PlusInfinity
8	MinusInfinity	Bordered
9	MinusInfinity	PlusInfinity
10	MinusInfinity	Contraction

Table 4.1. Combinations of possible conditions at $\partial_l\Phi$ and $\partial_r\Phi$. The same list can also be used for $\partial_b\Phi$ and $\partial_t\Phi$.
It is possible to encode all cases of conditions on the boundaries of $\partial\Phi$ as pair (LR, BT). For example $(1,1)$ defines a surface equivalent to a quadrangle, $(4,6)$ a surface equivalent to a sphere. The cases $(\{6,7,9,10\},4)$ are, for example, formed by elliptic quadrics that we discuss in §4.6.1. The double-identification $(4,4)$ forms genus-one surfaces, among which we discuss ring Dupin cyclides in §4.6.2. It is easy to also derive the pairs (LR, BT) for triangles, fans, half-planes, discs, and many other surfaces. However, it is unclear, whether for some combinations smooth surfaces exists, for example, $(6,6)$, $(7,7)$, or $(10,10)$.

arrangement of swept, or zoned, (weakly) x-monotone curves[27] needs to keep track the creation of new sub-curves. A new sub-curve is created whenever an intersection of more than one curve or a maximal curve-end is processed, that is, the portions of the curve(s) to the left of the event are inserted into the arrangement using one of the basic insertions procedures. We already mentioned them in §2.4.1. Each creates or updates relevant DCEL-features. The DCEL for bounded planar curves is unique and well-defined, in particular, there is only a single unbounded face.

What we like to emphasize is that the actual construction by the visitor utilizes only topological information available *during* the sweep (or zone) algorithm in order to perform the basic insertions of sub-curves — *without invoking any extra geometric information*. In contrast to perform a post-processing of the swept events, it is the on-line and interweaved fashion of the construction that is worth to mention.

We aim for a similar strategy when constructing a DCEL for an arrangement induced on a parametric surface whose parameter space may have special properties at its boundary sides; see our introduction in §4.2. Note that only special boundaries imply an elaborate handling. An empty arrangement consists of a single face and if no curve approaches or reaches the boundary, processing the curves is "isomorphic" to what we do for bounded curves in the plane. That is, *all curves lying in the interior of the parameter space can already be handled with the existing tools*. If no curves interacts with boundary, the nesting graph is supposed to be still a tree. Special diligence is only needed when curves meet the boundary of the parameter space. As a result, we scream for reusing existing

[27] Observe that we stay with CGAL's naming scheme, that is, we use x and y for the variables of the parameter space.

machinery as much as possible. *We only want to modify* CGAL*'s* `Arrangement_2` *package in its handling with respect to special boundaries.*

By now, it is completely open how to transform the various kinds of boundaries into an actual DCEL-representation, which is fine, as it turns out that various possibilities can exist. As we will see, these choices also lead to different nesting graphs.

Several operations on arrangements are quite similar in all cases. As examples we mention basic insertion and deletion procedures. It turns out that CGAL's `Arrangement_2` package already suites well to serve as a building block. We[28] extended it to CGAL's `Arrangement_on_surface_2` package which now serves as the centralized component that collects surface- and curve-independent algorithms and structures for two-dimensional arrangements on a wide range of surfaces and curves on them. The central class-template of the package has two parameters:

```
Arrangement_on_surface_2< GeometryTraits_2, TopologyTraits_2 >
```

As known, it is the `GeometryTraits_2` that provides the curve-specific components, and we have learned in §4.2 and §4.3 how to extend it in order to support curves embedded on a parametric surface with special kinds of boundaries. Remember that this class must also be aware of the geometry of the embedding surface, for example, to implement `Make_x_monotone_2`.

Similarly, all surface-specific procedures are expected from the new "external" component. We call the corresponding parameter `TopologyTraits_2`. Such a class encapsulates the topology of the surface on which the arrangement is embedded, determines the underlying DCEL representation, and supports its maintenance. It does so by defining nested types that are used in various arrangement-related operations. Additionally, it provides predicates and operations dealing with curve-ends or points related to $\partial\Phi$ that are required to consistently modify or update the DCEL. In §4.5 we present the full *ArrTopologyTraits_2* concept that an instance of type `TopologyTraits_2` must fulfill. Beforehand, we shortly review which tasks and components of the arrangement class are actually surface-dependent. This helps to clarify some design rationales of the *ArrTopologyTraits_2* concept; see also [BFWZ07].

As a first remark, we observe that the `TopologyTraits_2` parameter has replaced the `Dcel` parameter. Consequently, the new component must provide the DCEL-type. Internally, the arrangement derives the `Vertex-`, `Edge-` and `Face`-type to equip them with an interface that respects arrangement-specific goals. The actual interface of the arrangement class can be partitioned into three groups:

Traversal methods provide information about the number of DCEL-records (as cells), and the access to each valid one. We allow that a DCEL-record can be *geometrically invalid*, that is, it does not carry relevant geometric information, but only serves to encode some topological information. Such *fictitious* records should be filtered.

Basic insertions, deletions, and modifications are central operations on the DCEL; see §2.4.2. We distinguish the insertion of an isolated point and an x-monotone curve whose interior is disjoint from all existing vertices and edges of the current

[28] Central ideas by Ron Wein, Efi Fogel, Dan Halperin, and the author. Main coding by Ron Wein; significant contributions by Efi Fogel, Baruch Zukerman, and the author.

arrangement, the deletion of an edge (or of an isolated vertex), the splitting of an edge as prior operation for an insertion, and the merge of two edges as posterior operation of an edge-deletion.

Global functions are used to construct arrangements from scratch, to insert curves into an existing one, or to two overlay two instances. As learned, proper visitors can be combined with the generic `Sweep_line_2` template or the zone algorithm in order to provide these operations. The zone algorithm additionally requires a point-location strategy.

We are not going into the technical details for all of these interfaces. However, we are already able to identify surface-specific tasks expected by them. An example is the filtering of fictitious DCEL-records. Surface-specific are also specialized visitors used by the global functions that are tailored to certain DCEL-representations: They deploy additional knowledge which saves calls to geometrical and topological predicates in order to decide which basic insertion functions must be called. Only the basic insertions and deletions require elaborate modifications. We explain such when discussing the handling of connected components of a face's boundaries (CCBs). Another example is the extended support for the localization of points in the existence of a special property at a boundary side.

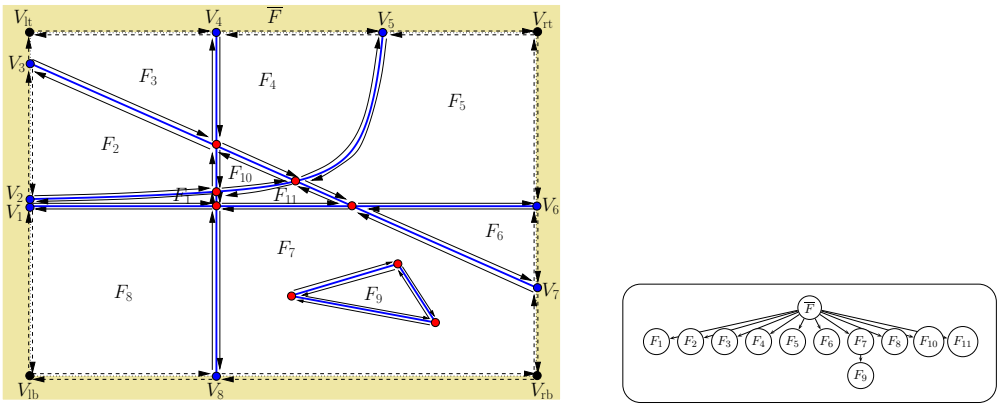
4.4.1. Choice of DCEL

An important fact is, that the generic arrangement class itself is no longer responsible to determine the actual DCEL representation for the induced subdivision. A subtask is to define how the DCEL of an empty arrangement on some S is encoded. It mainly must be decided whether the initial face is unbounded (e. g., for a plane, a paraboloid, or an open cylinder) or bounded (e. g., for a triangle, a sphere, a closed cylinder, or a torus). It is more challenging to commit to a certain representation as DCEL for the boundary of the parameter space as it is typical that several possibilities exists.

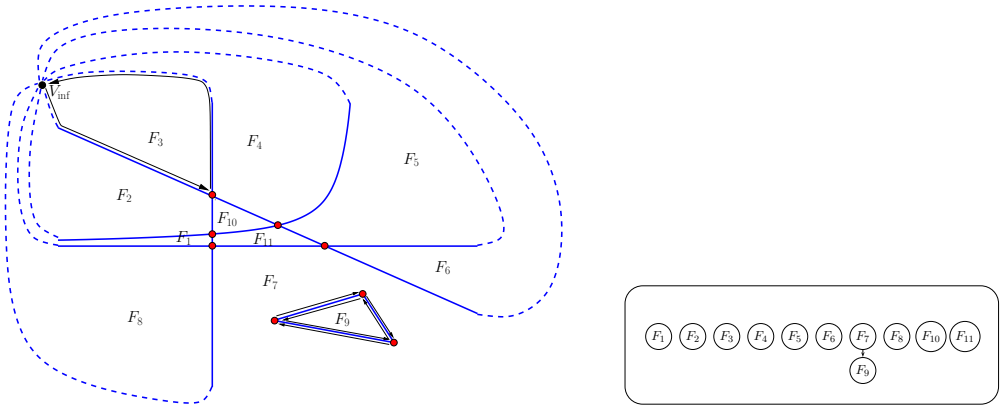
A tangible example is constituted by the unbounded plane. We aim to construct an arrangement that may contain several unbounded faces. We already chose not to clip at an explicit bounding rectangle. Instead, a possibility is to introduce an *implicit bounding rectangle* embedded in the DCEL, that is, it consists of fictitious edges. Each such edge does not represent any concrete planar curve; its sole purpose is to close the outer CCB of an unbounded face. Or vice-versa: A face is unbounded, if its outer CCB contains a fictitious edge. Actually, there is one special face that has no outer CCB, and its sole inner CCB consists of fictitious edges only. However, this face is of pure technical nature. The corners of the fictitious rectangle are given by special vertices $V_{\mathrm{bl}}, V_{\mathrm{tl}}, V_{\mathrm{br}}$, and V_{tr}. As they do not actually belong to the arrangement they must be filtered for a traversal. A curve-end that extends to infinity is represented as a fictitious vertex on this rectangle, but never coincides with one of the four corner vertices. The insertion of an unbounded curve implies a fictitious edge to split. Figure 4.9 (a) gives an illustration of such a DCEL. As it maintains a fictitious *outermost* face $\overline{F}$, the nesting graph of this DCEL is a tree.

An alternative solution, as shown in Figure 4.9 (b), is to use a single fictitious *vertex at infinity* V_{inf} and all curves extending to infinity are connected to this vertex. A face is then considered to be unbounded, if its outer CCB includes V_{inf}. For this choice, no split of a fictitious edge is required, but we need to determine the position of a new curve in the circular list of existing curves around V_{inf}. Note that there is no single outermost

Figure 4.9. Two possible DCEL-representations for the unbounded plane



(a) This DCEL uses fictitious edges (dashed) and four special vertices V_{lb}, V_{lt}, V_{rb}, V_{rt} that do not carry geometric information. The vertices $V_1, \ldots, V_8$ represent infinite curve-ends. The faces $F_1, \ldots, F_8$ are unbounded, as they are incident to a fictitious edge. The face $\overline{F}$ is fictitious without any geometric meaning. The nesting graph is a tree rooted at $\overline{F}$.



(b) This DCEL contains a single fictitious vertex V_{inf} to which all unbounded curve-ends are incident. The unbounded faces $F_1, \ldots, F_8$ are the ones that are incident to this vertex; see, for example, the indicated outer CCB of F_3. The nesting graph of this DCEL is a forest.

face: In particular, each unbounded face is a root in the nesting graph, and also a bounded face that is not a hole[29] of a another bounded face constitutes a root. That is, we obtain several *equitable outermost* faces. Following the nesting graph is a forest. A root grows to a tree, if it has at least one two-dimensional hole in it, such as F_7.

Both representation are useful and legitimate, and none can be preferred over the other. Actually, each can be more suitable than the other in different situations. In fact, even mixed cases are conceivable, for example, we can have two fictitious vertices for curves extending to $x = \pm\infty$ and sequences of fictitious halfedges connecting (at infinity) curves that extend to $y = \pm\infty$. The relevance of such a representation is questionable, but we do not judge on this. For CGAL 3.3, we decided to represent unbounded planar arrangements with the implicit bounding rectangle. Other representations might be included in future releases.

We next generalize the topological tasks beyond the unbounded plane, similar to the generalization of the geometrical tasks. We basically have two strategies to represent arrangements on parametric surfaces as DCEL.

Tree-strategy This strategy requires to agree upon a *single outermost* face. This is typically done by choosing a *reference point* that is expected to be contained in this closed face (i. e., its interior and its inner CCBs). We have to ensure that the creation of new faces, and in particular the assignment of CCBs that pop up, aim for a tree rooted at this outermost face. Below, we identify the tasks how to support this decision in order to maintain a tree.

Forest-strategy In this strategy, *several faces can be outermost*, that is, they are *equitable*. Making faces equitable means to separate them by outer CCBs. For this strategy it must be clear what outermost means for a specific surface. Once this is fixed, any operation that requires an update of CCBs (e. g., the creation of a new face) has to follow the chosen definition. The tasks we identify below help to implement the chosen definition for an *outermost* face.

Remarks.

- In both strategies, the first root of the nesting graph has no outer CCB.
- Note that already in the bounded plane, we have some kind of *equitable* faces; see, for example, faces F_2 and F_3 in Figure 2.6. However, they are surrounded by a common inner CCB; see in the example E_2 which determines that F_2 and F_3 are children of F_1. Equitable means that none makes the other locally non-simply connected; see also Definition 2.41. Thus, they are separated by outer CCBs from each other and none is a root, in contrast to the forest-strategy that already expects equitable roots.

We admit, that the strategies seems 1 abstract. On the other hand, the chosen strategy has implications on the nesting graph. That is, *by choosing a strategy, we are actually asking for a consistent way of assigning CCBs to the lists of inner and outer CCBs of faces. These assignments eventually define the nesting graph.* Thus, we concentrate on this classification when discussing CCBs below. There, we also extend our consideration to surfaces with identifications. More technical details are given in §4.5.5.

[29] A hole makes a face non-simply connected; see Definition 2.41.

4.4.2. Boundary tasks

In addition to the DCEL-decisions, we already have detected some surface-specific topological tasks:

- Remember that visitors (for sweep or zone) call the basic insertion functions to modify the DCEL with respect to the insertion of a curve c. There are special cases that the arrangement has to take care of. An example is that some curve-end of a curve c can coincide with an isolated vertex in a face, so the insertion is actually *from a vertex*. This is already a solved problem for the bounded plane. However, in our case the arrangement deals with boundaries of the parameter space. But it has no chance to decide itself how to insert the relevant curve-end. Remember the two ways (fictitious rectangle, fictitious vertex at infinity) to represent an unbounded arrangement as DCEL. In both cases, the insertion is actually *from a vertex at infinity*. Similar cases are conceivable for other topologies. Thus, our solution to this problem consists in the arrangement's query of the attached topology-traits class. It returns a `CGAL::Object` comprising *one* of the following types:
  - A handle for a fictitious halfedge, which means that the queried curve-end splits the designated fictitious halfedge in its interior. The split-point becomes the (fictitious) vertex representing the curve-end.
  - A handle for a vertex to which the curve-end is incident.
  - An empty object, which implies that it is required to create a new vertex representing the curve-end. The curve itself is the sole incident curve to the vertex that will be created.

  If only one curve is incident to a vertex, the insertion *from a vertex* is simple, otherwise, we refer to the next task.

  Remark (Isolated point). Remember that some topologies allow isolated points on $\partial\Phi$. Thus, the topology-traits class must also be able to compute the same piece of information for such a point, instead of a curve-end.

- Find the position of a curve incident to a DCEL-vertex V on $\partial\Phi$ in the circular order of curves around V. This holds for both fictitious and non-fictitious vertices.
- Split a fictitious edge. The counterpart of this operation consists of the detection and removal of a redundant vertex on the boundary.

We refer to §4.5 where we explain how these tasks are technically interfaced.

4.4.3. Faces

Note that faces of the subdivision (i.e., open connected point sets on the surface) are stored implicitly, that is, not special geometric object is deposit in the DCEL. However, part of the implicit representation is the correct assignment of connected components of the face's boundaries (CCBs). Each insertion or deletion of a curve can also imply a modification of a face's CCBs. In particular, a face can be split into two faces. For the different kinds of boundaries, we have to consider specific possibilities.

Unbounded faces If an unbounded face is split by a bounded curve, it must be decided by the topology-traits class which of the two resulting faces is unbounded.

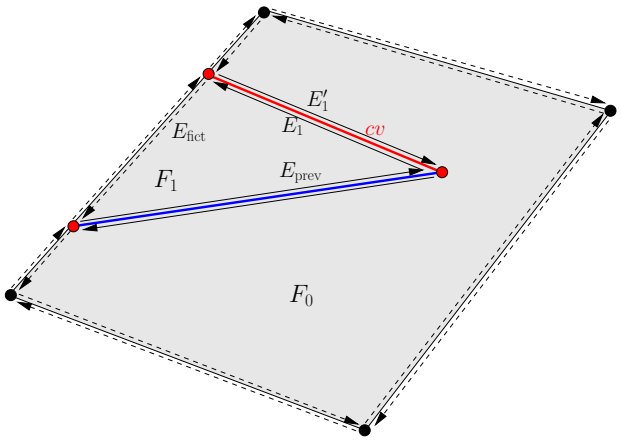


Figure 4.10. We consider an arrangement of line segments in a finite rectangle whose boundary is modelled with fictitious edges. The insertion of cv results in a split of F_0. The new face F_1 has an outer CCB that is formed by E_{prev}, E_1, and E_{fict}.

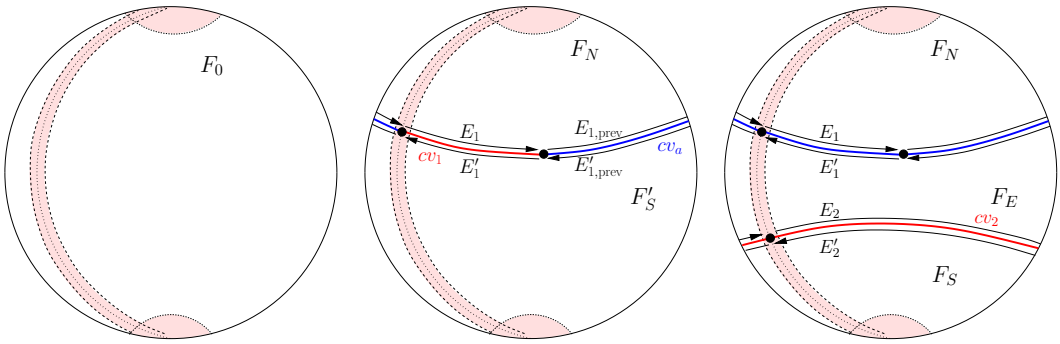
Bordered edges Consider a finite rectangle, which constitutes a compact surface whose boundary consist of four bordered sides. We can represent these bordered sides with the help of fictitious halfedges (as for the unbounded rectangle). In such a case, it is possible that the insertion of a bounded curve *from a single vertex* splits a face, such that one does not make the other locally non-simply connected; see the illustration in Figure 4.10. It is the topology-traits class that takes care of this decision.

CCBs, roots of the nesting graph — and identifications Remember the tree- and the forest-strategy that we only introduced abstractly. We next discuss examples for them on surfaces with a curve of identification. This helps us to detect the tasks that we require from the topology-traits class for any kind of parametric surface we want to consider.

In the tree-strategy, the definition of the root face is simple. It is defined by picking a reference point. On a sphere, we can choose, for example, the north pole. The following example is also illustrated in Figure 4.11: The initial DCEL consists of a single bounded face F_0. It does not have any outer CCB. This contrasts with the planar case, where each bounded face has an outer boundary. Next consider that we close the *tropic of Cancer* (*northern turning circle*) and the *tropic of Capricorn* (*southern turning circle*). For the first curve, the initial face is split into two. The face F_N now contains the north pole, that is, the reference point. Thus, according to our strategy, it should obtain an inner CCB (represented by E_1), that separates F'_S (containing the south pole and the equator) from F_N. This ensures that F'_S becomes a child of F_N. F'_S itself gets a single outer CCB (represented by E'_1). After adding the second tropic, there are now three faces F_N, F_E, and F_S. The latter two originate from the split of F'_S. Note that F_S gets a single outer CCB (represented by E'_2) and is separated from F_E corresponding to the strategy by an inner CCB (represented by E_2) of F_E. Observe that we come to *two decisions*: Make E'_1 the outer CCB of F'_S (and not E_1) and make E'_2 the outer CCB of F_S (and not E_2). A respective twin defines an inner CCB for the proper originating face.

Similarly, we can pick a "reference point" on a cylinder C, even if it is unbounded. For example, we choose as reference some point on C with $z = +\infty$. This case is illustrated

Figure 4.11. The tree-strategy on a sphere



(a) Single bounded face F_0 with neither outer CCB nor inner CCB.

(b) Closing the northern tropic by cv_1 results in two faces: F_N gets a face separated by an inner CCB that is defined by E_1. Face F'_S gets a single outer CCB defined by E'_1.

(c) Adding the southern tropic cv_2 results in another split: F_E inherits the outer CCB defined by E'_0, while the new face F_S is separated from it by an inner CCB defined by E_2. The split face F_S gets a single outer CCB defined by E'_2.

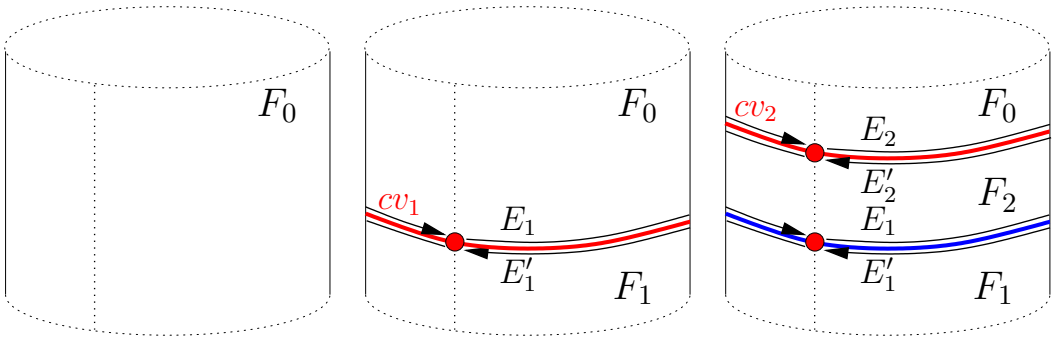
in Figure 4.12. Mind again, that two decisions help to define the the final DCEL: Make E'_1 the outer CCB of F_1 (and not E_1) and (again) make E'_2 the outer CCB of F_2 (and not E_2).

Remarks (on the tree-strategy).

- The tree rooted at a reference face in ensured by defining *which CCB becomes outer of a newly created face*. Let us keep this task in mind.
- Note that F_1 in Figure 4.12 is an *unbounded leaf* in the nesting graph. This may be not very intuitive, but remember that this is due to the fact that the tree-strategy simply *defines* an outermost face.

To avoid such unbounded leaves, we actually encourage to apply the forest-strategy in the case of a cylinder. We again start with an example, illustrated in Figure 4.13: There is the single unbounded face F_0. When adding cv_1 the face F_0 splits into the faces F_0 and F_1. As both are unbounded we do not want to make one nested below the other. Thus, we decide to make them *equitable* on the cylinder. Consequently, each gets its own outer CCB: E'_1 defines the one for F_0, while E_1 defines the outer CCB for F_1. We next insert cv_2. This separates F_2 from F_0. Again, we do make F_2 equitable to the other, that is, F_2 becomes a root. This time, the reason is that F_2 should be nested below F_0 and below F_2 at the same time. However, this would lead to a nesting graph, that is not a forest. We do not want to exclude this possibility in general, but it is less intuitive, *that a*

Figure 4.12. The tree-strategy on a cylinder



(a) Single unbounded face F_0 with neither outer CCB nor inner CCB.

(b) Adding cv_1 results in two faces. F_0 gets separated from F_1 by saying that E_1 defines an inner CCB of F_0. That is, by strategy, F_1 is a child of F_0 in the nesting graph.

(c) Adding cv_2 now splits F_0 again. Similarly, F_0 gets separated from F_2 by saying that E_2 defines an inner CCB of F_0. We make F_2 a child of F_0. In parallel F_1 becomes a child of F_2 as E_1 stays an inner CCB, but now for F_2.

set of points should be "somehow" a subset of two disjoint sets.[30] Thus, to make F_2 a root face is a nice and sensible solution. However, it is now surrounded by two outer CCBs.[31] While E'_1 is already determined to be one of them, it must be *decided* that E_2 defines the second (and not E'_2). Following, E'_2 automatically defines an outer CCB for F_0.

Remarks (on the forest-strategy).

- We require a definition that specifies the properties of a root. In the example, we choose "unboundedness" and "not a unique nesting". However, these conditions are not precise.
- Once root faces are decided, we need a test that determines whether a newly created outer CCB belongs to the same root face as another (fixed) outer CCB. Let us also keep this task in mind.

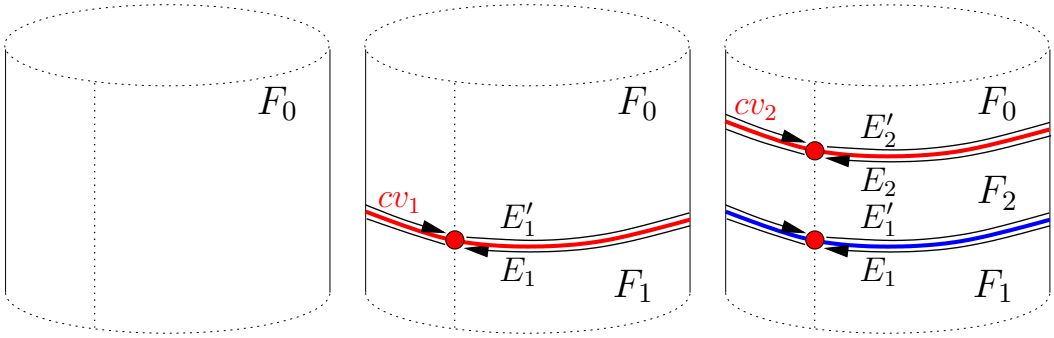
We remark that the forest-strategy also makes sense for bounded surfaces, for example, as defining a reference point might not reflect the user's wish. In particular, he maybe wants to avoid an artificial hierarchy of faces. Using the forest-strategy is a way out of this dilemma. As example, we mention the rectangle as in Figure 4.10, or we refer to Figure 4.14

Note that with these examples in mind, it makes sense to extend the DCEL: In addition

[30] Note that a face is supposed to represent a connected subset, and all faces of a DCEL model a disjoint decomposition of the input surface.

[31] The reason is that neither E'_1 nor E_2 can define an inner CCB, as this would model that F_2 is nested below another face.

Figure 4.13. The forest-strategy on a cylinder

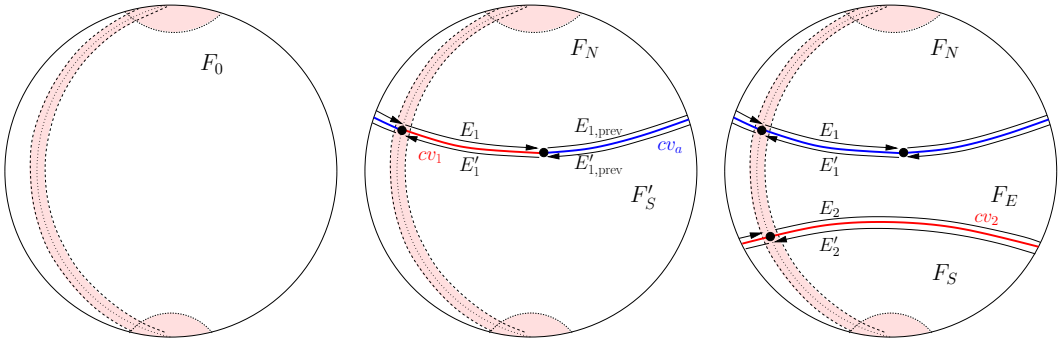


(a) Single unbounded face F_0 with neither outer CCB nor inner CCB.

(b) Adding cv_1 results in two faces: Both have a single outer CCB: E'_1 for F_0 and E_1 for F_1.

(c) Adding cv_2 splits F_0 again. F_0 still has a single outer CCB defined by E'_2. We determine that E_2 forms the second outer CCB of F_2, besides the one defined by E'_1. Thus, E'_2 defines an outer CCB for F_0.

Figure 4.14. The forest-strategy on a sphere



(a) Single bounded face F_0 with neither outer CCB nor inner CCB.

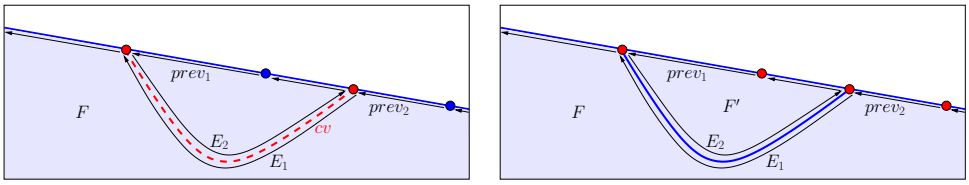
(b) Closing the northern tropic by cv_1 results in two faces: F_N and F'_S are equitable. Each gets its own outer CCBs. E_1 defines the one for F_N and E'_1 the one for F'_S.

(c) Adding the southern tropic cv_2 splits F'_S into F_E and F_S. F_S's outer CCB is defined by E'_2, F_E gets two outer CCBs. It must be determined that E'_1 is the second besides E_2. F_N then gets the outer CCB defined by E_1.

to maintaining a list of inner CCBs, we are now confronted with cases, where a face can have more than one outer CCBs. Thus, we require a DCEL-class that is able to store a list containing more than one outer CCB for a face. Note that each list of CCBs can also be empty.

We admit that our examples are chosen carefully to show which surface-specific topological tasks must be handled. The examples for the forest-strategy are even "restricted". That is, each of their faces is a root. However, there exist faces on such surfaces that are surely none-roots. In order to precisely define what makes a face a root, we have to reconsider the basic insertion of a curve at two vertices into a face F. Among the basic functions modifying the DCEL, this is the only one that can construct a new face; see Figure 2.9. We know that F models a connected set of points on an orientable surface whose boundaries are described by the given CCBs of F. Each CCB forms a cycle and the intended face is to the left of the oriented edges contained in theses cycles. When inserting a curve cv at two (non-isolated) vertices, we are given two predecessor edges, each lying on some CCB of F. Remember that the interior of cv must completely lie in the face F. Following, both CCBs belong the F. Upon this insertion, we remove a one-dimensional set of points from the face by adding edges for cv. These edges get somehow connected in between the predecessor edges and their successors. It results in either merging two CCBs into one, or we get two individual CCBs. We basically have to deal with 3 different combinations:

Figure 4.15. Inserting a curve at an outer CCB



(a) Curve cv is added at two vertices. Its predecessor edges lie on the same outer CCB.

(b) F' is split from F, but it does not make F locally non-simply connected. Thus, $prev_2$ still defines F's outer CCB (with E_1), while $prev_1$ (with E_2) defines the new face F''s outer CCB.

Both predecessor edges belong to the same outer CCB: We can assume that the CCBs of F only consist of this single outer CCB. Thus, the face is two-dimensional and it looks in the neighborhood of cv like an open half-plane; see Figure 4.15 (a). Following, cv separates a two-dimensional set F' from F. Even more: F' cannot make F locally non-simply connected. Thus, F' models a new face and we obtain two individual CCBs. One becomes the new outer CCB of F', while the other stays the outer CCB of F. That is, F' is equitable to F; see Figure 4.15 (b). We call this case an *outer split*.

In the nesting graph, the node for F gets replaced by two nodes: One for F' and one for the remaining part of F. If F was a child of some $\hat{F}$, then F' becomes a child of

$\hat{F}$ as well; consider, for example, faces F_2 and F_3 in Figure 2.6. More important is: If F was a root, then F' becomes a root face as well.

The predecessor edges belong to different CCBs There exist several combinations and all have in common that the insertion of cv adds edges that merges the two involved CCBs into a single CCB. That is, we merge two boundaries of a face. This keeps the face connected. If one of the originating CCBs was outer, the merged CCB also becomes outer. In case that both were inner, the merged also constitutes an inner CCB. As no new face is constructed, these cases are simple and of no relevance for our further objectives. Especially not with respect to changes on the nesting graph upon face creations in the tree- or forest-strategy.

Both predecessor edges belong to the same inner CCB: This case requires a more elaborate study. First of all, observe that the existing inner CCB separates a set of points that is considered to be a child of F in the nesting graph. This set is either two- or one-dimensional. If it is two-dimensional the insertion is analog to the outer split: Simply replace "outer" with "inner". The inner CCB gets rerouted, while a new outer CCB appears that separates the split set of points F'. The difference to the outer split is that F cannot be a root, and so F'.

If the insertion of cv closes a one-dimensional set to a one-dimensional non-simply connected loop, three possibilities exist. They are depending on the involved curves and mainly on the surface that embeds the curves:

(1) F gets split into two disjoint two-dimensional sets, such that one makes the other locally non-simply connected
(2) F gets split into two disjoint two-dimensional sets, where one does not make the other locally non-simply connected
(3) the loop of curves describes a one-dimensional subset of F, but does not make F locally non-simply connected

The examples in Figures 4.9, 4.11, 4.14, 4.12, and 4.13 show different situation where we have to distinguish between case (1) and case (2). Case (3) is more special, as it only occurs on surfaces with two identifications. Below we give further details on this case; a concrete example is given in Figure 4.25 that is included in §4.6.2, where we discuss a family of surfaces whose parameterization comprises two identifications. Note that the "inner split" that we described first can be seen as a variant of case (2). Actually, the reduction can be established by contracting the two-dimensional set to a one-dimensional.

Summing up, we detect that in most cases, the required modifications of the DCEL by the basic insertion of a curve at two vertices are straightforward — except for the insertion at two vertices that connect a common inner CCB. For this situation, we have identified three different cases, that must be distinguished with the help of the topology-traits class. Thus, we next concentrate on this task.

Notice that the former inner CCB splits into two CCBs, and we have to decide what happens with them; compare also with the examples presented in this section. There are basically four options, and we shortly see that the chosen strategy has implications on which option gets launched.

(a) Create a new face F' and nest it below F, that is, assign one of the resulting CCBs to the list of inner CCBs of F, while the other becomes the outer CCB of F'.
(b) Create a new face F' and make it equitable to F, that is, one CCB becomes outer for F', while the other CCB must be added to the list of outer CCBs of F.
(c) No new face is created and the two resulting CCBs become inner for F, that is, there is a one-dimensional "hole" surrounded by *two* inner CCBs.
(d) No new face is created and the two resulting CCBs become outer for F, that is, F is now surrounded by *two* outer CCBs.

It is obvious that options (a) and (b) must correspond to case (1) and (2), while options (c) and (d) are related to case (3). The topology-traits implements either the tree- or the forest-strategy. The strategy guides the basic insertion in the following way: For case (1) both strategies chose option (a); this is straightforward. In contrast to case (2), where the tree-strategy has again to trigger option (a), while the forest-strategy chooses option (b). For the special case (3), option (c) is the choice for the tree-strategy, as this ensures that there is always an outermost face not having an outer CCB.[32] This face is supposed to constitute the root of the tree. However, there can be an *innermost* face, that is surrounded by two outer CCBs. As this face would be nested below two other faces, we do not encourage the tree-strategy for a surface with two identifications. For such, we recommended the forest-strategy that decides for option (d) in case (3). This ensures that further splits of F result in faces that are equitable, and thus can model different roots.

Observe that options (a) and (b) still need some more guidance from the topology-traits class, as seen in the examples. In option (a), it is unclear which of the two CCBs becomes outer for a newly created face. This must be decided surface-specifically. Actually, it is advantageous to know for a loop of curves attached to a CCB whether it is contractible to a point on S. Then, the answer can be derived as for bounded planar curves:

- Determine the direction of the predecessor halfedge at the lexicographical[33] smaller vertex.
- If it is from left to right[34] this halfedge defines the outer CCB of the new face.
- Otherwise, the predecessor halfedge at the other vertex defines the outer CCB of the new face.

Following, the open question is only of substantial nature if a loop of curves on S *cannot* be contracted to a point. By how we parameterize surfaces, this is only possible if at least one curve of identification exists; all other surfaces are homeomorphic to a disc. For surfaces with identification the problem is more elaborate. The open question in option (b) is: Which of the two CCBs gets assigned to the list of F's outer CCBs? We shortly give further details on realizing these "CCB-tasks" for surfaces with identifications.

Let us reconsider roots of the nesting graph. If the topology-traits class implements the tree-strategy, we never create a new root: The initial root has no outer CCB, so no outer split can happen. In addition, inner splits do not create new roots, and finally, option (b) is never triggered, which constitutes the remaining possibility to create a new root.

In contrast, the forest-strategy creates new roots. Note that the first new one must be triggered by option (b), as the initial face has no outer CCB. Further roots can appear upon outer splits and constructions by option (b).

[32] Observe that otherwise the initial face is the candidate to get an outer CCB.
[33] Given in parameter space
[34] Again in parameter space

Definition 4.5 (Root).

Tree-strategy: A root is a face without an outer CCB. Note that there is only one.

Forest-strategy: A face that makes another face locally non-simply connected cannot be a root. Each other face is a root.

This concludes the discussion on face creations and what to do with CCBs. The full technical interface is given in §4.5.5. Observe that by this abstraction the full control on the assignments of the CCBs is given to the topology-traits class. Thus, it constitutes the entity that decides which strategy is implemented for its surface, and following whether the nesting graph is a tree or a forest. This way, we conserve the possibility to represent the subdivision of a surface with different strategies — depending on the user's preferences.

Remark (Relocation of holes). Remember that the split of a face implies some queries: Namely, we have to check for each isolated vertex and for each inner component in the original face, whether it should be moved to the newly created split face. This task boils down to determine the lexicographical (always finite) minimal point of such an object and to let the topology-traits class check whether it is contained in a newly constructed face. We have to incorporate the topology-traits class here, as the special boundaries, in particular identified ones, do not allow to derive a surface-independent strategy. Note that this also has implications on the nesting graph.

Remark (Removal). The arrangement also demands for basic removal functions. Among them, it is the deletion of an edge that demands in some cases help from the topology-traits class that provides surface-specific answer. The key question for this task is, whether the deletion of a pair of twin halfedges, each lying on an outer CCB, cause the creation of a new inner component; otherwise two incident faces should be merged.

We refer to §4.5.5 where we give technical details and the interface for all required tasks of a topology-traits class. That is, we present CGAL's new *ArrTopologyTraits_2* concept. Concrete examples of models are then discussed in §4.6. We illustrate details on the implementation for two families of curved surfaces with identifications. Both models exploit a technique that we present next.

Realizing a model for surfaces with identifications We previously identified in a high-level description which tasks a model of the topology-traits class has to provide with respect to faces and their CCBs. Several models exists in CGAL, out of which we discuss two concrete examples for surfaces with identifications in §4.6.1 and §4.6.2. To simplify their presentation, we already recapitulate the tasks and give tools to realize each.

The tree-strategy expects the following decisions:

- How to detect case (3)?
- How to decide which CCB out of two gets outer for a new face in option (a)?

The forest-strategy expects an enhanced set of decisions:

- How to distinguish between case (1), (2), and (3)?
- How to decide which CCB out of two gets outer for a new face in option (a)?
- How to decide which CCB out of two gets also outer for a new face in option (b)?

Note that the tasks for the tree-strategy are a "subset" of the tasks for the forest-strategy.

Surface has one curve of identification: Before we really turn to such surfaces, think of any loop in a surface that is homeomorphic to an (open or closed) disc. As the surface is simply connected such a loop is contractible to a single point. In general, this does not hold for a loop on a surface with a curve of identification. In particular, *when also respecting possible contraction points.* If we remove such points, each surface with a single curve of identification is homeomorphic to an open or closed cylinder. In what follows we assume w.l.o.g. that this cylinder's parameterization comprises a left-right identification.

We can distinguish two kinds of loops: Loops that are contractible to a point, and loops that are not.[35] Let us have a closer look at properties of such loops: Assume that a loop L does not cross the curve of identification. Then, it is contractible to a single point, as the image of the parameter space's interior is, by precondition, bijective to an open disc. Moreover, consider a local continuous transformation of a loop's non-cyclic subpath such that this part now crosses (not touches) the curve of identification twice. As the surface is orientable a local map exists that supports this transformation. Vice versa, we can conclude that every loop that crosses a curve of identification $2n$ times is contractible to a point, by the "reversed" transformation. Now consider a loop that has exactly one crossing with the curve of identification. It is easy to see, that there is no cover of maps homeomorphic to open discs such that the loop can be contracted to a single point in their union. Thus, such a loop is non-contractible. By the same argument as in the *even* case, we can locally transform a non-cyclic subpath of the loop to cross a curve of identification $2n+1$-times. Still, it is non-contractible.[36]

Definition 4.6 (Perimetric loop, CCB, and face). Let S be a parametric surface with an identification excluding possible contraction points, and L be a loop on it. We say that L is *perimetric* if it is non-contractible to a point. This property is equivalent to L having an odd number of crossing with the curve of identification on S. A CCB is called perimetric, if the attached curves form a perimetric loop on S. We call a face F on S *perimetric* if it has a perimetric CCB.

Example 4.7 (Perimetric loop). Examples of perimetric loops are curves (cv_1, cv_a) in Figure 4.11 (b), curve (cv_2) in Figure 4.11 (c) (and each also in Figure 4.14) curve (cv_1) in Figure 4.12 (b), curve (cv_2) in Figure 4.12 (c) (and each also in Figure 4.13), curves (cv_a, cv_1, cv_b) in Figure 4.25 (a), and curves (cv_c, cv_2, cv_d) in Figure 4.25 (b).

Definition 4.8 (Directed loop). A *directed loop* $\overrightarrow{L}$ is a sorted sequence of curves $(cv_0, \ldots, cv_k)$ that are traversed in a specified common direction: Let $\overrightarrow{\max}(cv_i)$ be the maximal curve-end of cv_i in the order of the traversal, and let $\overrightarrow{\min}(cv_i)$ be cv_i's corresponding minimal curve-end. It holds $\forall 0 \leq i \leq k : \overrightarrow{\max}(cv_i) = \overrightarrow{\min}(cv_{(i+1) \mod k}) =: p_i$. We call p_i th *i-the connection point.*

[35] The two sets are identical to the homotopy groups of the cylinder.

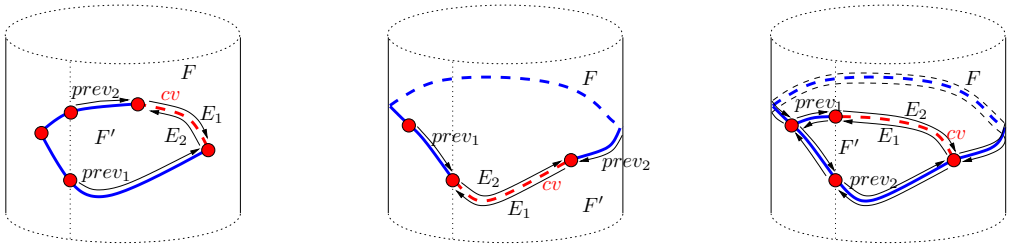
[36] Touching intersections and crossings in the corner can be "removed" by symbolically perturbing the curve of identification: That is, if moving the curve of identification, a touching intersecting either vanishes or crosses it twice in opposite directions. Following we can ignore it. Crossings in the corners can be handled by moving them on $\partial\Phi$ in clockwise direction symbolically by a tiny amount. Note that this is already reflected by assigning a curve-end uniquely to one of the four boundary sides.

We consider two sources of directed loops:

- The insertion of a curve cv at two vertices that short-cuts an inner CCB with given predecessor edges $prev_1$ and $prev_2$ defines two directed loops $\overrightarrow{L}_{prev_1} = (cv_0^1, \ldots, cv_{k_1-1}^1, cv)$ and $\overrightarrow{L}_{prev_2} = (cv_0^2, \ldots, cv_{k_2-1}^2, cv)$. The curves cv_i^1 are those attached to the edge-range [`prev2->next()`, `prev1`] (using `->next()`). The curves cv_i^2 are those attached to the edge-range [`prev1->next()`, `prev2`] (using `->next()`). Note that the two directed loops traverse cv in opposite directions.
- A CCB defined by an edge E specifies a directed loop of curves $\overrightarrow{L}_E$. The direction of E determines the direction of the curves' traversal.

An important property is that no interior of a curve being part of a directed loop intersects with the curve of identification. The reason is, that `Make_x_monotone_2` splits curves at such intersection. Following, these intersections only take place at the connection points of a directed loop.

Figure 4.16. Insertions on a surface with an identification



(a) Case (1): Adding cv splits F' from F by two non-perimetric loops.

(b) Case (2): Adding cv splits F' from F by two perimetric loops.

(c) Case (2): Adding cv splits F' from F by a perimetric and a non-perimetric loop.

We next show that directed loops constitute a decisive tool which help to distinguish between case (1) and case (2). Consider the directed loops $\overrightarrow{L}_{prev_1}$ and $\overrightarrow{L}_{prev_2}$ emerging upon the insertion of cv on the "cylinder" (i. e., S without possible contraction points). We have two possibilities:

- Each loop is contractible to a point, that is, non-perimetric. Then, they define a two-dimensional subface F' from F that makes F non-simply connected. Following, we are in case (1); see Figure 4.16 (a).
- At least one of the loops is perimetric. If exactly one is perimetric, the other separates a subset F' that does not make F locally non-simply connected. This situation is similar to an inner split; see Figure 4.16 (c). However, we do not know which one is perimetric, and in addition, both can be perimetric. This case also results in a separation of some subset F'. But this time, it is along the whole loop; an example is given in Figure 4.16 (b). However, both situations result in case (2).

Definition 4.9 (Sign of a directed loop). Let $\overrightarrow{L} = (cv_0, \ldots, cv_k)$ be a directed loop with connection points p_i, and cv_i^Φ the pre-image of cv_i in the parameter space Φ of S. Remember that we assume $\partial_r\Phi$ and $\partial_l\Phi$ to be identified. The other case is symmetric.

The *sign of* $\overrightarrow{L}$ *at* p_i is given by

$$\operatorname{sign}(\overrightarrow{L}, i) := \left\{ \begin{array}{rl} +1 & \text{if } \overrightarrow{\max}(cv_i^\Phi) \in \partial_r\Phi \wedge \overrightarrow{\min}(cv_{(i+1) \mod k}^\Phi) \in \partial_l\Phi \\ -1 & \text{if } \overrightarrow{\max}(cv_i^\Phi) \in \partial_l\Phi \wedge \overrightarrow{\min}(cv_{(i+1) \mod k}^\Phi) \in \partial_r\Phi \\ 0 & \text{in all other cases} \end{array} \right\}$$

More intuitively, the sign of a directed loop at connection point p_i is $+1$ if the pre-image of the loop approaches the right boundary of the parameter space, crosses the left-right identification, and continuous emanating from the left boundary; the analogy is similar for the negative case.[37]

The sign of a directed loop is simply the sum of the signs:

$$\operatorname{sign}(\overrightarrow{L}) = \sum_{i=0}^{k} \operatorname{sign}(\overrightarrow{L}, i)$$

Observe that a loop with sign zero corresponds to an even number of crossing with the identification, that is, this loop is non-perimetric. In contrast, a non-zero sign implies its perimetricy. By how we defined the sign of a directed loop, we also obtain some geometric interpretation with respect to the corners of the parameter space:

Definition 4.10 (Orientation). Let $\overrightarrow{L} = (cv_0, \ldots, cv_k)$ be a directed loop with connection points p_i and $\operatorname{sign}(\overrightarrow{L}) \neq 0$. That is, $\overrightarrow{L}$ is perimetric. Denote with cv_i^Φ the pre-image of cv_i in the parameter space Φ of S. Let w be a corner of the parameter space Φ.

We say that $\overrightarrow{L}$ *turns to* w if there is a cv_i with the following conditions:

- $p_i^\Phi := \overrightarrow{\max}(cv_i^\Phi) \in \partial\Phi$.
- When traversing $\partial\Phi$ in counter-clockwise order starting in p_i^Φ *we meet* w before hitting any other p_j^Φ.

Otherwise, we say that $\overrightarrow{L}$ *abandons from* w.

Combining Definitions 4.9 and 4.10 we get the following:

Corollary 4.11. *A directed loop* $\overrightarrow{L}$ *with* $\operatorname{sign}(\overrightarrow{L}) = 1$ *turns to* $w_{\max} = (u_{\max}, v_{\max})$ *and abandons from* $w_{\min} = (u_{\min}, v_{\min})$. *If* $\operatorname{sign}(\overrightarrow{L}) = -1$, *then it turns to* $w_{\min}$ *and abandons from* $w_{\max}$.

The corollary's proof is by constructing the different cases. For an example, see Figure 4.17 (c): L_2 in the specified direction has positive sign and thus turns to $w_{\max}$

[37] In case that some p_i^Φ is identical to a corner of the parameter space, we again consider a consistent symbolic perturbation in clockwise direction along $\partial\Phi$.

and abandons from $w_{\min}$. Following, the area on S to the left of the directed loop with positive sign must comprise $\varphi(w_{\max})$, while $\varphi(w_{\min})$ is definitely not contained in this area. The negative case is analog. The reason is that a perimetric loop on a cylinder is separating, that is, it splits the "cylinder" S into two disjoint sets.

We are left with the assignments of CCBs. If both $\overrightarrow{L}_{prev_1}$ and $\overrightarrow{L}_{prev_1}$ are non-perimetric, the answer which predecessor edge defines the outer CCB can be determined by the direction of the edge whose target is the leftmost curve-end of cv. In the example of Figure 4.16 (a) $prev_1$ is this edge and it is directed from left to right. Thus, it defines the outer CCB of the new face F'.

In the other cases, we rely on $s_{prev_1} := \text{sign}(\overrightarrow{L}_{prev_1})$ and $s_{prev_2} := \text{sign}(\overrightarrow{L}_{prev_2})$. Two possibilities exist for option (a); see also Figure 4.16 (b) and (c).

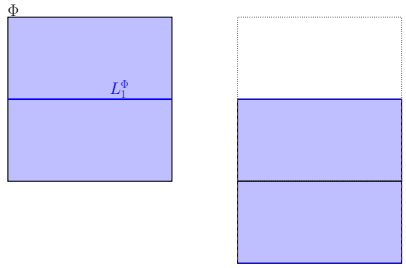
- If $s_{prev_1} = 0$, then $prev_1$ defines the outer CCB of the new F'. If $s_{prev_2} = 0$, then $prev_2$ defines the outer CCB of the new F'.
- Otherwise $s_{prev_1} \neq 0$ and $s_{prev_2} \neq 0$. In addition, it must hold that $s_{prev_1} \neq s_{prev_2}$. Thus, we only consider s_{prev_1}. For the tree-strategy, we have to ensure that the nesting tree with respect to the reference point is ensured. The CCB defined by some edge $prev_1$ is outer for the new face F' if $\overrightarrow{L}_{prev_1}$ abandons from the reference point. We can sensibly assume w.l.o.g. that the reference point is identical to $w_{\max}$. Following, $prev_1$ defines the outer CCB of F' if $s_{prev_1} = -1$. In some cases, we may want to choose $w_{\min}$ as reference point. If so, $prev_1$ defines the outer CCB of F' if $s_{prev_1} = 1$. For the forest-strategy, this test is only involved if F originally has no outer CCB. But as we make F and F' equitable on S, we can let any of $prev_1$ or $prev_2$ be defining for the outer CCB of F'.

If we aim for the forest-strategy and F originally has some perimetric outer CCB defined by some edge E_0, then F' is split from F in the neighborhood of this CCB. Thus, E_0 defines the first outer CCB of F'. However, as F' is separated equitable from F it demands for a second one. It will be one of the edges E_1 (succeeding $prev2$) or E_2 (succeeding $prev1$) we added for cv. Note that E_1 and E_2 define outer CCBs by the forest-strategy. Both CCBs are perimetric and it holds $0 \neq s_{E_1} := \text{sign}(\overrightarrow{L}_{E_1})$, $0 \neq s_{E_2} := \text{sign}(\overrightarrow{L}_{E_2})$, and also $s_{E_1} \neq s_{E_2}$. The test which of the two forms the desired second CCB can also be realized in terms of these signs:

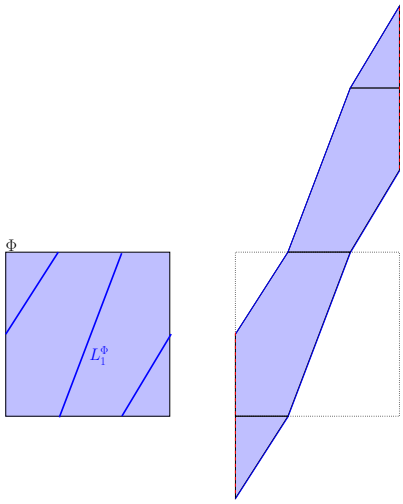
- We know that $0 \neq s_{E_0} := \text{sign}(\overrightarrow{L}_{E_0})$. By Corollary 4.11 and its implications, the outer CCB defined by E_1 also *points* into F' if $s_{E_1} \neq s_{E_0}$. Similarly, the outer CCB defined by E_2 belongs to F' if $s_{E_2} \neq s_{E_0}$. Note that exactly one of s_{E_1} or s_{E_2} is expected to fulfill this property.

Surface has two curves of identification: We are left with the case that the parametric surface S comprises two curves of identification, that is, there is a left-right identification and a bottom-top identification. Such a surface is homeomorphic to a torus; as example, we discuss ring Dupin cyclides in §4.6.2. We basically want to apply the same ideas as for a surface with a single curve of identification. Fortunately, this case can be simulated: Ideally, one would actually split S along some curve of identification. This would be the simple solution. However, this "pre-processing"

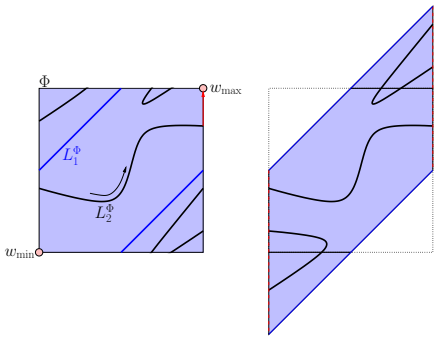
Figure 4.17. Removing a non-contractible loop L_1 from a surface with two identifications results in a subsurface $S \setminus L_1$ that can be parameterized with a single identification. See red dashed lines in the "recombined" views of parameter space (on each right side).



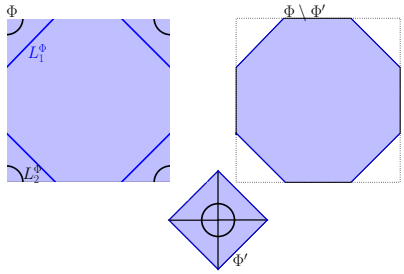
(a) L_1 crosses left-right identification once



(b) L_1 crosses left-right identification once and bottom-top identification twice



(c) L_1 crosses left-right identification once and bottom-top identification once. The black loop L_2 is a perimetric loop in $S \setminus L_1$, and thus also in S. If traversed in the specified direction is has positive sign and thus turns to $w_{\max}$ and abandons from $w_{\min}$. Similar loops exists in Figure (a) and (b).



(d) Counter-example: L_1 crosses left-right identification twice and bottom-top identification twice. This splits Φ (and so S) into two disjoint sets of points, in contrast to Figures (a-c). In addition, each loop in $\varphi_S(\Phi')$ is contractible to a point (e. g., L_2)

contrasts with the on-line strategy of the visitor. Note that due to double identification any first non-contractible loop L_1 does *not* split the surface into two disjoint components. In contrast, the surface "exists" to both sides of the loop, that is, $S \setminus L_1$ is homeomorphic to an open cylinder. We refer to Figure 4.17 for some examples of such loops in parameter space. This property of L_1 is also the reason why we have to deal with case (3). Actually, the detection of this case is still undetermined. Notice that a loop is non-contractible if it crosses a curve of identification an odd number of times. That is, to decide (3) upon the insertion of a curve, we only have to test, whether it triggers the first loop that crosses some identification an odd number of times. Depending on the strategy we can then select option (c) or option (d). See again Figure 4.17 (a-c): The left-right identification is crossed an odd number of times, while the number of bottom-top crossings varies. The parameter spaces can be recombined such that a single identification remains, namely the one that has been crossed by L_1 an odd number of times, and thus has been *selected*. If both identifications are crossed by L_1 an odd number of times, each can be chosen.

In fact, as L_1 is formed by curves embedded on S no other loop on the surface can cross this "curve of identification" L_1. Thus, everything we previously presented for a surface with a single curve of identification now holds for $S \setminus L_1$. We only have to restrict signs of paths with respect to the one identification of S (out of two!) that is selected by L_1.

4.5. The *ArrTopologyTraits_2* concept

In the previous section we contoured which changes the `Arrangement_2` package has undergone during its transition to the `Arrangement_on_surface_2` package, and we identified tasks expected from an instance of the `TopologyTraits_2` parameter. In this section we tight the specifications and exactly define the *ArrTopologyTraits_2* concept. In our presentation, we group tasks serving related (or similar) purposes. Although technical, we omit details in our presentation that are usually expected by an actual reference manual. In §4.6 we shortly review available models for different surfaces, and deep the description of implementation details for two selected families of surfaces.

4.5.1. Nested types

We expect that each model of the *ArrTopologyTraits_2* concept is parameterized by a suitable geometric-traits class, so each also knows the proper geometric type definitions. As already noticed, the `TopologyTraits_2` parameter replaces the `Dcel` parameter, so first of all, a model is expected to provide the following type.

- `Dcel` — the DCEL-model that is used to represent the two-dimensional subdivision. It must be a model of CGAL's *ArrangementDcel* concept (see [WFZH07a]). We here only remember the non-standard extension for it, namely that a face can have no, one, or several inner and outer CCBs (and isolated vertices). We also remark the possibility to define a DCEL type that allows to extend its records by additional data; see also §2.4.3.

As mentioned, visitors combined with CGAL's generic `Sweep_line_2` class-template enable to compute various output. As we are aiming to support a basic subset, each model of the *ArrTopologyTraits_2* concept has at least to provide the following visitors:

- `Sweep_line_construction_visitor` — this visitor is expected to construct a new arrangement from a set of input curves (or points). It is used by the global `CGAL::insert` function for aggregated insertion of curves into an arrangement, if it is empty. The concept also expects the `Sweep_line_non_intersecting_construction_visitor` type, which either implements a specialized version for non-intersecting curves, or it just redefines the `Sweep_line_construction_visitor`, knowing that its `Intersect_2` function object is never queried.

- `Sweep_line_insertion_visitor` — using this visitor while sweeping over an existing arrangement inserts a set of new input curves into it. The `Arrangement_on_surface_2` packages dispatches this visitor, when calling the global `CGAL::insert` function when aggregately inserting a set of curves in an arrangement which is not empty. Like for the construction, the `Sweep_line_non_intersecting_insertion_visitor` type is also expected. Again, either a specialized implementation takes advantages of the non-intersection property, or the model redefines the `Sweep_line_insertion_visitor` type knowing that `Intersect_2` is never called.

- `template <class ArrA, class ArrB, class OverlayTraits>`
  `Sweep_line_overlay_visitor` — this visitor is combined with the sweep line algorithm in the global `CGAL::overlay` function with the goal to compute the overlay of two arrangements (of different types "A" and "B", but with same geometry of curves and same topology of the underlying surface). The recombination of attached data to two DCEL-records into one is processed according to the given `OverlayTraits` type.

- `template< class OutputIterator >`
  `Sweep_line_batched_point_location_visitor` — combining this visitor with the sweep line algorithm enables to answer a batched point-localization, that is, to localize a set of points.

In order to simplify the development of visitors, there exists for each task a class-template that can be specialized using small helper structures respecting the surface's topology. The template implements the surface-independent code for a certain objective (constructing, inserting, overlaying, et cetera) while the helper "fills in" the missing surface-specific details. Of course, it is allowed, though not encouraged, to develop each visitor from scratch.

As for the sweep line algorithm, CGAL's zone algorithm can also be combined with a visitor instance in order to compute visitor-specific output during the zone computation. For arrangements on surfaces, an *ArrTopologyTraits_2* model is expected to support the insertion of curves to an arrangement with the following visitor.

- `Zone_insertion_visitor` — the insertion of a single (weakly) x-monotone curve into an existing, not necessarily empty, arrangement with the global `CGAL::insert` function is internally performed by combining this visitor with the zone algorithm.

Besides this minimal set of visitors, each model can also provide visitors that enable other applications. For example, there exists a class-template for a visitor that computes the vertical decomposition of an arrangement while the sweep processes. As for the others, we only have to provide the surface-specific helper class. Remember that the zone algorithm expects the possibility to locate points (or curve-ends). In order to support this, the following type is expected.

- `Default_point_location_strategy_2` — this type must be a model of CGAL's *ArrangementPointLocation* concept. It supports point-location queries in an arrangement. As not all point-location strategies work on all surfaces, a model of the *ArrTopologyTraits_2* concept has to define this type which specifies the default approach for point locations if no other strategy is provided by the user (e. g., for an incremental insertion).

4.5.2. The boundary of the parameter space

In addition to the nested types, a model of CGAL's *ArrTopologyTraits_2* concept also has to provide some member functions. We start with very basic ones. The first provides information about what happens on the boundary of the parameter space.

- ```
  Arr_boundary_type boundary_type(
     Arr_parameter_space ps
  )
  ```

  returns the boundary type for a given location on the boundary of the parameter space: For given `ARR_LEFT_BOUNDARY`, `ARR_RIGHT_BOUNDARY`, `ARR_BOTTOM_BOUNDARY`, or `ARR_TOP_BOUNDARY` it returns a value of the following enumeration.

  ```
  enum Arr_boundary_type
  {
    ARR_BORDER = 0,
    ARR_UNBOUNDED,
    ARR_CONTRACTION,
    ARR_IDENTIFICATION
  };
  ```

4.5.3. Members for the DCEL

The next members are related to the DCEL.

- `Dcel& dcel()`

  returns a reference to the internal DCEL representation. This ensures referential modifications of the DCEL by the `Arrangement_on_surface_2` class-template for non-boundary cases.

- `void init_dcel()`

  initialize an empty DCEL structure for the specific topology of the surface.

- `bool is_empty_dcel()`
  ```

returns `true` if the arrangement is *empty*, and `false` otherwise. An empty arrangement is attained, if no curve or point induces a one- or zero-dimensional cell on S. In particular, it returns `true` when called right after `init_dcel()`.

Remember that we allow *fictitious* DCEL-records. Such records do not store geometric information, but some topologies rely on them to model certain boundaries as DCEL. On the other side, a user of an instantiated `Arrangement_on_surface_2` class-template does not want to care about such artificial objects. Thus, the arrangement in cooperation with the topology-traits class filters unwanted records.

Definition 4.12 (Valid and concrete DCEL-records).

- A face is called *valid* if it represents an open two-dimensional subset of points on S. See Figure 4.9 (a): F_i are valid for the unbounded plane, while $\overline{F}$ is invalid.
- A halfedge is called *valid* if it is incident to a valid face and represents an open one-dimensional subset of points on S. The solid halfedges in Figure 4.9 (a) are valid, while the dashed ones are invalid.
- A vertex is called *valid* if it is incident to a valid halfedge. The vertices V_i in Figure 4.9 (a) are valid, while V_{bl}, V_{tl}, V_{br}, and V_{tr} are invalid.
- A vertex is called *concrete* if is valid and has a finite point attached. All vertices except V_i and V_{bl}, V_{tl}, V_{br}, and V_{tr} in Figure 4.9 (a) are concrete (i. e., the red ones).

To the user, the arrangement class filters non-concrete vertices, and non-valid halfedges and faces. For this purpose the following members are expected. There are other filters that also return valid vertices. These are required, for example, in case one wants to run a graph algorithm on an arrangement.

- ```
  bool is_valid_face(
    const face *f
  )
  ```

  checks whether a given face is valid.

- `Size number_of_valid_faces()`

  returns the number of valid faces stored in the DCEL. Return type is `Size` which is a nested type in `Dcel`.

- ```
  bool is_valid_halfedge(
    const Halfedge *he
  )
  ```

  checks whether a given halfedge is valid.

- `Size number_of_valid_halfedges()`

  returns the number of valid vertices stored in the DCEL.

- ```
  bool is_valid_vertex(
    const Vertex *v
  )
  ```

  checks whether a given vertex is valid.
  ```

- `Size number_of_valid_vertices()`

  returns the number of valid vertices stored in the DCEL.

- ```
  bool is_concrete_vertex(
    const Vertex *v
  )
  ```

  checks whether a given vertex is concrete.

- `Size number_of_concrete_vertices()`

  returns the number of concrete vertices stored in the DCEL.

4.5.4. Vertices and edges on the boundary

In §4.4 we already detected that the arrangement class is able to handle DCEL-records in the interior of the parameter space on its own, while for DCEL-records related to the boundary of the space it relies on external and surface-specific query results. For this reason it interacts with the following members of a model of the *ArrTopologyTraits_2* concept.

- ```
  CGAL::Object place_boundary_vertex(
    Face *f,
    X_monotone_curve_2 xcv,
    Arr_curve_end ce,
    Arr_parameter_space psx,
    Arr_parameter_space psy
  )
  ```

  We are searching for the position of a vertex to be constructed that represents the given curve-end. The location of the curve's end is on the boundary, that is, exactly one of `psx` or `psy` is equal to `ARR_INTERIOR`. The returned object may either be empty, it may wrap a fictitious edge that is going to split for the vertex, or it comprises of a vertex to which to curve's halfedges will be connected.

- ```
  void notify_on_boundary_vertex_creation(
    Vertex *v,
    X_monotone_curve_2 xcv,
    Arr_curve_end ce,
    Arr_parameter_space psx,
    Arr_parameter_space psy
  )
  ```

  This member is called to notify the instance of the *ArrTopologyTraits_2* model by the arrangement on the creation of a new vertex on the boundary. This notification helps to keep the internal structure of the model up to date, for example, to maintain a sorted list of vertices for an identification. On the other side, the arrangement class is still able to send notifications to observers upon structural changes of the DCEL.
  ```

- ```
  void locate_curve_end(
    X_monotone_curve_2 xcv,
    Arr_curve_end ce,
    Arr_parameter_space psx,
    Arr_parameter_space psy
  )
  ```

  While `place_boundary_vertex` is called when information about the face containing the curve-end is available (e. g., during the sweep), this member locates the DCEL feature that contains a given curve-end, which must relate to the boundary of the parameter space. It can either be an existing vertex, an existing edge, or an existing face. The method forms a subtask demanded by a point location operation.

- ```
  Halfedge* locate_around_boundary_vertex(
    Vertex *v,
    X_monotone_curve_2 xcv,
    Arr_curve_end ce,
    Arr_parameter_space psx,
    Arr_parameter_space psy
  )
  ```

  If a curve-end is detected to be incident to a vertex on the boundary, this function locates the predecessor halfedge in the circular order of halfedges around the vertex. The location on the boundary is encoded with `psx` and `psy` as for the other two localizations members. If the vertex is isolated, it returns `NULL`.

- ```
  Halfedge* split_fictitious_edge(
    Halfedge *he,
    Vertex *v
  )
  ```

  On the other hand, the localization of a curve-end on the boundary might return a fictitious edge. This member performs the split of the edge at the vertex that represent the new curve-end. It returns one of the newly incident halfedges to the vertex. Note that the topology-traits class implements this function, as it is a modification of the DCEL representing the boundary of the parameter space.

- ```
  bool are_equal(
    Vertex *v,
    X_monotone_curve_2 xcv,
    Arr_curve_end ce,
    Arr_parameter_space psx,
    Arr_parameter_space psy
  )
  ```

  Checks if a given vertex on the boundary is associated with the given curve-end on the boundary. Is used, for example, to distinguish whether the minimal or maximal end of a curve is incident to the vertex.

The creation of boundary vertices is not the sole purpose of the traits. If deleting a curve related to the boundary (or an isolated vertex on it) the DCEL also requires surface-specific updates that are supported by the following two member functions.

- ```
  bool is_redundant(
    const Vertex *v
  )
  ```

  Determines whether the given vertex on the boundary has become redundant. If so, the arrangement triggers its deletion.

- ```
  Halfedge* erase_redundant_vertex(
    const Vertex *v
  )
  ```

  Erases the given redundant vertex (e. g., by merging fictitious edges). The function is not expected to free the vertex. It returns one of the merged twins of halfedges.

4.5.5. Faces and their boundaries

For the last set of members, we turn towards the designated faces of the arrangement. We start with two simple predicates.

- ```
  bool is_unbounded(
    const Face *f
  )
  ```

  Decides whether a given face is unbounded.

- ```
  bool is_in_face(
    const Face *f,
    Point_2 p,
    const Vertex *v
  )
  ```

  Determines whether the given point lies in the interior of the given face, ignoring inner components and isolated vertices contained in it. If the point is already associated with a vertex, then v is not null and finite.

Finally, each model of the *ArrTopologyTraits_2* concept must provide information required to correctly construct or delete faces in sync with proper update of relevant CCBs.

- ```
  std::pair<bool, bool> face_split_after_edge_insertion(
    const Halfedge *prev1,
    const Halfedge *prev2,
    X_monotone_curve_2 xcv
  )
  ```

  This member is queried when the curve `xcv` is going to be inserted at the target vertices of `prev1` and `prev2`. Both determine the position where to insert the new pair of halfedges in the circular order of halfedges around the vertices. We also know that both predecessor halfedges belong to the same inner CCB. The function has to compute what happens when the insert is accomplished. To do so, it returns a pair of boolean values. The first flag indicates whether the insertion will cause the face to split. If yes, the second determines whether the split face will form a new inner component nested below the original face. Otherwise, the split face becomes
  ```

equitable to the originating one. If the first returns false, the second determines, whether the two CCBs emerging from a non-simply connected loop on S should be transformed into two outer CCBs (`false`) or two inner CCBs (`true`).[38] We remark that this function implements the topology-traits class' decision which out of the four options (a), (b), (c), or (d) presented in §4.4.3 (on page 138) should be triggered.

- ```
  bool hole_creation_after_edge_removal(
    const Halfedge *he
  )
  ```

  The function somehow constitutes the complement of the previous one. It determines whether the removal of a given halfedge (and, of course, its twin) will cause the creation of a hole. The function is only queried if both `he` and its twin lie on an outer CCB, and both do not represent the tip of an antenna.

The remaining two members are related to assignments of the CCBs:[39]

- ```
  bool is_on_new_face_boundary(
    const Halfedge *prev1,
    const Halfedge *prev2,
    X_monotone_curve_2 xcv
  )
  ```

  The situation is similar as for `face_split_after_edge_insertion`, that is, the two halfedges are predecessor edges of the same inner CCB that is perimetric. They are used for the insertion of `xcv` which separates a new face. It must be decided whether `prev1` will be incident to this new face or not. That is, it decides whether `prev1` is going to define the outer CCB of the new face. The split face can be perimetric or not. The originating one stays perimetric in any case.

  Consider as an example the closing of the northern tropic on a sphere in Figure 4.11 (b), where $E'_{1,\mathrm{prev}}$ is finally inside the new face F'_S and thus E'_1. Similar situations are given in Figure 4.11 (c), Figure 4.12 (b) and (c) and Figure 4.16 (b). In all these example the split face is perimetric. Figure 4.16 (c) gives an input where the split is non-perimetric.

- ```
  bool boundaries_of_same_face(
    const Halfedge *he1,
    const Halfedge *he2
  )
  ```

  The situation is as follows: a perimetric face has just split into two perimetric and equitable faces. That is, no new inner CCB is constructed. Only two outer CCBs appear. The halfedge `he1` defines an outer CCB of the original face, while `he2` is an outer CCB that just emerged along one of the two sides of the perimetric loop that triggered the split. It must be determined, whether `he2` points into the same face as `he1`. The actual question is whether the two outer CCBs have different *directions* with respect to the face defined by `he1`.

[38] This second case, it not yet realized in CGAL's implementation.

[39] The current concepts expects `is_on_new_perimetric_face_boundary()`. However, its actual semantics is not covered by this. Thus, for this presentation we chose to give a less restricted name.
  ```

For an example see Figure 4.13 (c): It must be determined whether the CCB defined by E_2 or the one defined by E_2' (both just emerged) belongs to the same face as E_1' does, namely to the new split face F_2. Similar situations are given in Figure 4.14 (b) and (c), and Figure 4.25 (b).

Let us give some final remarks.

Remarks.

- Remember that DCEL-records for objects in the interior of the parameter space are created and maintained by the arrangement class itself, while the topology-traits modifies those related to the boundary of the parameter space. This has implications on observers attached to an arrangement. Remember that an observer receives notifications about the arrangement's structural changes. Our chosen design still allows the arrangement to send such notifications, even if DCEL-records related to the boundary of the parameter space are constructed or deleted. For example, it sends `before_split_fictitious_edge()` prior to calling `split_fictitious_edge()`, and `after_split_fictitious_edge()` after calling this topology-traits method. Other examples are the creation and deletion of DCEL-vertices on the boundary.
- Models of the *ArrTopologyTraits_2* concept can provide special surface-specific member functions. An example is the access to a sorted sequence of DCEL-vertices along identified boundary sides.

We have to admit, that the concept, although quite stable, is still under development. The presented details correspond to its status at the date of thesis's submission. Further changes that improve or extend the interface are conceivable. In particular, it must be checked what is missing to finally support isolated vertices on and curves fully contained in the boundary of the parameter space. In addition, the interface with respect to CCBs is serving all cases; however, it seems complicated. We hope to be able to simplify it. However, the design is successful: This fact is emphasized by the variety of existing models. In §4.6, we first list available classes, followed by a detailed discussion of two models that support important non-linear surfaces.

4.6. Examples

Combining the different possibilities for the four boundaries of the parameter space results in a large list of feasible (and also some infeasible) topology-traits classes; see Table 4.1. The combinations representing basic families of surfaces are already implemented, that is, CGAL provides geometric-traits and topology-traits classes for them:

For the plane, we distinguish one topology-traits class for bounded curves, and one for unbounded curves that implements the implicit rectangle of fictitious edges around the scene; see §4.4 and [WFZH07a]. A set of geometric-traits classes for various kinds of curves in the plane exists. We exemplary mention classes handling linear objects, circles, conics, rational curves, and Bézier curves; see also §2.4.3. All of them fulfill *UnboundedBoundaryTraits* concept at all four sides, that is, each supports curves that extend to infinity in any direction. The same holds for CGAL's the generic model named `Curved_kernel_via_analysis_2` that we presented in §2.4.4. It is used in [EK08a] to compute arrangements of unbounded algebraic curves of any degree by instantiating the class-

template with a suited bivariate algebraic kernel. The authors of the article provide CGAL's adequate `Algebraic_curve_kernel_2`.

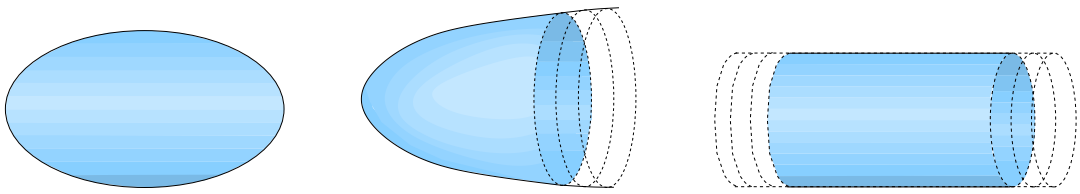
As first non-planar surface, CGAL provides a topology-traits class for the sphere, which contracts bottom and top boundary and identifies left and right boundary. A geometric-traits class for geodesic arcs on the unit-sphere is available. A geodesic arc is the shortest connection between two points on a surface. Exact rational arithmetic suffices to provide all relevant geometry-traits operations. The authors of [FHS08] give details on the traits classes, and also show various applications. An example is the overlay of maps on a *model of the earth*, or to compute a Voronoi diagram of points on the sphere using CGAL's generic divide-and-conquer algorithm for lower envelopes. Another application is the exact computation of Minkowski sums of convex polyhedra using Gaussian maps; see [BFH+07]. There is also a video [FSH08]. Sébastian Loriot from INRIA (Sophia-Antipolis) is working on a geometric-traits class that deals with arbitrary circles on a sphere. He adapts previous work [CL07] with respect to the design of CGAL's `Arrangement_on_surface_2` package. It is worth to mention, that he is possible to use the existing topology-traits class for the sphere. We do not discuss details on these workings.

In contrast, at the end of this chapter, we now focus on two sophisticated examples of surfaces, namely elliptic quadrics and ring Dupin cyclides. The later constitute a generalization of tori. We present details on both surface-specific topology-traits models whose discussion comprises interesting aspects to consider with respect to the occurring identifications. For each surface we provide a juicy geometry-traits class. The remarkable fact for both geometric-traits classes is, that they reduce the geometry on the surface to a planar geometry. More detailed, each geometric-traits class inherits from the planar `Curved_kernel_via_analysis_2`, and augments (modifies) it case-specifically in order to model the appropriate *"CombinedBoundaryTraits"* concept required for the surface.

For each of the two examples in §4.6.1 and §4.6.2 we first give a short introduction, followed by details on the geometry- and topology-traits classes, and conclude with results.

4.6.1. On a quadric

Figure 4.18. Elliptic quadrics



(a) ellipsoid **(b)** elliptic paraboloid **(c)** elliptic cylinder

Given a list of quadrics $q_0, q_1, \ldots, q_n$. Remember from Definition 3.3 that a quadric is an algebraic surface that is formed by the vanishing set of a trivariate polynomial of total degree 2. We often abuse notation and refer to q_i as the polynomial and the vanishing set, depending on the context. We call q_0 the *reference quadric*, while q_i, $1 \leq i \leq n$ are supposed to intersect with q_0, constituting the *intersecting set*. We show how to compute the arrangement on q_0 induced by the intersecting set using CGAL's `Arrangement_on_surface_2`

package that is instantiated with a proper geometric-traits and topology-traits class. This implementation is robust, that is, it handles all degeneracies,[40] and is exact, as all underlying geometric operations follow the exact geometric computation paradigm. For this example case, we restrict the choice of the reference quadric q_0 to be an x-elliptic one.

Definition 4.13 (Elliptic quadric). A quadric q is *x-elliptic*, if the intersection of any plane $x = x_0$ with q is an ellipse (embedded in the given plane).

The set of x-elliptic quadrics comprises all ellipsoids, elliptic cylinders that are unbounded in x-direction, and paraboloids that are either unbounded towards $x = -\infty$ or $x = +\infty$. Figure 4.18 collects the three cases. These quadrics have pretty properties: First, they consists of a single connected component and second, they allow a nice geometric-traits class that we derive next.

Remarks.

- The techniques that we deploy next can be similarly applied to all other quadrics. For references quadrics consisting of two connected components (e.g., hyperboloid of two sheets) two individual arrangements must be constructed.
- There is no restriction on the choice of quadrics $q_1, \ldots, q_n$ in the intersecting set. They can be arbitrary. In fact, in Chapter 5 we present techniques that enable us to consider algebraic surfaces of any degree as intersecting set and still using the same special *constructed parameter space* that we introduce here.

The geometry

The non-xy-functional elliptic quadric q_0 can be subdivided into two xy-functional surfaces ($z = f(x, y)$) by a single curve. This *silhouette* is given by $silhouette(q_0) := V(q_0) \cap V(\frac{\partial q_0}{\partial z})$. It induces the *lower* and *upper* part of q_0. For example, the equator splits the sphere into the northern and into the southern hemisphere. Both hemispheres are xy-functional. The *projected silhouette* of q_0 onto the xy-plane is algebraically defined by $\mathrm{Res}_z(q_0, \frac{\partial q_0}{\partial z})$.

Consider the spatial intersection curve of q_0 with another quadric q_i, that is, $V(q_0) \cap V(q_i)$. The (Zariski-closed) projection of this set onto the xy-plane is a real algebraic plane curve of total degree 4, defined by $\mathrm{Res}_z(q_0, q_i)$. As in Chapter 3 we remember that such a projected curve can be split at its critical points and its intersection with the projected silhouette of q_0, resulting in isolated points and (weakly) x-monotone curves. Each such object can be assigned to the lower or upper part of q_0 (in some cases also to both parts); see [BHK+05] for details, or §3.2 for a rollback. In that original work, two individual arrangements that constitute the subdivisions on the lower part and on the upper part, respectively, are computed; to merge the two DCEL instances is missing.

In contrast, we here deploy the fact that an x-elliptic quadric q_0 is nicely parameterizable by $\Phi = U \times V = [l, r] \times [0, 2\pi]$, with $l, r \in \mathbb{R} \cup \{\pm\infty\}$, using $\varphi_{q_0}(u, v) = (u, y(u, v), r(u, y(u, v), -\sin v))$.[41] We define $y(u, v) = y_{u,\min} + (\sin \frac{v}{2})(y_{u,\max} - y_{u,\min})$. The interval $[y_{u,\min}, y_{u,\max}]$ denotes the y-range of the ellipse that q_0 induces on the plane

[40] Though described in §4.2.2, the implementation of the `Arrangement_on_surface_2` package currently lacks support for isolated points and curves on the boundary of the parameter space. Thus, some special input is not yet handled — in software.

[41] Actually, the interval U is open on the sides where l or r are infinite.

$x = u$. The function $r(x, y, s)$ returns the minimal ($s \leq 0$) or maximal ($s > 0$) element of $\mathcal{R}_{q_0,x,y} := \{z \,|\, q_0(x, y, z) = 0\}$, $|\mathcal{R}_{q_0,x,y}| \leq 2$.

However, this parameterization is stated only to show its existence. For our practical realization, we make use of its properties only. Note that the sin-function divides the parameter space "horizontically" into two parts, namely $\Phi_0 := [l, r] \times [0, \pi]$ and $\Phi_1 := [l, r] \times (\pi, 2\pi)$. These parts directly correspond to the (closed) lower part of q_0 and the (open) upper part of q_0. As $\varphi_{q_0}(u, 0) = \varphi_{q_0}(u, 2\pi)$, we detect a curve of identification for this parameterization. This curve is a subset of q_0's silhouette. Depending on the type of q_0, if l (or r) is finite, we detect a contraction point (ellipsoid, bounded tip of paraboloid) or an unbounded side (infinite end of paraboloid, cylinder). In Figure 4.19 we illustrate such a partitioning on the example of a paraboloid that is intersected by some other quadrics.

The partitioning into two areas is the key tool to define our special geometry on the reference quadric using as basic ingredient a planar geometry. Given a point $w_0 = (u_0, v_0)$, with $p_0 := \varphi_{q_0}(u_0, v_0) = (x_0, y_0, z_0)$ being its counterpart on q_0, the *level* of p_0 is $\ell \in \{0, 1\}$ if $w_0 \in \Phi_\ell$. We represent a point $p_i = (x_i, y_i, z_i)$ on q_0 as the combination of a planar point $\overline{p}_i(x_i, y_i)$ and its level $\ell_i \in \{0, 1\}$. Given two points p_1, p_2, the uv-lexicographic order of their counterparts w_1, w_2 in parameter space is reflected by the order of $x_1 = u_1$ and $x_2 = u_2$, and if $u_1 = u_2$ we infer the v-order from (y_1, ℓ_1) and (y_2, ℓ_2): If $\ell_1 < \ell_2$ then $w_1 <_{\text{lex}} w_2$ (and thus $p_1 <_{\text{lex}} p_2$), else if $\ell_1 = \ell_2 = 0$, then w_1 and w_2's v-order is identical to the y-order of $\overline{p}_1$ and $\overline{p}_2$. If, finally, $\ell_1 = \ell_2 = 1$, then w_1 and w_2's v-order is attained by the opposite of $\overline{p}_1$ and $\overline{p}_2$'s y-order.

A u-monotone arc cv on q_0 is represented by a projected arc $\overline{cv}$ that is enhanced by three levels, namely $\ell_{\min}$ at the minimal end of $\overline{cv}$, $\ell_{\max}$ at the maximal end of $\overline{cv}$ and ℓ representing the level in the interior of $\overline{cv}$.

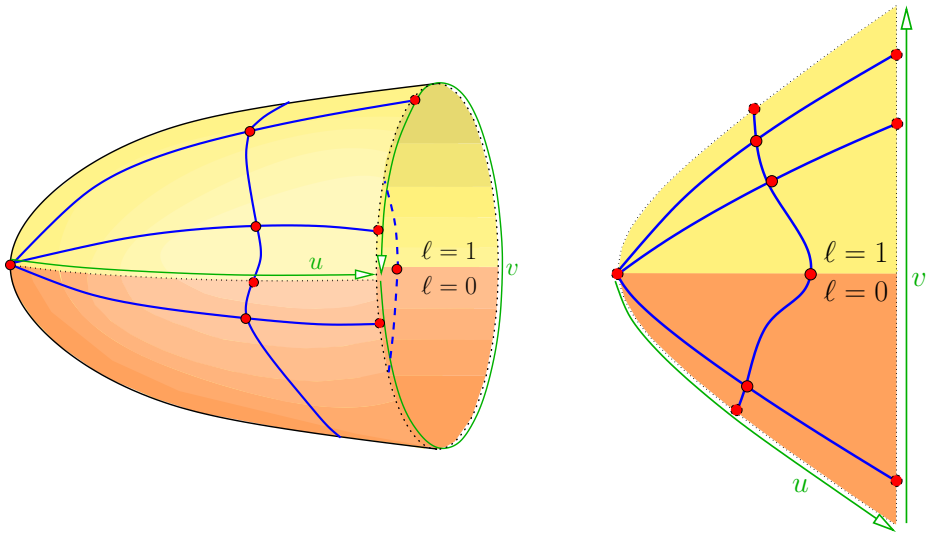
Remarks.

- Note that the level in the interior of an arc is constant, as we split each projected intersection curve also at its intersections with the projected silhouette.
- Remember that Φ_0 is closed, which has the following implication: Consider an arc with $\ell = 1$ (lying on the upper part of q_0). If one of its ends lies on q_0's silhouette, the level of this end is 0. This holds, in particular, if the end meets the curve of identification.

Our goal is to reuse the `Curved_kernel_via_analysis_2` (that is instantiated with CGAL's `Algebraic_curve_kernel_2`) in order to provide a geometric-traits model that reflects our defined lexicographic order in the constructed parameter space of q_0. In particular, we developed the following steps for our model of the *"CombinedBoundaryTraits"* concept.

1. Derived `Quadric_point_2` from `Point_2` and `Quadric_arc_2` from `Arc_2` that extend the projected objects with one level (point) or three levels (arc).
2. Derived `Quadrical_kernel_via_analysis_2` from `Curved_kernel_via_analysis_2`. In this step we replace the point and the arc type by the quadrical derivations. This requires some worth-to-mention sophisticated template programming. However, we chose not to do, as these technical details do not serve the simplicity and elegance of this approach and presentation.
3. Adapted `Make_x_monotone_2` to partition the spatial intersection curve of q_i with q_0

Figure 4.19. Illustration of simulation of a paraboloid's parameterization: the dark-shaded (orange) area represents Φ_0, the bright-shaded (yellow) area corresponds to Φ_1.



(a) On the paraboloid

(b) Simulation in the plane by *inversion* of upper part

into instances of type `Quadric_arc_2` and `Quadric_point_2` (for isolated points). Note that the input type `Curve_2` is a second quadric. The actual input is implicitly defined by the reference quadric and any second. We finally store locations of points and ends of curves with respect to the parameter space (`ARR_INTERIOR` or `ARR_[LEFT,RIGHT,TOP,BOTTOM]_BOUNDARY`).

4. Derived all geometric predicates that involve y-comparisons. The relevant functors are `Compare_xy_2`, `Compare_y_at_x_2`, and `Compare_y_at_x_right_2`. We modified them to reflect the special lexicographic order of our constructed parameter space. More detailed, we return the opposite result of comparison of y-coordinates, if both input objects have level 1.
5. Derived and modified constructions with respect to levelling. Relevant functors are, for example, `Intersect_2` and `Split_2`.
6. Derived and modified `Compare_[x,y]_near_boundary_2` to reflect the order of curves close to contracted and infinite boundaries (in "x"-direction) or close to the identification (in "y"-direction).
   We only discuss `Compare_y_near_boundary_2` on the example of a comparison next to a left contraction point: Given two minimal curve-ends of curves $cv_1 \mathrel{\widehat{=}} (\overline{cv}_1, \ell_1)$ and $cv_2 \mathrel{\widehat{=}} (\overline{cv}_2, \ell_2)$ approaching the left boundary (the contraction point). The levels at their minimal ends must both be 0. If $\ell_1 \neq \ell_2$ (the interior levels), the desired v-order is simply the order of ℓ_1 and ℓ_2. If $\ell_1 = \ell_2$, then both arcs lie on the lower part or both on the upper part of q_0. In the first case, the correct v-order is attained by the result when the projected counterparts $\overline{cv}_1$ and $\overline{cv}_2$ intersecting at the minimal ends (due to contraction) get compared slightly to the right of the projected contraction point. This is established by calling the planar `Compare_y_at_x_right_2` for $\overline{cv}_1$ and $\overline{cv}_2$ and

their common minimal point (i. e., the projection of the contraction). In case that both curves lie on the upper part, the opposite of this result reflects the correct v-order of cv_1 and cv_2 near the left contraction. In case of a right contraction point, we have to use the planar `Compare_y_at_x_left_2` and the corresponding maximal ends. The others cases (unbounded, compare near x) similarly combine level comparisons and planar predicates.

7. Implemented `Compare_x_on_boundary_2` to interface the order of points along the curve of identification. This predicates uses the planar `Compare_x_2` for points. Note that this is required to implement the *IdentifiedBoundaryTraits* concept.

Conceptually, all these modification and extensions are simple recombinations of the existing planar (much more sophisticated) counterparts. However, it is the straightforwardness of the levelling that allows to simulate the constructed parameter space of the elliptic quadric in terms of projection without explicitly knowing the actual parameterization. As the levels tests are purely combinatorial, we expect that the planar operations mainly influence the performance of this geometric-traits.

Concerning the implementation, our well-designed derivation hierarchy based on template programming allows the definition of the `Quadrical_kernel_via_analysis_2` in its various details. We can even use CGAL's `Filtered_curved_kernel_via_analysis_2` as the planar *oracle* inside the `Quadrical_kernel_via_analysis_2`; see §2.3.3 (page 58 ff).

The topology

The topology of the reference quadric q_0 requires special handling. We next discuss details of our topology-traits class (`Arr_qdx_topology_traits_2`) that combines the various cases (ellipsoid, paraboloid, cylinder). Remember that the topology-traits classes mainly helps to consistently construct a DCEL respecting the surface's topology. We already remark that our model realizes the *tree-strategy for ellipsoids and paraboloids*, while it *applies the forest-strategy for cylinders*.

It starts with the initialization of the DCEL, which for our elliptic quadric q_0 requires to construct a *single face* that has *no outer CCBs* and *no inner CCBs*. It is *bounded*, if q_0 is an ellipsoid, and *unbounded* if q_0 is a paraboloid or a cylinder.

The topology-traits class also maintains special DCEL-vertices, namely those related to the four sides of the parameter space. For the left and the right side, two special vertices V_{left} and V_{right} are designated. Such a vertex records the incidences of curves to either a point of contraction, or an unbounded end depending on q_0's shape: For an ellipsoid both vertices represent contraction points, for a cylinder both represent unbounded sides, and the orientation of the paraboloid determines whether the left is a contraction and the right is unbounded, or vice versa. Vertices on the identification of the bottom and top boundary are maintained in a sorted sequence (`std::map`). The order of stored vertices is defined by the order of attached points using the geometric-traits functor `Compare_x_on_boundary_2`.

The topology-traits class for quadrics also implements the localizations of curve-ends with `place_boundary_vertex` (and the similar `locate_curve_end`). Actually, this is a feasible task using a case-distinction on the given location in the parameter space. For curve-ends related to the left (right) boundary, we simply check if V_{left} (V_{right}) is `NULL`. If so, we return `NULL` which triggers its construction, if not, we return the existing vertex. For the identification this process is preceded by a look-up in the sorted sequence, that is, we actually check whether the topology-traits is already aware of a vertex on the identification at a cer-

tain x-coordinate. The update of the records after `notify_on_boundary_vertex_creation` relies on the same case-distinction.

It is also expected that `locate_around_boundary_vertex` locates a curve in the circular list of incidence curves around a vertex on the boundary. If the vertex equals V_{left} or V_{right} our implementation relies on the geometric comparisons `Compare_y_near_boundary_2`. For vertices on the identification, our code makes use of an internal functor of CGAL's `Arrangement_2` package: `Arr_traits_adaptor_2::Is_between_cw_2` checks whether a given curve is in counter-clockwise order between two curves that are already incident to a vertex. Its implementation is an elaborate combination of the `Compare_y_at_x_right_2` and `Compare_y_at_x_left_2` predicates.

Besides the localizations, the topology-traits must also provide information on the consistent construction of faces, and CCBs, in particular, if identifications are existing. The functions that must be implemented are listed in §4.5.5. As illustrated in §4.4.3 each of them can be implemented with the help of directed loops and the chosen strategy. Let us start with `face_split_after_edge_insertion` that is called upon the insertion of a curve whose predecessor edges belong to the same inner CCB. We have to decide two answers. The first is whether a face splits. The answer is always `true` as our parameterization of a quadric only involves a single curve of identification. Following, the special case (3) that would require to return `false` cannot occur. It remains to decide whether the split face should be nested below the originating one, or to become equitable to originating one. If q_0 is an ellipsoid or a paraboloid, we decided to go for the tree-strategy, and, thus, return `true`. That is, it gets nested. If q_0 is a cylinder, we follow the forest-strategy and thus evaluate the signs s_{prev_1} and s_{prev_2} of the two directed loops $\overrightarrow{L}_{prev_1}$ and $\overrightarrow{L}_{prev_2}$ that emerge upon the insertion of the curve cv in focus. If both are non-zero we return `true`. This implies that the new split face gets nested below the originating. We trigger option (a). Otherwise, at least one directed loop is perimetric and thus splits F such that no set of points makes the other locally non-simply connected. That is, we are in case (2) and return `false`. This triggers option (b) (as we are in the forest-strategy). The split face is then equitable to the originating one.

The required function `is_on_new_face_boundary` also exploits the non-zero values of directed loops to provide their answer. We rely on Corollary 4.11 for this purpose; see also §4.4.3. A directed loop with positive sign turns to $w_{\max}$, which corresponds to V_{right}. Following, we decide that the CCB defined by $prev_1$ becomes the outer CCB of the new face, if the $\text{sign}(\overrightarrow{L}_{prev_1}) \neq 1$. These decisions imply that the face which contains $\varphi_{q_0}(w_{\max})$ is considered to be outermost if we follow the tree-strategy, as we do for ellipsoids. This invariant is also feasible for a paraboloid that opens towards $x = +\infty$. Note that it even avoids unbounded leaves in the nesting tree. To also avoid unbounded leaves for a paraboloid that opens towards $x = -\infty$, we revert the decision: The CCB defined by $prev_1$ defines the outer CCB of the new face, if the $\text{sign}(\overrightarrow{L}_{prev_2}) \neq -1$. Following, for such a paraboloid, the face which contains $\varphi_{q_0}(w_{\min})$ forms the root of the nesting tree. For the cylinder, where we implement the forest-strategy, we can go with any consistent turning of directed loops towards some corner of Φ. Thus, we implement for such a q_0 the same decision as for an ellipsoid.

Lastly, the forest-strategy explicitly demands for `boundaries_of_same_face` that tests, whether a queried perimetric and outer CCB defined by E' belongs to the same face as another given perimetric and outer CCB defined by E. We have seen that this is the

case if $\text{sign}(\overrightarrow{L}_{E'}) \neq \text{sign}(\overrightarrow{L}_E)$. Thus, for cylinders, the `Arr_qdx_topology_traits_2` class implements this comparison.

Remark. It is superfluous to discuss functions related to fictitious edges here, as the chosen representation as DCEL goes without such. In addition, we also skip other straightforward members of the topology-traits concept.

As mentioned in §4.5.1, the topology-traits is finally expected to provides some visitor types. Fortunately, CGAL's `Arrangement_on_surface_2` package already provide generic implementations for construction, insertion, and overlay utilizing its `Sweep_line_2` class-template. We have to provide the quadric-specific helper classes. The "constructive" helper is responsible to pre-process events of the sweep line: Whenever an event on the boundary is going to be considered next during the sweep process, the helper first checks whether the attached topology-traits class already stores a corresponding DCEL-vertex for the event's point (or curve-end, in case of an event at infinity). If this is not the case, it simply triggers its constructions. In any case, it stores with the event a pointer to the obtained vertex. This later helps to correctly insert sub-curves into the DCEL that emerge to the right of the current event. This helper is also responsible to maintain a list of sub-curves that can see the top boundary of the parameter space. These sub-curves are candidates of inner components that must be relocated into a newly created split face. We also provide the helper classes that are required to insert curves into an existing arrangement, or to overlay two arrangements. Their implementations are similar: Actually, for each involved arrangement (*one* in the insertion case, the *red* and the *blue* arrangement in the overlay case), they maintain a pointer to the currently topmost face. A face F of an arrangement is called *currently topmost* if there is a simply-connected path in F from the current sweep event to the image of the parameter space's top boundary. In other words: If the current event would result in an isolated vertex, then, this vertex would be isolated in F. Both helpers update the corresponding pointer(s) upon processing events, that is, each modifies the pointer(s) when an event on the top boundary is "swept".

Results

We instantiated CGAL's `Arrangement_on_surface_2` class-template with the two described traits-classes, which results in a robust algorithm to compute an arrangement on an elliptic quadric. Even if the arrangement is highly degenerated it is successfully constructed by this piece of software, as the example in Figure 4.20 shows.

Base	**Ellipsoid**				**Cylinder**				**Paraboloid**			
Data	**#V**	**#E**	**#F**	**t (s)**	**#V**	**#E**	**#F**	**t (s)**	**#V**	**#E**	**#F**	**t (s)**
q50	5722	10442	4722	28.3	1714	3082	1370	12.5	5992	10934	4944	29.3
q200	79532	155176	75646	399.8	27849	54062	26214	189.9	82914	161788	78874	418.3
e50	870	1526	658	7.2	1812	3252	1442	14.4	666	1092	428	6.6
e200	10330	19742	9414	74.6	24528	47396	22870	175.8	9172	17358	8189	68.8

Table 4.2. Performance measures for arrangements induced on three base quadrics by intersections with 50 or 200 quadrics (q), or ellipsoids (e).

To demonstrate efficiency we also measured the performance when computing the arrangement on given base quadrics induced by intersections with other quadrics. As base

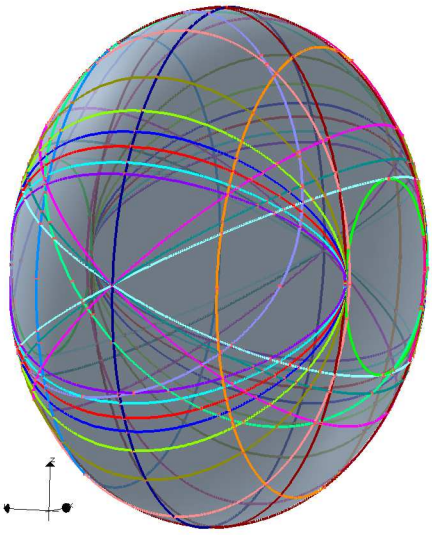


Figure 4.20. Degenerate arrangement on an ellipsoid induced by 23 other ellipsoids intersecting it

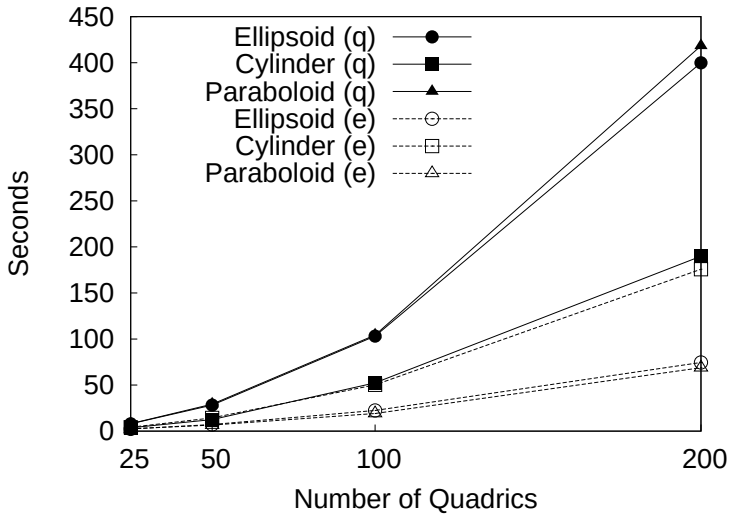


Figure 4.21. Performance measures for arrangements induced on three base quadrics by intersections with quadrics and ellipsoids (in seconds).

quadrics we created a random ellipsoid, a random cylinder, and a random paraboloid. These quadrics are intersected by two different families of random quadrics. The first family consists of sets with up to 200 intersecting generic quadrics, sets of the other family include up to 200 ellipsoids intersecting each of the base quadrics. The coefficients of all quadrics are 10-bit integers. All performance checks are executed on a 3.0 GHz Pentium IV machine with 2 MB of cache, with the exact arithmetic number types provided by LEDA and using CGAL's `Algebraic_curve_kernel_2` in wrapping mode for analyses of curves in the plane. That is, we rely on the curve analyses specialized to the quadrical case taken from EXACUS; see also [BHK+05].

Table 4.2 shows for selected instances the number of induced cells, as well as time consumption in seconds required to construct the individual arrangements. Figure 4.21 illustrates the average running time on up to 5 instances containing sets of ellipsoids (e) and general quadrics (q) of different sizes intersecting different base quadrics. Growth is super linear in the number of quadrics, as one expects for a sweep line approach.

Clearly, the more complex the arrangement, the more time is required to compute it. To give a better feeling for the relative time consumption, we indicate the time spent for each pair of half-edges in the DCEL of the computed arrangement. This time varies in the narrow range between 2.5 ms and 6.0 ms. Other parameters have significant effect on the running time as well, for example the bit-size of the coefficients of the intersection curves.

We next want to analyze the influence of the chosen topology-traits class. For these experiments we intersect instances from [Hem08] containing 10,20,40, and 80 quadrics with the three reference quadrics (ellipsoid, cylinder, and paraboloid). For each combination we compute three arrangements. Two planar ones, as in [BHK+05], that separately represent the induced arrangement on the lower and upper of the reference quadric, and one directly embedded on the surface using our new `Arrangement_on_surface_2`-framework with the quadrical topology-traits class. The splitting step is identical in both cases, as we have to assign arcs of planar curves to the lower and upper part of the reference. These experiments were executed on an AMD Dual-Core Opteron(tm) 8218 multi-processor Debian Etch platform, each core equipped with 1 MB internal cache and clocked at 1 GHz. The total memory consists of 32 GB. As compiler we used `g++` in version 4.1.2 with flags `-O2 -DNDEBUG`. For analyses of planar curves we rely on CGAL's new `Algebraic_curve_kernel_2` (in non-wrapping mode).

Table 4.3 gives the performance numbers of these computations. First of all, the obtained results show that the quadrical topologies are almost as fast as the two planar arrangements. However, they also compute slightly more: The two planar arrangements are not yet connected and this step requires non-trivial further processing. In contrast, the quadrical arrangement already correctly represents the reference's subdivision into cells of dimension 0, 1, and 2 induced by the intersecting quadrics. This also explains the non-matching numbers of cells: Vertices lying on the silhouette of the reference quadric are reported in *each* planar arrangement, but *only once* in the quadrical one. Further more, the planar arrangements are not bounded by the projected silhouette. Thus, their number of faces is typically smaller than for the on-surface arrangement. Besides saving a post-processing step, there is one more thing: CGAL's `Arrangement_on_surface_2` package is able to directly overlay two such arrangements. And one more: CGAL supports point location queries on such arrangements. And more; see [WFZH07b].

Reference: **Ellipsoid**									
	Split	**sweep two planar arrangements**				**sweep ellipsoidal arrangement**			
#	**t (s)**	#V	#E	#F	**t (s)**	#V	#E	#F	**t (s)**
10	**2.36**	213+217	295+289	84+84	**1.29**	396	584	190	**1.36**
20	**4.18**	544+540	844+838	302+300	**4.53**	1038	1682	646	**4.90**
40	**7.62**	1831+1837	3192+3210	1363+1375	**20.57**	3568	6402	2836	**21.50**
80	**15.47**	7187+7191	13363+13379	6178+6190	**97.66**	14144	26742	12600	**104.56**

Reference: **Cylinder**									
	Split	**sweep two planar arrangements**				**sweep cylindrical arrangement**			
#	**t (s)**	#V	#E	#F	**t (s)**	#V	#E	#F	**t (s)**
10	**1.65**	191+179	260+240	71+64	**1.17**	344	500	158	**1.23**
20	**3.38**	551+509	852+780	303+273	**4.74**	1012	1632	622	**5.00**
40	**6.76**	1821+1755	3168+3040	1349+1287	**21.28**	3474	6208	2736	**22.57**
80	**14.28**	7086+6914	13179+12831	6095+5919	**100.91**	13768	26010	12244	**108.76**

Reference: **Paraboloid**									
	Split	**sweep two planar arrangements**				**sweep paraboloidal arrangement**			
#	**t (s)**	#V	#E	#F	**t (s)**	#V	#E	#F	**t (s)**
10	**1.02**	28+16	37+13	11+2	**0.14**	36	50	17	**0.14**
20	**1.86**	124+96	181+129	60+35	**0.93**	196	310	116	**0.96**
40	**4.83**	469+337	787+533	321+198	**5.21**	756	1320	566	**5.38**
80	**9.87**	1303+1267	2309+2272	1008+1006	**20.25**	2472	4580	2110	**20.90**

Table 4.3. Comparing planar and quadrical topologies: We report performance measures (in seconds) for random quadrics intersecting three reference quadrics and distinguish the computation of two planar arrangements and one quadrical arrangement.

4.6.2. On a (ring) Dupin cyclide

We come to our final example, namely to compute arrangements on a parameterized ring Dupin cyclide Z. The family of Dupin cyclides contains regular tori as a special subset. The arrangements that we consider are induced by intersection of the arbitrary algebraic surfaces $S_1, \ldots, S_n$ with the given *reference cyclide* Z. This example is interesting for two reasons. First, the reference surface has genus one. Secondly, the geometric-traits class that we derive for this purpose is the first non-planar class that really makes use of a surface's (rational) parameterization. Remember that the quadrical class simulates the parameter space by projection, while the one representing geodesic arcs on the unit sphere relies on vectorial directions; see [FHS08].

We first shortly introduce Dupin cyclides, along with a rational parameterization, then show how we provide a suited geometric-traits class that does not assume generic position, followed by details on how to consistently construct the DCEL with the help of a cyclidean model of CGAL's *ArrTopologyTraits_2* concept. This finally leads to an implementation of an algorithm to construct and overlay arrangements on a cyclide. We conclude with experimental results.

Dupin cyclides have been introduced by Dupin as surfaces whose lines of curvature are all circular [Dup22]. Later, the usage of the term *cyclide* has switched for quartic surfaces that contain a circle at infinity as double curve [For12]. Since then, Dupin's surfaces are explicitly tagged with his name, namely *Dupin cyclides*. We only refer to the original definition. Hence, and for short notation, we always simply refer to *cyclides*. One can

imagine a (ring) Dupin cyclide as a torus with variable, but positive,[42] tube radius. Dupin cyclides are the generalization of the *natural* geometric surfaces like planes, cylinders, cones, spheres, and tori. Due to this fact they are privileged for applications in solid modeling; see, for example, [CDH89], [Pra90], [Boe90], [Joh93], [Pra95].

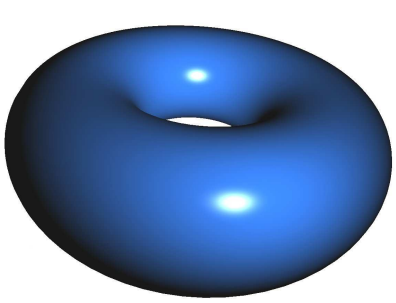
The following parameterization already appears more detailed in [Büh95, §1], while a quite intuitive construction of a (Dupin) cyclide is due to Maxwell, who we cite from Boehm [Boe90]:

> "Let a sufficiently long string be fastened at one end to one focus of an ellipse, let the string be kept always tight while sliding smoothly over the ellipse, then the other end sweeps out the whole surface of a cyclide Z."

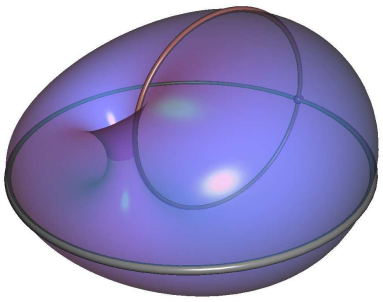
Observe that a torus is yield if the ellipse is actually a regular circle. For simplicity of presentation, we assume that a cyclide is in *standard position and orientation*, that is, the chosen base ellipse is defined by

$$(x/a)^2 + (y/b)^2 = 1, \quad a \geq b > 0$$

Figure 4.22. Two examples of ring Dupin cyclides



(a) $a = 2$, $b = 2$, $\mu = 1$



(b) $a = 13$, $b = 12$, $\mu = 11$. We indicate outer circle, tube circle, and pole; see below.

All cyclide pictures are produced with `xsurface` that is based on CGAL's planar curve renderer [Eme07]. The author thanks Pavel Emeliyanenko for his contribution.

For our practical realization, below, we allow cyclides to be translated or even rotated by a rational matrix. Three parameters uniquely define the cyclide in standard position: a and b determine the base ellipse, while μ helps to encode the length of the *string* given by $\mu - a$. However, choosing arbitrary values for these parameters, may also lead to cyclides that contain self-intersections, that are currently beyond the scope of our work.[43] We define $c = \sqrt{a^2 - b^2}$, which represents the distance between the focus and the center of

[42] If the radius would drop to zero at one position, we would get the disallowed croissant surface; see Figure 4.4.

[43] Self-intersections of surfaces are not (yet) handled by CGAL' `Arrangement_on_surface_2` framework.

the ellipse. In combination with μ it allows to distinguish three types of cyclides; see also [Bez07].

$0 < \mu < c$ In this case, the cyclide has two pinch points and is called *horned cyclide*. Such a surface looks like a torus with two contractions (i. e., the union of two surfaces topological equivalent to spheres, but touching at two isolated points; none is inside the other).

$c < \mu < a$ In this case, the cyclide looks like a squashed torus. Such a surface is free of (real) pinch points. It is called *ring cyclide*. Its shape looks like a closed tubical loop of variable radius; see Figure 4.22 for two examples. *We focus on such cyclides for our work.*

$a < \mu$ This relation results in a *spindle cyclide*. The resulting surface contains again two pinch points that connect two components that are topologically equivalent to spheres. In contrast to a horned cyclide, one of these components is "in the interior" of the other (except for the touching points).

$\mu = c,\ \mu = a$ These cases form intermediate degenerate cases (e. g., $\mu = a$ is a surface with a single pinch point) that are (currently) of no special interest for our objectives.

For more details on the classification of cyclides (there are, e. g., also *parabolic* cyclides), we refer to [CDH89] and, for a quick overview, to [17].

Very important for us is that ring Dupin cyclides are rational surfaces; see Definition 2.32. Several parameterizations exists. The following goes back to Forsyth [For12]. He proposed two alternative implicit equations of the regular cyclide (torus). The non-torus case is a natural extension of the following.

$$(x^2 + y^2 + z^2 - \mu^2 + b^2)^2 = 4(ax - c\mu)^2 + 4b^2y^2 \tag{4.1}$$
$$(x^2 + y^2 + z^2 - \mu^2 - b^2)^2 = 4(cx - a\mu)^2 - 4b^2z^2 \tag{4.2}$$

It is easy to prove that the intersection of the cyclide with the plane $y = 0$ results in two circles [Joh93]

$$(x + a)^2 + z^2 = (\mu + c)^2 \tag{4.3}$$
$$(x - a)^2 + z^2 = (\mu - c)^2 \tag{4.4}$$

and the intersection with $z = 0$ are the two circles

$$(x + c)^2 + y^2 = (a + \mu)^2 \tag{4.5}$$
$$(x - c)^2 + y^2 = (a - \mu)^2 \tag{4.6}$$

As we are considering the case of a ring cyclide, we always have that the interiors of of (4.3) and (4.4) are disjoint, and that the circle (4.6) is fully contained in the interior of (4.5).

A (trigonometric) parameterization of the cyclide is given by

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix} \mapsto \begin{pmatrix} \frac{\mu(c - a\cos\alpha\cos\beta) + b^2\cos\alpha}{a - c\cos\alpha\cos\beta} \\ \frac{b(a - \mu\cos\beta)\sin\alpha}{a - c\cos\alpha\cos\beta} \\ \frac{b(c\cos\alpha - \mu)\sin\beta}{a - c\cos\alpha\cos\beta} \end{pmatrix}$$

with $\alpha, \beta \in [-\pi, \pi]$.

Special diligence is required for the boundaries of the parameter space.

Lemma 4.14. *If $\alpha = \pi$ or $(\alpha = -\pi)$ is fixed, the parameterization above yields the circle $(x+a)^2 + z^2 = (\mu + c)^2$. If $\beta = \pi$ (or $\beta = -\pi$) is fixed, it yields the circle $(x+c)^2 + y^2 = (a+\mu)^2$. We call these circles tube circle and outer circle, respectively.*

Proof. Fix $\alpha = \pi$, which yields to the parameterization

$$\beta \mapsto \begin{pmatrix} \frac{\mu(c+a\cos\beta)-b^2}{a+c\cos\beta} \\ 0 \\ \frac{-b(c+\mu)\sin\beta}{a+c\cos\beta} \end{pmatrix}$$

Since the denominator does not vanish, this parameterizes a closed path in the plane $y = 0$, so it must be one of the circles (4.3) or (4.4). By setting $\beta = \pi$, we get the point $(-\mu - c - a, 0, 0)$, so it must be circle (4.3). The same argument can be used for $\beta = \pi$. □

The point $p := (-\mu - c - a, 0, 0)$ itself is special, as it is the intersection of the tube circle and the outer circle. We refer to it as the *pole* of the cyclide.

By now, the parameterization is trigonometric. However, we aim for a rational parameterization that allows to represent the intersection of an algebraic surface with Z as planar algebraic curve. We use the standard trick to get rid of the trigonometric functions (compare [Gal01]) using the following identities:

$$\cos\theta = \frac{1 - \tan^2\frac{\theta}{2}}{1 + \tan^2\frac{\theta}{2}} \qquad \sin\theta = \frac{2\tan\frac{\theta}{2}}{1 + \tan^2\frac{\theta}{2}}$$

If we now set $u := \tan\frac{\alpha}{2}$ and $v := \tan\frac{\beta}{2}$, we obtain

$$\begin{array}{rcl} \mathring{P} : \mathbb{R}^2 & \to & \mathbb{R}^3, \\ \begin{pmatrix} u \\ v \end{pmatrix} & \mapsto & \begin{pmatrix} \frac{\mu(c(1+u^2)(1+v^2)-a(1-v^2)(1-u^2))+b^2(1-u^2)(1+v^2)}{a(1+u^2)(1+v^2)-c(1-u^2)(1-v^2)} \\ \frac{2u(a(1+v^2)-\mu(1-v^2))b}{a(1+u^2)(1+v^2)-c(1-u^2)(1-v^2)} \\ \frac{2v(c(1-u^2)-\mu(1+u^2))b}{a(1+u^2)(1+v^2)-c(1-u^2)(1-v^2)} \end{pmatrix} \end{array}$$

Observe, that the image of $\mathring{P}$ is the cyclide without the tube circle and the outer circle. To close this gap, we set $\alpha = \pi$ (or $\beta = \pi$) and apply the same trick. This yields rational parameterizations of the tube circle and of the outer circle. Alternatively, we also get these circles by taking the limit of $\mathring{P}$ when $u \to \pm\infty$ ($v \to \pm\infty$), that is, we could consider an (implicit) compactification of $\mathbb{R}^2$ as $U \times V$.

There is also a geometric intuition behind this parameterization. We can think of cutting the cyclide along the outer circle and tube circle and "roll out" the surface to cover the plane. Thus, we also refer to the outer circle and the tube circle of a cyclide as its *cut circles*.

Note that there also exists other parameterizations of the cyclide that do not roll it out to the whole plane, but only to a bounded space [Bez07]. However, what follows does not benefit from such a parameterization, in fact, we later re-interpret infinity which simplifies matters.

Internally, we deal with a homogeneous parameterization of the cyclide, that is, the non-zero denominator can be written as a separate variable. Define $u_+ := 1+u^2$, $u_- := 1-u^2$, $v_+ := 1+v^2$ and $v_- := 1-v^2$:

$$\hat{P} : \mathbb{R}^2 \to \mathbb{P}^3_{\mathbb{R}}, \begin{pmatrix} u \\ v \end{pmatrix} \mapsto \begin{pmatrix} \mu(cu_+v_+ - au_-v_-) + b^2u_-v_+ \\ 2u(av_+ - \mu v_-)b \\ 2v(cu_- - \mu u_+)b \\ au_+v_+ - cu_-v_- \end{pmatrix}$$

Homogenization also applies for the outer circle

$$\hat{PO} : \quad \mathbb{R} \to \mathbb{P}^3_{\mathbb{R}}, \quad u \mapsto \begin{pmatrix} \mu(cu_+ + au_-) + b^2u_- \\ 2u(a+\mu)b \\ 0 \\ au_+ + cu_- \end{pmatrix}$$

and the tube circle

$$\hat{PT} : \quad \mathbb{R} \to \mathbb{P}^3_{\mathbb{R}}, \quad v \mapsto \begin{pmatrix} \mu(cv_+ + av_-) - b^2v_+ \\ 0 \\ -2v(c+\mu)b \\ av_+ + cv_- \end{pmatrix}$$

Finally, we also write the pole in homogeneous coordinates. Note that $\hat{p}$ indeed represents p, since $b^2 = a^2 - c^2$.

$$\hat{p} := \begin{pmatrix} -\mu(a-c) - b^2 \\ 0 \\ 0 \\ a-c \end{pmatrix}$$

We eventually consider as parameterization of Z the function φ_Z whose parameter space Φ is the compactified plane $\mathbb{R}^2$. The function φ_Z is combined from $\hat{P}$, $\hat{PO}$, $\hat{PT}$, and $\hat{p}$. Φ has interesting conditions on its boundaries. Namely, we detect identification of both opposite pairs of boundaries. More precisely, $\forall v \in V, \varphi_Z(u_{\min}, v) = \varphi_Z(u_{\max}, v)$ and $\forall u \in U, \varphi_Z(u, v_{\min}) = \varphi_Z(u, v_{\max})$, so for each point on the outer- and the tube-circle there exist two pre-images in parameter space. For the pole we even see four such. We have to deal with these identifications. For example, when we sweep with a circle of variable radius along the tube of the cyclide, that is, the image of the line $u = u_s$ under φ_Z. Two goals must be achieved: First, we require a unique order of events in the parameter space. Second, for a point on the cyclide with multiple pre-images, we actually want to construct only one DCEL-vertex. How to tackle these two problems has abstractly been discussed previously. Practically, it is required to provide a suited geometric-traits class and a suited topology-traits class. We next present both and start with details on a geometric-traits class that allows to consider arrangements on a ring cyclide induced by algebraic surfaces intersecting Z. Below, we continue with particularities on a proper model of the *ArrTopologyTraits_2* concept required for the cyclidean topology.

The geometry

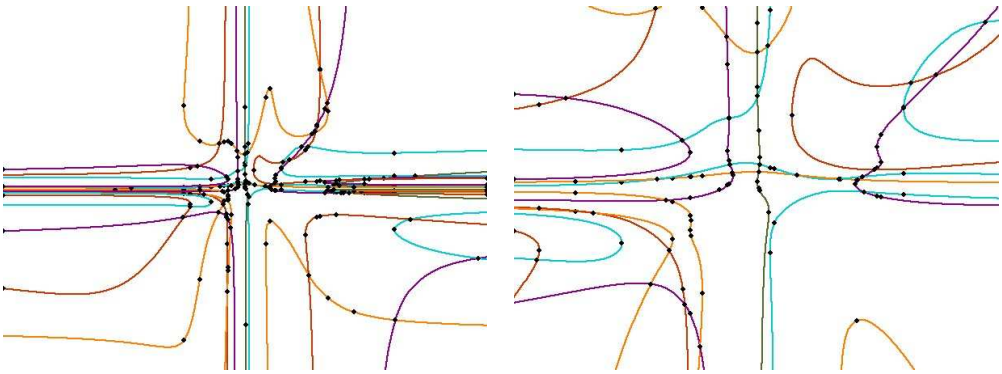
Consider the reference cyclide Z and an algebraic surface S_i intersecting it. We aim to represent the induced curve $Z \cap S_i$ as algebraic curve in the two-dimensional parameter space of Z. However, we have to deal with some peculiarities when interpreting a curve in the parameter space as "existing on the cyclide".

Let $g_i \in \mathbb{Z}[x, y, z]$ be the defining polynomial of surface S_i, with total degree D_i. We denote with $\hat{g}_i$ the homogenization of g_i.

Lemma 4.15. *The vanishing set of $f_i := \hat{g}_i(\hat{P}(u, v)) \in \mathbb{Z}[u, v]$ parameterizes the intersection points of g_i with the cyclide without those at the cut circles.*

Proof. By definition, the vanishing set of $g_i(\mathring{P}(u, v))$ in $\mathbb{R}^2$ defines the intersection curve of g_i and $\mathring{P}$ away from the cut circles. On the other hand, $g_i(\mathring{P}(u, v)) = 0$ if and only if $f_i = \hat{g}_i(\hat{P}(u, v)) = 0$. □

Figure 4.23. Two cut-outs of an arrangement in the planar parameter space of a cyclide. It is induced on the surface by 5 intersecting surfaces of degree 3 and consists of 208 vertices, 314 edges, and 107 edges. rendered with [7]



(a) Overview **(b)** Closeup view

That is, for a set of input surfaces $g_1, \ldots, g_n$ intersecting the cyclide, we obtain a set of real algebraic curves in the parameter space of the cyclide defined by polynomials $f_1, \ldots, f_n \in \mathbb{Z}[u, v]$. Figure 4.23 shows an example of such curves. This way we reduced the geometric part of the arrangement computation on the cyclide to a geometric part of an arrangement computation in the plane. However, this still requires to compute an arrangement of algebraic curves embedded in the real plane. The curves we have to consider have a relative high degree. Correctly, they reach bidegree $(2 \cdot \deg(g_i), 2 \cdot \deg(g_i))$. As we allow the g_i to have arbitrary degree, we require a model of CGAL's *ArrangementTraits_2* concept that supports algebraic curves in $\mathbb{R}^2$ of any degree in order to compute the induced planar arrangements. Such a model is given by CGAL's `Curved_kernel_via_analysis_2` (see §2.4.4), if instantiated with CGAL's `Algebraic_curve_kernel_2` provided by Eigenwillig and Kerber [EK08a]; we call this planar traits `Curved_kernel_via_analysis_2< ACK_2 >`, or `CK_2` for short. Details about the efficiency of the used algebraic kernel are collected in §2.3.3. We only remember, that the non-avoidable symbolic computations in the kernel (computation of subresultant sequences), actually limits its usability for curves of higher

degree; and thus for surfaces intersecting the cyclide. The planar kernel assumes no conditions on the input. Covertical events, vertical asymptotes, and singularities poses no problem for the outcome of expected analyses of curves and pairs of them. Only running time can be affected by such degeneracies. For example, some cases require a linear change of coordinates (i. e., shear) with a subsequent back-shear step in order to report the results with respect to the original coordinate system. Nevertheless, we can conclude that no conditions on the algebraic surfaces g_i intersecting the cyclide are imposed.

Remark. There might be other parameterizations of the cyclide that lead to curves f_i of smaller (bi-)degree, which would also show that $\hat{P}$ results in curves of non-optimal degree. However, it is unknown whether such a parameterization (if existing) is applicable for our purpose. In particular, it must be checked whether the chosen implementation still works, and if so, which modification are expected.

Representation The `CK_2` itself is a model of the *UnboundedBoundaryTraits* concept in both variables; see hierarchy in Figure 4.8. We have to adapt it with respect to the cyclidean topology. We next show how to turn it into a model, called

`Arr_surfaces_intersecting_dupin_cyclide_traits_2`

and fulfilling the *IdentifiedBoundaryTraits* concept; again in both variables. For simplicity, we refer to it as the `Cyclide_geo_traits_2`.

The `Cyclide_geo_traits_2` is derived from `CK_2`. An instance is constructed from a given reference cyclide, which is stored as the traits' *status*. The first required modification is the redefinition of the nested `Curve_2` to `Algebraic_surface_3`,[44] that is, the type of algebraic surfaces. This redefinition implies also an adaption of the model's `Make_x_monotone_2` functor,[45] which splits an instance of type `Curve_2` into instances of type `Point_2` and `X_monotone_curve_2`. At this point, we mention that points and arcs on the cyclide are represented with respect the cyclide's parameter space. This also explains the derivation of `Cyclide_geo_traits_2` from `CK_2`. Thus, the realization of `Make_x_monotone_2` is two-step. First, we apply for the given surface Lemma 4.15. This requires access to the stored reference cyclide. Second, we decompose the resulting planar curve into (weakly) x-monotone arcs and isolated points using `CK_2`'s version of `Make_x_monotone_2`. Observe, that we do not need to derive specialized classes for `Cyclide_geo_traits_2`'s `Point_2` and `X_monotone_curve_2` types. Even the assignment to the boundaries of the parameter space keeps valid, with the difference that we now interpret the infinite boundaries as identifications.

Remark (Points and curves on cut circles). Actually, there is one subtlety in this interpretation. Isolated points and curves fully embedded in one of the cut circles cannot be represented with the `CK_2`'s point and curve type. Remember that such objects have multiple pre-images in the cyclide's parameter. `CK_2` is not expected to represent such objects at infinity, while `Cyclide_geo_traits_2` re-interprets the compactification of $\mathbb{R}^2$ as being on the surface of the cyclide. However, although theoretically described how to deal with events related to such special points and arcs (see §4.2), the completion of the

[44] `typedef Algebraic_surface_3 Curve_2;`

[45] Observe for this part of the text that the geometric-traits class uses the variable names x and y, while in our case we actually refer to u and v.

`Arrangement_on_surface_2` package with respect to such objects is planned for the future. Once this objective is reached, derived `Cyclide_point_2` and `Cyclide_x_monotone_curve_2` classes with *specialized constructors* become required.

Anyhow, let us mention that for a complete surface g_i, the formal leading coefficients of the resulting polynomial f_i already encodes some special intersections with respect to the cut circles of Z. Observe that $\deg_{\text{total}}(f_i) \leq 4n$, $\deg_u(f_i) \leq 2n$ and $\deg_v(f_i) \leq 2n$.

Lemma 4.16. *Let* $\operatorname{coef}(f_i, x_h, r) \in \mathbb{R}[x_1, \ldots, x_{h-1}, x_{h+1}, \ldots, x_n]$ *denote the coefficient of* f *in* x_h^r. *Then, we have*

$$\begin{aligned} \hat{g_i}(\hat{PT}(v)) &= \operatorname{coef}(f_i, u, 2D_i) \\ \hat{g_i}(\hat{PO}(u)) &= \operatorname{coef}(f_i, v, 2D_i) \\ \hat{g_i}(\hat{p}) &= \operatorname{coef}(\operatorname{coef}(f_i, u, 2D_i), v, 2D_i). \end{aligned}$$

Proof. The function $\operatorname{coef}(\,\cdot\,, x_h, r)$ is linear. Thus, it suffices to show the equality for the case that $\hat{g} = x^{d_x} y^{d_y} z^{d_z} w^{d_w}$ is a monomial with $d_x + d_y + d_z + d_w = D_i$. We show the first part of the lemma, while the two remaining statements follow similar arguments.

Since for $d_y > 0$, $\hat{g_i}(\hat{PT}(v)) = 0$, and also, $\deg_u(f_i) < 2D_i$, we can assume that $d_y = 0$. Let $\hat{P}_1, \ldots, \hat{P}_4$ denote the polynomials of $\hat{P}$'s parameterization. Then, we have

$$\operatorname{coef}(f, u, 2D_i) = (\operatorname{coef}(\hat{P}_1, u, 2))^{d_x} (\operatorname{coef}(\hat{P}_3, u, 2))^{d_z} (\operatorname{coef}(\hat{P}_4, u, 2))^{d_z},$$

and comparing this with $\hat{g_i}(\hat{PT}(v))$ yields the desired equality. □

Lemma 4.16 also has a geometric interpretation, namely it shows that isolated intersection points on the cut circles appear as real roots of $\operatorname{coef}(f_i, u, 2D_i)$ or $\operatorname{coef}(f, v, 2D_i)$. In addition, it is possible to detect special intersections with the cyclide.

Corollary 4.17.

- $\deg_u(f_i) < 2D_i$ *if and only if* g_i *and* Z *intersect in the whole tube circle of* Z.
- $\deg_v(f_i) < 2D_i$ *if and only if* g_i *and* Z *intersect in the whole outer circle of* Z.
- $\deg_{\text{total}}(f_i) < 4D_i$ *if and only if* g_i *and* Z *intersect in the pole of* Z.

This information can be used in the future when constructing special representations for points and arcs embedded in the cut circles. We remark that computing the degrees is a cheap task, while the root isolation is performed anyway, namely when determining the asymptotes of f_i below. We already encourage to cache such information in an actual implementation.

Predicates and constructions Besides the geometric representation, we also expect from the `Cyclide_geo_traits_2` class to provide geometric predicates and constructions. Not any modification of the `CK_2` is required to predicates that relate to the interior of the parameter space. First, remember that the `Arrangement_on_surface_2` package cleverly combines the outcome of a set of comparisons of near (or on) the boundaries in order to obtain a unique order for the sweep line events. In particular, the geometric-traits class is asked for the horizontal or vertical alignment of two curve-ends infinitesimally away

from a boundary. In our case, the order of curve-ends approaching a cut circle is encoded by the order of the corresponding curve-ends in parameter space approaching infinity. Thus, we again only re-interpret `CK_2`'s existing functors `Compare_x_near_boundary_2` and `Compare_y_near_boundary_2` that compare curve-ends approaching infinity in parameter space as functors that compare curve-ends approaching a cut circle.

However, some functors have to explicitly care about the boundary of the parameter space. The prominent among them are the ones demanded by the *IdentifiedBoundaryTraits* concept, in particular, `Compare_x_on_boundary_2` and `Compare_y_on_boundary_2`. Both must compare "points" that are lying at infinity in the parameter space. To simplify, we can assume, that we consider curve-ends of unbounded arcs of a curve f_i. There are two representations for such an end:

- Either, the arc is asymptotic to a vertical line $u = u_0$, that is, it approaches the top- or bottom-boundary. Then, we know a symbolic endpoint $(u_0, f_i, \pm\infty)$. By Theorem 2.24 we know that u_0 is a root of $\mathrm{lcf}_y(f_i)$. The order of two such points on the bottom-top-identification is given by the order of their u-values.
- Or, second, the arc approaches the left or right boundary, which means that its end is represented by a symbolic point $(\pm\infty, f_i, a_i)$, where a_i is the point's arc number on f_i. However, this information is not sufficient to compute the v-order of two such points, especially to detect their equality. Thus, we next show how to obtain more information on the symbolic endpoint of arcs that extend to $u = \pm\infty$. Such an arc can have a horizontal asymptote $v = v_0$. In this case it represents an arc on the cyclide that intersects the *interior* of the tube circle at $PT(v_0)$ and thus lies on the left or right boundary. Finally, it can also be unbounded in v as well. Then it converges to one of the four *corner points* $(\pm\infty, \pm\infty)$ in parameter space. On the cyclide, such an arc runs into the cyclide's pole.

For the further considerations on this second case, we restrict to a single algebraic plane curve f. In the actual realization of `Compare_y_on_boundary_2`,[46] we apply the following method to both curves currently in focus. It is well known, that an algebraic curve only has a finite number of easily computable horizontal asymptotes. Their v-values are defined as roots of the leading coefficient $\mathrm{lcf}_u(f)$; see Theorem 2.24.

This observation leads to an algorithm that assigns curve-arcs approaching $u = \pm\infty$ to the finite number of possible symbolic endpoints $(\pm\infty, v_l)$, $l = 0, \ldots, k+1$, where $v_0 = -\infty$ and $v_{k+1} = +\infty$, and $v_1 < \ldots < v_k$ denote the sorted real roots of $\mathrm{lcf}_u(f)(v)$. We next define $k+2$ *buckets* $(-\infty, q_0)$, (q_0, q_1), $\ldots$, (q_{k-1}, q_k), (q_k, ∞) with the help of computed intermediate rational values $q_0, \ldots, q_k$ with $v_l < q_l < v_{l+1}$ for all $l \in \{0, \ldots, k\}$. Observe that each bucket (q_l, q_{l+1}) contains exactly v_l. The handling of the left and the right side boundary are similar, thus, we restrict Algorithm 4.1 for simplicity to the left case.

[46] Observe the naming $v \equiv y$.

Algorithm 4.1. Assign arc numbers of curve to non-vertical asymptotes

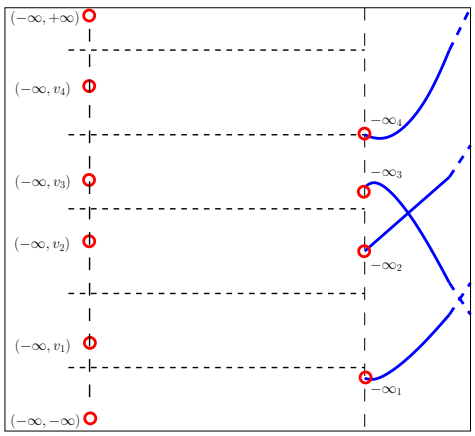
INPUT: Plane algebraic curve f

OUTPUT: Assignment which arcs number of f at $u = -\infty$ correspond to which non-vertical asymptote of f.

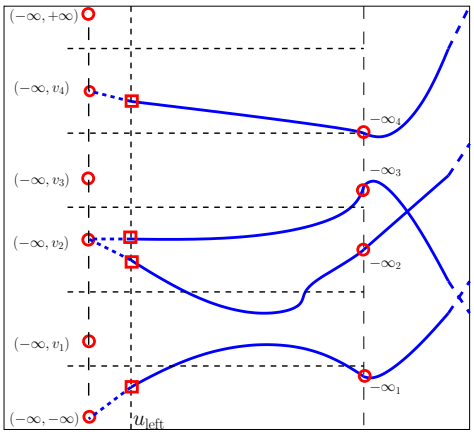
1. Choose a (rational) value u_r to the left of any *critical x-coordinate* of f (i. e., x-coordinates of f's singularities, f's x-extreme points or f's vertical asymptotes are critical). The required u_r is easy to compute, as f's analysis is aware of all of its critical x-coordinates.
2. Next, compute
$$u_{\text{left}} := \min\{u_r, \min_{l=0,\ldots,k} \min\{\mu \mid f(\mu, q_l) = 0\}\}$$
by isolating the real roots of $f(x, q_l)$.
3. Finally, isolate the real roots $v'_1, \ldots, v'_k$ of $f(u_{\text{left}}, v)$, and determine with interval refinements the bucket each v'_l falls into. This gives the desired assignment.

An illustration of Algorithm 4.1 is given in Figure 4.24. Theorem 4.18 gives the correctness of the algorithm. In our implementation, we do not use the algebraic number u_{left}, but a rational value to its left. This choice still ensures the correct assignment.

Figure 4.24. Connecting arcs with non-vertical asymptotes taken from [BK08]



(a) Symbolic endpoints for the left ends of the curve, and the buckets of the curve.



(b) Roots of the curve for a u_{left} that is to the left of any bucket change. Information about non-vertical asymptotes can be read off directly.

Theorem 4.18. *Let the v'_l of $f(u_{\text{left}}, y)$ be in the bucket of v_j. Then, the l-th arc of f with $u \to -\infty$ converges to $(-\infty, v_j)$.*

Proof. Since $u_{\text{left}} < u_r$, v'_l lies on the l-th arc of f that goes to $u = -\infty$. Moreover, u_{left} is smaller than any root of $f(x, q_h)$, $h = 0, \ldots, k$. It follows that f does not intersect any line $x = q_h$ on the left of u_{left}. Consequently, the l-th arc of f cannot change the bucket anymore to the left of u_{left}. So, $(-\infty, v_j)$ is the only possible end of the arc. □

All other "planar" functors only need small wrappings in order to work "on the cyclide". For example, each end of a curve is now finite, or `Intersect_2` also has to report intersections on the boundary, which again requires to detect whether two arcs have the same asymptote. The computation of the vertical alignment of two curves right (left) of an intersection point must also be adapted if the intersection lies on the boundary. Usually, a proper call of `Cyclide_geo_traits_2`'s functor `Compare_y_near_boundary_2` gives the answer, except for the pole that requires to use the information whether the arcs actually approach the "bottom corner points" or "top corner points" of the parameter space.

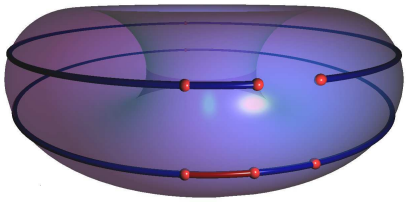
The topology

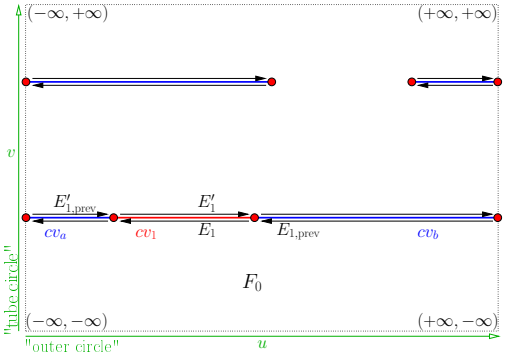
As for a quadric, the topology of a cyclide requires special attention. We already remarked on the existence of two identifications in its parameter space. Our cyclidean topology-traits class (`Arr_dupin_cyclide_topology_traits_2`) is aware of these specialties with respect to this surface of genus one.

The initial DCEL of an empty arrangement on a cyclide consists of a single *bounded* face that has neither an inner nor an outer CCB. We are going to implement the forest-strategy for this traits class. For each identification we maintain a sorted list of DCEL-vertices, or more intuitively: One for each cut circle. Their order is determined by `Compare_x_on_boundary_2` and `Compare_y_on_boundary_2` provided by our new cyclidean geometric-traits class. The functors compare the parametric values of points on the cut circles, that is, according to $\hat{PO}$ and $\hat{PT}$. The localization of vertices on the boundary (with the help of `place_boundary_vertex` and `locate_curve_end`) is again feasible. It only requires to perform a binary search in the correct list. Either, a vertex is found and reported, or `NULL` is returned. If so, the found position is used for the subsequent update operation triggered by `notify_on_boundary_vertex_creation`. This way, the arrangement itself is responsible to construct vertices, while the topology-traits class keeps the control for DCEL-records on the boundary. This process forms an important part of the on-line realization of the existing identifications. For the deletion of a vertex the process is similar. Again, the localization of a curve in the circular list of incident curves around a vertex is performed with the help of `Arrangement_on_surface_2`-internal functor `Is_between_cw_2` that returns `true` if a curve is counter-clockwisely in between two curves meeting at the same point.

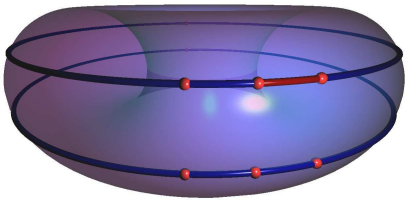
An instance of `Arr_dupin_cyclide_topology_traits_2` also monitors whether the insertion or deletion of a curve implies a face split or a hole creation. We already discussed in §4.4.3 which cases can occur. We remember that we have to detect the *first perimetric loop* L_1 and to select which curve of identification is crossed by L_1 an odd number of times. Upon this detection of L_1 by `face_split_after_edge_insertion`, it returns `std::pair< false, false >`. The value-pair triggers the special option (d) for the basic insertion function, that is, the initial face gets now bounded by two outer CCBs. Note that this exactly corresponds to what is expected by the forest-strategy. After L_1 is closed, the implementation of all further predicates with respect to faces and their CCBs (`face_split_after_edge_insertion`, `is_on_new_face_boundary`, `boundaries_of_same_face`) are identical to the cylinder case presented in §4.6.2. That is, we are left with an implicit single curve of identification, which we have to concentrate on when counting crossings of further directed loops. An illustration of the two first steps is given in Figure 4.25.

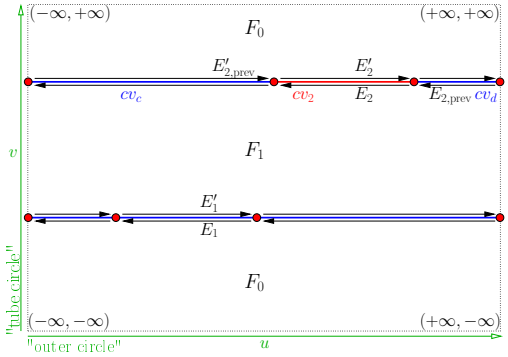
Figure 4.25. Closing loops on a cyclide. We start in (a) with a single bounded face F_0 that has two inner CCBs defined by $E_{1,\text{prev}}$ (or $E'_{1,\text{prev}}$) and $E_{2,\text{prev}}$ (or $E'_{2,\text{prev}}$). The views in parameter space (right) are schematic.





(a) Adding cv_1 (and thus E_1 and E'_1) splits the inner CCB of F_0 defined by $E_{1,\text{prev}}$ into two outer CCBs (defined by E_1 and E'_1). There is no face-split, due to the two identifications. However, F_0 is now surrounded by the two outer CCBs defined by $E_{1,\text{prev}}$ and $E'_{2,\text{prev}}$.





(b) Adding cv_2 (and thus E_2 and E'_2) splits the inner CCB defined by $E_{2,\text{prev}}$ into two outer CCBs (defined by E_2 and E'_2). Now there are two perimetric faces F_0 and F_1. Each has two outer CCBs: F_0's CCBs are defined by E_1 and E'_2, F_1's CCBs are given by E_2 and E'_1. The outer CCBs have different directions and different non-zero signs. There are no more inner CCBs.

Remark. The remaining methods of `Arr_dupin_cyclide_topology_traits_2` are either simple to implement or related to fictitious edges, that, again, do not occur for this topology.

The cyclidean topology-traits is also obliged to provide some nested types, namely the visitor classes required for arrangement construction, insertion, and overlay via the sweep line approach, a visitor class for the incremental construction, and the default point location strategy. For most of them generic templates exists. As for quadrics, each must only be adapted with a surface-specific helper classes: For example, the helper for the construction via sweep line is responsible to pre-process events, namely to assign the correct vertices to each, which finally helps to construct and insert curves that emanate to the right of an event. In addition, it maintains a list of curves that only see the top boundary above them. The relocation of holes after splitting a face relies on this information. The remaining helpers and classes are very similar to the quadrical case; see §4.6.1.

Results

With the presented traits classes, we can successfully, robustly, and efficiently construct arrangements on Dupin cyclide using CGAL's `Arrangement_on_surface_2` class-template. An example is given in Figure 4.26.

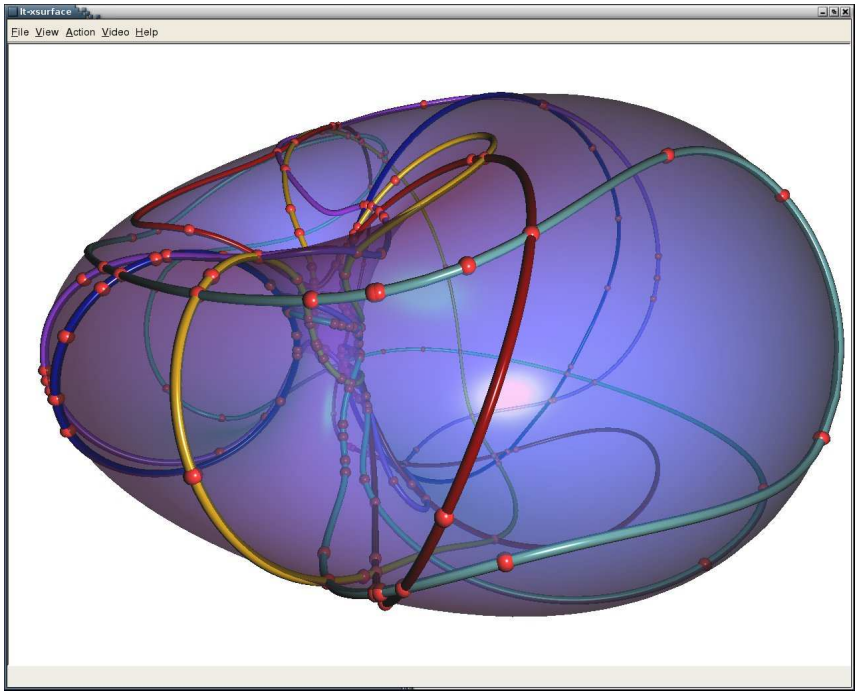


Figure 4.26. The shown arrangement on a cyclide is induced by 5 algebraic surfaces of degree 3 intersecting the reference surface. It consists of 240 vertices, 314 edges, and 74 faces. It is visualized with `xsurface` by Pavel Emeliyanenko.

We also run experiments to check that this approach does not lack efficiency. All test are executed on an AMD Dual-Core Opteron(tm) 8218 multi-processor Debian Etch platform,

Instance	#S	#V	#E	#F	t (onCyclide)	t (onPlane)
ipl-1	10	119	190	71	**0.14**	0.14
ipl-1	20	384	682	298	**0.58**	0.58
ipl-1	50	1837	3363	1526	**2.14**	2.00
ipl-2	10	358	575	217	**1.07**	1.25
ipl-2	20	1211	2147	937	**3.14**	3.04
ipl-3	10	542	847	305	**4.84**	4.62
ipl-3-6points	10	680	1092	412	**32.43**	31.17
ipl-3-2sing	10	694	1062	368	**5.82**	5.57
ipl-4	10	785	1204	419	**50.42**	49.97
ipl-4-6points	10	989	1529	540	**461.74**	450.54
ipl-4-2sing	10	933	1471	538	**53.01**	52.78

Table 4.4. Running times (in seconds) to construct arrangements on Z_1 induced by algebraic surfaces

Instance	#S	#V	#E	#F	t (onCyclide)	t (onPlane)
ipl-1	10	169	280	111	**0.53**	0.46
ipl-1	20	456	808	352	**0.86**	0.54
ipl-1	50	3228	6084	2856	**3.78**	3.33
ipl-2	10	450	710	260	**1.22**	1.21
ipl-2	20	1323	2247	924	**3.44**	3.57
ipl-3	10	474	682	208	**5.24**	5.36
ipl-4	10	988	1406	418	**50.93**	52.43

Table 4.5. Running times (in seconds) to construct arrangements on Z_2 induced by algebraic surfaces

each core equipped with 1 MB internal cache and clocked at 1 GHz. The total memory consists of 32 GB. As compiler we used `g++` in version 4.1.2 with flags `-O2 -DNDEBUG`. Two results were obtained for each instance. First, we computed the arrangement using the *cyclidean topology* (`onCyclide`). Second, we computed the two-dimensional arrangement of the induced intersection curves *in uv-parameter space*, that is, with the topology of an unbounded plane (`onPlane`).

Our implementation allows to transform a cyclide in standard position and orientation, that is, to translate it by a vector and to rotate it with respect to a rotational matrix with rational entries. In our tests, we used two different reference cyclides. First, the *standard torus* Z_1 with $a = 2$, $b = 2$, $\mu = 1$, centered at the origin with no applied rotation. Second, a non-torical cyclide Z_2 with $a = 13$, $b = 12$ and $\mu = 11$, centered at $(1, 1, 1)$ and a rotation defined by the matrix

$$\frac{1}{3}\begin{pmatrix} 2 & -2 & 1 \\ 2 & 1 & -2 \\ 1 & 2 & 2 \end{pmatrix}$$

Our first class of test examples are surfaces of fixed degree which interpolate randomly chosen points on a three-dimensional grid, having no or some degeneracies with respect to Z_1: the surfaces in "6points" instances share at least 6 common points on Z_1, one of

Instances	#S	#V,#E,#F	t
quadrics	10	428,646,219	**1.59**
degree-3	5	240,314,74	**1.56**
Overlay	-	942,1508,566	**1.91**
degree-3	10	794,1218,424	**6.25**
degree-4	10	325,418,93	**13.36**
Overlay	-	1623,2644,1021	**13.83**
degree-4	10	816,1188,372	**50.86**
degree-4	5	325,418,93	**13.52**
Overlay	-	1581,2488,907	**47.30**

Table 4.6. Running times (in seconds) to construct arrangements induced by algebraic surfaces of different degree on Z_2, and to overlay them afterwards

them is the pole of Z_1. The surfaces in the "2sing" instances induce (at least) two singular intersections on Z_1.

Our obtained running times are listed in Tables 4.4 and 4.5. For such random examples, our algorithm shows a good general behavior, even for higher degree surfaces. Degeneracies with respect to the reference surface result in higher running times as the instance "6points" shows. But this effect already appears in parameter space; we remark on the similar running times in the `onPlane`-column. In general, it is observable and remarkable that in all tested instances, the spent time on the cyclides is (almost) identical to the computation of the curves in their parameter space. This allows to conclude two results:

1. The performance of our implementation is not harmed by the cyclidean topology-traits class, that is, the cyclidean model is as efficient as the topology-traits class for the unbounded plane.
2. The additionally required computation of horizontal asymptotes seems (as expected) to be a cheap task. Most time is spent for geometric operations on algebraic curves.

Thus, we infer that the chosen approach strongly hinges on the efficiency of the underlying implementation for arrangements of algebraic plane curves, in particular the (bivariate) algebraic kernel, and conclude the parametric ansatz to be successful in its idea.

The cyclidean topology-traits also provides the visitor classes for various sweep line construction, in particular the model that enables the `Arrangement_on_surface_2`'s overlay mechanism. That is, we are able to overlay two arrangements on the same cyclide by using the capabilities of generic programming. Therefore, we also generated instances of random surfaces with degree up to 4 intersecting Z_2, picked two of them, computed their arrangement and finally overlaid them. A selection of such combinations along with the sizes of the resulting arrangements and running times is presented in Table 4.6. We remark, that due to persistent caching, the times for the overlay are usually less than the sum of the times required to create the two originating arrangements. The reason is simply that during the overlay only some additional pairs of algebraic curves have to be newly created.

We should also mention that the localization of a point (given by its parametric coordinates) in an arrangement on the cyclide is supported by the `Arrangement_on_surface_2` package. We obtain the cell of the arrangement that contains the point. Again, the dependency on the planar backup is expected to be the bottleneck.

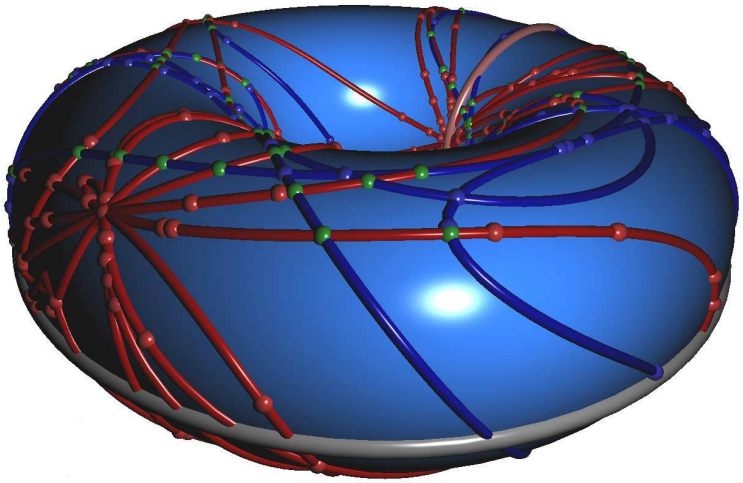


Figure 4.27. Overlay of two arrangements: The red is induced by five surfaces of degree 3 that induced degeneracies on the torus. The blue is induced by five other surfaces of degree 2. Overlay intersections are shown in green.

4.7. Conclusion and outlook

Achievements We have seen how to construct and maintain two-dimensional arrangements on parametric surfaces. We pay special attention to code reuse. In particular, we revised the abstraction of main arrangement-related algorithms and data structures from basic geometric operations and extracted new abstractions with respect to surface-specific topological operations. This "parameterization" simplifies the development of traits classes for handling new families of curves and new surface topologies in a straightforward manner. Such extensions benefit from a highly efficient (and well-tested) code base for the main arrangement-related classes.

Beyond a rough overview of existing traits classes, we discussed two concrete examples of surface families in their details, namely elliptic quadrics and ring Dupin cyclides. For both we provide valid models of the new *ArrTopologyTraits_2* concept. Their implementations are family-specific, however they also share basic ideas. We also provide geometric-traits classes that allow to compute arrangements on such reference surfaces, induced by their intersections with other quadrics or even algebraic surfaces. Both classes cleverly, but differently, modify a model that originally suites for planar algebraic curves only. The enhancement "lifts" the planar curves on the reference surface itself. In both cases, the applied changes do not significantly harm the efficiency of the approach, that is, the performance of the traits classes for arrangements on quadrics and ring Dupin cyclides is mainly determined by the efficiency of the underlying algebraic kernel that already supports the analyses of planar curves.

The chosen strategy also shows the power of generic programming. Developing, surface-specific traits-classes is a comparably small task compared to an implementation from scratch, not using the CGAL's `Arrangement_on_surface_2` framework and its algebraic kernels. This results in faster development time and less code to debug. In addition we benefit from advanced programming techniques applied to CGAL's `Arrangement_on_surface_2` package [WFZH07b]. In particular, there is immediate support for observers that notify on structural changes of the arrangement, or the possibility to extend the DCEL with data.

Future work Beyond what we have presented on the geometric-traits classes, there is room for further improvements. For example, it would be nice to allow algebraic surfaces of arbitrary degree intersecting an elliptic quadric. It is the lifting onto the lower or upper part that must be adapted. In fact, we present in Chapter 5 (in particular in §5.5.3) the required tools, to compute such space curves. For the case of the cyclide, we also believe that the performance could be further improved: We analyze the planar curves used to represent intersection of the cyclide with algebraic surfaces without any beneficial knowledge induced by the used parameterization. In particular, it is possible to simplify the one resultant whose roots define a curve's critical x-coordinates by a non-trivial factor. That is, the real-root isolation can actually deal with a much simpler polynomial. In addition, such planar curves often contain numerous vertically asymptotic arcs; see, for example, Figure 4.23. However, we use the strategy described in [EKW07], that is, to shear and to shear-back such non-regular curves. This step is expensive, in particular, if applied to a large fraction of the curves. A desirable goal is to develop a comparably efficient alternative approach that avoids to shear curves of this sub-class.

It would also be nice to consider more families of surfaces, in particular, if they provide a rational parameterization, as ring Dupin cyclides do. In principle, we can derive a similar version of a geometric-traits class that explicitly elaborates the parameterization of such surfaces. Of course a suitable model for the *ArrTopologyTraits_2* concept is also expected. However, in practice, the degrees of the algebraic curves in the parameter space constitutes our current limit of practical usability of the parametric approach.

Concerning the framework itself, we already proposed in theory how to deal with isolated points and curves fully embedded in the boundary of the parameter space. However, the code has not yet been adapted with respect to these ideas. This step is planned for the near future.

With introducing the *ArrTopologyTraits_2* concept, we successfully abstracted topological operations required to maintain a surface-specific DCEL from more generic arrangement-classes. However, the topology-traits classes for elliptic quadrics and ring Dupin cyclides (and even for the omitted one of the sphere) show some visible similarities. For example, all maintain a sorted sequence of points on an identification, and the decision with respect to face splits and their CCBs rely on similar information. As future direction, it should be analyzed, in how far a unified model can be established. Such a model can be configured with respect to various topologies, by constructing it, for example, by just naming what happens on the boundaries of the parameter space.

In this work, we also restricted ourselves to the single domain case, that is $\Phi = U \times V$. Another future goal is to extend the framework to handle general orientable surfaces, which can be conveniently represented by a collection of domains, each of which supported by a rectangular parameter space. It is known which polygonal maps give rise to orientable

surfaces and each orientable surface has a normal form, which already includes surfaces of higher genus. In addition, one might to consider surfaces with singularities (e. g., a double cone), which requires to decompose them such that singularities only appear on the boundary of parameter spaces. We give such a decomposition for algebraic surfaces in Chapter 5. Concerning the framework, the different individually obtained parameter spaces are glued together according to the topology of the surface and therefore will naturally be described in, and handled by, an extension of the *ArrTopologyTraits_2* concept. However, arrangements on surfaces with singularities cannot be represented with a usual DCEL. The reason is that a vertex can be incident to two faces at the same time. An example is the apex of a double-cone.

Arrangements on surfaces can also be a tool in other utilizations. For example, it can serve as basic support to compute the *adjacency graph* that is induced by a set of surfaces. This objective requires to identify equal vertices and edges on different surfaces. How to do this for quadrics has been shown in [Hem08]. The chosen approach uses a direct parameterization of the quadrics. However, the important subtask, namely the identification of vertices and edges can be formulated almost abstractly. Then, it should be possible to easily combine it with the `Arrangement_on_surface_2` package in order to compute adjacency graphs for all surfaces on which we can compute arrangements. It might be required to add another geometric primitive that robustly determines the equality of two vertices (and edges?) in the parameter spaces of two different surfaces. The construction of a fully-fledged three-dimensional arrangement *of* surfaces is the ultimate objective. Although it is beyond the scope of this thesis, we conclude that the (in combination with the adjacency-graph to compute) our contributions constitute major building blocks towards this goal.

5

Efficient Stratification of Algebraic Surfaces with Planar Arrangements

In this chapter, we increase the dimension by one and turn towards the topological and geometrical analysis of algebraic surfaces combining three main tools: Planar arrangements induced by algebraic curves, the bitstream Descartes method, and interval arithmetic. Our concern is beyond the theoretical design of a new algorithm, but aiming for a clever combination of existing tools to provide a robust and efficient implementation for the final problem:

Given a finite set $\mathcal{S} = \{S_1, \ldots, S_n\}$ of square-free primitive, and coprime algebraic surfaces in $\mathbb{R}^3$, defined by polynomials $f_i \in \mathbb{Q}[x, y, z]$, $i \leq 1 \leq n$, with $D_i = \deg_{\text{total}}(f_i)$ and $D := max_i(D_i)$. We are interested in the geometric and topological information to describe $\mathcal{S}$. So, we aim for a cell decomposition of the surfaces with respect to $\mathcal{S}$ into cells of dimension 0, 1, and 2. The cells should form smooth subvarieties of some S_i. We are also interested in how the cells are connected. In addition, the cells should share the *boundary property*, that is, the boundary of a single cell is formed by a union of other cells in the decomposition. Such a decomposition is also known as *stratification*, while a single cell is called *stratum*; see [BPR06, §5.5] and compare also with the CW complex that we present in §2.1.7. The obtained decomposition is similar to a clustered cylindrical algebraic decomposition of $\mathbb{R}^3$. Of course, we also allow that $n = 1$, which actually constitutes a special case.

The approach consists of three steps:

1. First, we *project* the z-critical points of $\mathcal{S}$ to compute an unbounded planar arrangement $\mathcal{A}_\mathcal{S}$ with a finite number of relatively open cells. Each cell shares some invariant properties for all of its points. In particular, they share the same z-pattern.
2. A z-pattern at some point p encodes the sequence of intersections of $S_i \in \mathcal{S}$ with the vertical line ℓ_p at p and is computed for each cell during the *lifting*. It suffices to compute a z-pattern only for a sample point of each cell of $\mathcal{A}_\mathcal{S}$. The lifting of the

sample points leads to our cell decomposition Ω_S.

3. As final step, we obtain the *adjacency relation* between the cells of Ω_S.

The approach is similar to Collins' cylindrical algebraic decomposition (cad); see §2.1.6. In each of these steps, we exploit methods that try to replace costly symbolic computations by combinatorial deductions and certified approximative solutions. We exemplary mention the bitstream Descartes method with its m-k-extension for the non-square-free case; see §2.1.2. In any case, we guarantee to reflect the mathematical correct stratification, as expected by the exact geometric computation paradigm (EGC); see §2.2.2. This is done by either certifying that the approximative filters compute the desired result, or eventually switching to an exact method. Our decomposition consists of $O(n^5D^5)$ many cells. It is possible to refine the decomposition into simply connected cells without compromising the final complexity.

We remark that our approach is free of assumptions on the input surface. Algorithmically, we never change the spatial coordinate system in order to prevent degeneracies. The (geometric) output is with respect to the original coordinate system. While this has advantages, for example to enable arbitrary dense sampling of the decomposition, it also means that degenerate situations must be handled, in particular vertical lines contained in a surface. To satisfy the boundary property of the cells in the decomposition, such lines must be decomposed further.

Our implementation is robust and efficient. To our knowledge it is the first EGC-software for the topological analysis of algebraic surfaces, including singular ones. As basic tool, we rely on arrangements of planar algebraic curves; see §2.3.3, §2.4.3, and §2.4.4. The code follows the generic programming paradigm, which allows to tackle the problem in two related parts: One constitutes a framework that extends a planar (unbounded) arrangements in order to support the lifting into the third dimension. The framework defines the new *SurfaceTraits_3* concept, that is, it expects from surfaces some types and operations. The concept breaks down the rather complex challenge into a small set of simple tasks demanded on surfaces, like to compute approximations of $S_i \cap \ell_p$ for some p. It is the accountability of the framework to combine the output of these operations to obtain the desired output.

We provide two models fulfilling the *SurfaceTraits_3* concept that form the second part of the implementation: One model for quadrics and one model for algebraic surfaces of any degree. The first benefits from the low degree of quadrics, while the second requires a more sophisticated handling to efficiently tackle the non-restricted input.

This way, the implementation decouples geometric operations and combinatorial information. The combinatorial output allows to consider various utilizations by other geometric algorithms, especially if restricting to such that only involve a small number of surfaces at a time.[47] The reason is that the complexity of Ω_S is $O(n^5D^5)$. We give a basic set of well-known examples: The framework supports the analysis and meshing of a single surface, the analysis and construction of a space curve defined by two surfaces, or the computation of the lower envelope of surfaces. It can also serve in the future as a key ingredient in a three-dimensional arrangement. Some of these applications are even already available as software.

We present experiments that show good performance. However, it must be remarked that the projection step of our approach defines a bound on the practical applicability

[47] Either the task is defined such, or each substep of the focussed algorithm involves only some surfaces.

for high-degree surfaces. The reason is that we have to consider algebraic curves of degree $O(D^2)$, where D is the largest degree that occurs. Compared to that effort, the lifting only requires a fraction of the total running time.

The outline of the chapter is as follows: We next present related work. In §5.1 we introduce the problem theoretically and derive some conditions that surfaces are required to fulfill and identify simple tasks. How to realize them with algebraic surfaces is explained in §5.2. Then, §5.3 discusses the generic part of the implementation — the framework. We also introduce the *SurfaceTraits_3* concept. We continue in §5.4 with the details on our models. Both rely on the same projection, but differ in the lifting and adjacency tasks. Details on the individual handling of a vertical line possibly contained in an algebraic surface is postponed to this part of the chapter. A set of possible algorithms utilizing the framework is surveyed in §5.5. We conclude with experimental results in §5.6 and a summary in §5.7 that also shows further directions.

Main parts of this chapter are based on results obtained in collaboration with Michael Kerber and Michael Sagraloff from the Max-Planck-Institut für Informatik, Saarbrücken, Germany. They previously appeared in [BS08] and [BKS08].

Related work Our strategy for algebraic surfaces in general follows elimination theory [BPR06] and main ideas of the powerful cylindrical algebraic decomposition (cad); see our introduction in §2.1.6 that presents the basic algorithm and also a series of improvements that reduce the number of considered polynomials. A collection of articles emblazing different aspects of cad is given in [CJ98]. Some ideas of our algorithm already appeared in those articles; for other problems, we propose novel alternatives. We discuss the similarities and differences with the appropriate references when we discuss the algorithm in detail.

Many algorithms in computational geometry can be expressed in terms of a cad-instance. A famous example is the Piano Mover's problem that is extensively discussed in [SSH87]. Unfortunately, many implementations, if any, avoid this technique. We believe for two reasons. The first is the quite high complexity of cad. The other is the algebraic focus, that usually requires good knowledge of the topic. Thus, with our framework we want to close the gap, between cad-techniques and implementations of algorithms in computational geometry. Our goal is to provide an easy-to-use framework, with full power on the analysis of surfaces, while always focusing towards applications in computational geometry. As we decouple combinatorics from predicates, it depends on the model used, whether the instantiated framework follows the exact computation paradigm [Yap04]. Note that most generic implementations of geometric algorithms show an undetermined behavior or fail to stop if instantiated with floating-point arithmetic. Thus, we strongly encourage to use the framework with models relying on exact number types and to apply consistent and certified filters for speed-ups. Our models do so.

If restricting to the three-dimensional case, we already mentioned earlier CGAL's `Nef_3` package that provides a robust and efficient implementation of three-dimensional Nef-polyhedra; see [HKM07] and [HK07b]. Its extension for quadrics is currently under development [HL08] relying on the parameterization of the the quadrics' intersections [DHPS07]. However, up to now, no complete implementation for arrangements of algebraic surfaces is available (even not for low degrees). [MTT05] presented a method to compute arrangements of quadrics using a space-sweep. An implementation is missing. For two quadrics,

a specialized projection approach is available as software [BHK+05]. In contrast to that work, the proposed framework can deal with more than two quadrics, allows more surfaces, and does not pose any generic position assumptions. Thus, it can be interpreted as a key step towards arrangements of surfaces.

Even if we restrict to one or two surfaces, our work constitutes an important step. Principally, there are two approaches for the topology computation of an algebraic surface: One considers *level-curves* of the surface for certain critical values and to connect the components of these levels in order to obtain a topological description of the surface; for example, Mourrain and Técourt [MT05] (see also [BCSM+]), Fortuna et al. [FGL04], [FGPT03] (for non-singular curves), and Alcázar et al. [ASS07] (with missing connection) follow this idea. The other approach relies on a projection of the critical points of the surface to the plane. The topology is then deduced by lifting the features induced by this projection. Note that our work falls into this category; see also Cheng et al. [CGL05] and the mentioned relations to cad.

It should be remarked that all algorithms that compute a surface's topology are similar, that is, they require to analyze curves and have to detect *critical points* of the surface. This typically involves resultant-calculus or Groebner bases. To simplify, most algorithms apply a linear (topology-preserving) shear; for example, [MT05], [FGL04], [FGPT03], and [CGL05] (for vertical lines). We abstain from this strategy, as we also want to preserve geometric properties of the input. In addition, it seems not easy to derive a back-shear algorithm, as it is established in the planar case; see [BKS08] and [EK08a].

Unfortunately, practical performances are not stated for any of these articles [MT05], [FGL04], [FGPT03], [ASS07], [CGL05], if they provide an implementation at all. Practical results are included only for special sub-classes, such as quadrics [BHK+05] and non-singular surfaces [PV07]. All other carry out symbolic computations, or abstain from reporting on implementations of certain substeps.

Recently, results on space curves that are defined by the intersection of two surfaces have been published [Kah08], [AS05], [GLMT05], and [DMR08]. The special case of tori that are intersected by natural quadrics has been analyzed by Reithmann [Rei08].

In contrast to all the previous work, our results profit from certified approximative methods that accelerate the algorithm significantly. We take this as the main reason of the overall good practical performance of our algorithm.

5.1. Problem

Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ be a set of surfaces, that is, two-dimensional manifolds in three-dimensional Euclidean space. We next introduce our objective formally, which allows to split the problem into a set of subtasks. For $p = (p_x, p_y) \in \mathbb{R}^2$, we denote with $\ell_p = \{(p_x, p_y, z) \in \mathbb{R}^3\} \subset \mathbb{R}^3$ the vertical line through p. We denote $V_i := \{p \in \mathbb{R}^2 \mid \ell_p \subset S_i\}$ the set of all points $p \in \mathbb{R}^2$ where S_i contains the vertical line ℓ_p. Let $\mathcal{V} = \bigcup_{1,\ldots,n} V_i$.

We tackle the following abstract problems, that is, we consider a surface as set of points.

Problem 5.1 (Intersections with vertical line). Given a set of surfaces $\mathcal{S}$, compute for an arbitrary point $p \in \mathbb{R}^2$ the ordered sequence of intersections of all $S_i \in \mathcal{S}$ with ℓ_p (or that $\ell_p \cap S_i = p \times \mathbb{R}$).

In order to encode the sequence of intersections of $S_i \in \mathcal{S}$, $i = 1, \ldots, n$, with ℓ_p we use an ordered sequence of subsets:

Definition 5.2 (z-pattern). We call the sequence $W_{p,\mathcal{S}} = w_{p,1}, \ldots, w_{p,k}$ of subsets of $\{1, \ldots, n\}$ a *z-pattern* with respect to p and $\mathcal{S}$. The pattern also comprises a subset $w_p^{\downarrow} := \{i \in \{1, \ldots, n\} \mid p \in V_i\}$. All subsets can be empty.

Intuitively, $W_{p,\mathcal{S}}$ describes how the surfaces behave along ℓ_p. Some of them are vertical at p, the remaining ones have finite intersections with ℓ_p. Each $w_{p,l}$ corresponds to a z-coordinate z_l where at least one such surface intersects ℓ_p, that is, $w_{p,l} := \{i \in \{1, \ldots, n\} \mid (p, z_l) \in S_i\}$.

Example 5.3. Consider $\mathcal{S} = \{S_1, S_2\}$ consisting of two unit spheres: S_1 centered at the origin and S_2 centered at $(0, 0, -2)$. That is, the south pole of S_1 intersects with the north pole of S_2. This is the only intersection of the spheres. Let $p_1 = (0, 0)$, $p_2 = (\frac{1}{2}, 0)$, $p_3 = (1, 0)$, and $p_4 = (2, 0)$.

Then, $\forall h = 1 \ldots 4$ we have $w_{p_h}^{\downarrow} = \emptyset$. The other sequences are: $W_{p_1,\mathcal{S}} = \{2\}, \{1, 2\}, \{1\}$. $W_{p_2,\mathcal{S}} = \{2\}, \{2\}, \{1\}, \{1\}$, $W_{p_3,\mathcal{S}} = \{2\}, \{1\}$, while $W_{p_4,\mathcal{S}}$ is an empty sequence.

If we fix p, Problem 5.1 can be split into two, the consecutive Problems 5.4 and 5.22.

Problem 5.4 (Compute z-pattern). Given a surface S_i and a point $p \in \mathbb{R}^2$ compute $W_{p,\{S_i\}}$.

We require the following condition.

Condition 5.5 (Finite number of vertical lines). For a given surface S_i it holds $|V_i|$ is finite. This implies that $\mathcal{V}$ also has finite size.

We introduce the following container.

Definition 5.6 (z-fiber). Let $S_i \in \mathcal{S}$, $p = (p_x, p_y) \in \mathbb{R}^2$. A finite subset $Z_{p,i} \subset \{z \in \mathbb{R} \mid (p_x, p_y, z) \in S_i\} \cup \{\pm\infty\}$ is called *z-fiber of S_i at p*. We *sort* its $m_{p,i} + 2$ elements in the following way:

$$-\infty = z_{p,i,-1} < z_{p,i,0} < \ldots < z_{p,i,m_{p,i}-1} < z_{p,i,m_{p,i}} = +\infty$$

Whereas the container is intended to encode the intersections of a surface S_i with ℓ_p for $p \notin V_i$, its purpose for $p \in V_i$ is to store interesting z-coordinates of S_i. Its actual content with respect to $p \in V_i$ is specified in Definition 5.10 and fixed by the Conditions 5.7 and 5.9 that define how surfaces are allowed to be connected.

We concentrate on the fact, that $m_{p,i}$ denotes its number of finite elements. In general, we cannot compute $Z_{p,i}$ for all $p \in \mathbb{R}^2$. Thus, we aim for a subdivision of the plane into finitely many (relatively) open and connected cells of dimension 0, 1, and 2 with the property that all points of a cell carry the same m-value. Such a finite subdivision can be represented as a planar arrangement; see §2.4. More detailed, we aim for surfaces to fulfill the following condition.

Condition 5.7 (Finite surface arrangement). Given a surface $S_i \in \mathcal{S}$. An arrangement $\mathcal{A}_{\{S_i\}}$ with the following properties exists:

- $\mathcal{A}_{\{S_i\}}$ consists of a *finite* number of cells and is induced by a *finite* number of continuous curves and a *finite* number of isolated points.
- $\mathcal{A}_{\{S_i\}}$ contains every point in V_i as vertex.
- Each cell Γ of $\mathcal{A}_{\{S_i\}}$ is invariant with respect to m, that is, $\forall p_1, p_2 \in \Gamma : m_{p_1,i} = m_{p_2,i} =: m_{\Gamma,i}$.

Such an arrangement is called *m_i-invariant*.

As a consequence, it suffices to only consider a sample point p_Γ of a cell Γ, if one is simply interested in $m_{p,i}$ for any point $p \in \Gamma$. This piece of information is valid for the whole cell. On the other hand, geometry is local to the point: In general, the entries of $Z_{p_1,i}$ differ from $Z_{p_2,i}$ if $p_1 \neq p_2$, even if $p_1, p_2 \in \Gamma$. Anyhow, we denote, for convenience, the z-fiber of Γ's sample point p_Γ with $Z_{\Gamma,i}$. It is possible to lift Γ:

Definition 5.8 (Lift). Let S_i be a surface, and Γ be an m_i-invariant set. For each $l = 0, \ldots, m_{\Gamma,i}$, the *$l$-th lift of S_i over Γ* is given by

$$\Gamma^{(l,i)} := \{(p_x, p_y, z_{p,i,l}) \in \Gamma \times \mathbb{R} \mid z_{p,i,l} \in Z_{p,i}\}$$

Lifts allow to decompose S_i into open cells, which requires Condition 5.9. Below, we introduction decompositions formally.

It is missing, how the entries of $Z_{\Gamma_1,i}$ and $Z_{\Gamma_2,i}$, for Γ_1, Γ_2 being cells of $\mathcal{A}_{\{S_i\}}$, are related, and, thus, encodes the adjacencies of lifts. Let us introduce a condition that helps to precisely define this relation.

Condition 5.9 (Continuation). Let $S_i \in \mathcal{S}$, $\mathcal{A}_{\{S_i\}}$ an m_i-invariant arrangement, and Γ_1, Γ_2 being two cells of it with $\dim(\Gamma_1) > 0$. Then, S_i is continuous in the following sense:

1. Let $p_t \in \Gamma_1$ be a sequence of points with a unique limit in Γ_2, that is, $\lim_{t\to\infty} p_t = p \in \Gamma_2$. Let $Z_{p_t,i} = \left\{z_{p_t,-1}, \ldots, z_{p_t,i,m_{\Gamma_1,i}}\right\}$. Then, for any $l \in \{0, \ldots, m_{\Gamma_1,i} - 1\}$ we have $\{\lim_{t\to\infty} z_{p_t,i,l} \mid p_t \in \Gamma_1 \text{ with } p_t \to p\} = [z_{p,i,v_l^-}, z_{p,i,v_l^+}] =: I_{p,i,l}$ with $z_{p,i,v_l^\pm} \in Z_{p,i}$ and $\lim_{t\to\infty} z_{p_t,i,l_1} \leq \lim_{t\to\infty} z_{p_t,i,l_2}$ for $-1 \leq l_1 < l_2 \leq m_{\Gamma_1,i}$.

2. For $p \notin V_i$ each interval $[z_{p,i,v_l^-}, z_{p,i,v_l^+}]$ consists of exactly one point, that is, $z_{p,i,v_l^-} = z_{p,i,v_l^+}$.

   Note that the number of intervals $[z_{p,i,v_l^-}, z_{p,i,v_l^+}]$ must be finite, as $m_{\Gamma_1,i}$ is finite.

This neighborhood-relation suffices to encode the connectivity of all lifted cells. We remark that from the above conditions it follows that each cell $\Gamma \in \mathcal{A}_{\{S_i\}} \setminus V_i$ is the projection of $m_{\Gamma,i}$ connected, disjoint cells of S_i respectively. Note the similar notation of delineation in §2.1.3. For $p \in V_i$ the intervals $[z_{p,i,v_l^-}, z_{p,i,v_l^+}]$ play an important role when decomposing ℓ_p in Definition 5.28.

Whereas the z-fiber at a point $p \notin V_i$ is determined precisely, its content at points $p \in V_i$ is only implicitly given by the chosen arrangement and Condition 5.9:

Definition 5.10 (Content of z-fiber). For a surface $S_i \in \mathcal{S}$ with m_i-invariant arrangement $\mathcal{A}_{\{S_i\}}$ the entries of the z-fiber $Z_{p,i}$ are defined as follows:

- $Z_{p,i} := \{z \in \mathbb{R} \mid (p_x, p_y, z) \in S_i\} \cup \{\pm\infty\}$ for $p \notin V_i$
- $Z_{p,i} := \{z \in \mathbb{R} \mid \exists \Gamma \in \mathcal{A}_{\{S_i\}}, l \in \{-1, \ldots, m_{\Gamma,i}\}$ such that z is an endpoint of some $I_{\Gamma,i,l}\}$ for $p \in V_i$

Problem 5.11 (Compute z-fiber). For given surface $S_i \in \mathcal{S}$ and given point $p \in \mathbb{R}^2$, compute $Z_{p,i}$, even if $p \in V_i$. For an illustration see Figure 5.1.

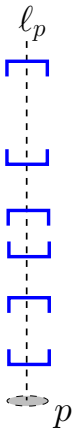


Figure 5.1. Computing the z-fiber for a surface at given point: We aim to represent finite entries as refineable intervals.

It remains to compute the connections between lifted cells which is encoded in terms of connections between lifted sample points.

Problem 5.12 (Adjacency). Given $\mathcal{A}_{\{S_i\}}$ for a surface $S_i \in \mathcal{S}$. Let Γ_1, Γ_2 denote incident cells of $\mathcal{A}_{\{S_i\}}$ and p_1, p_2 their respective planar sample points. Then, we are interested in how an entry of $Z_1 := Z_{\Gamma_1,i}$ is connected with the intervals defined by the entries of $Z_2 := Z_{\Gamma_2,i}$. We are asking for a list L of pairs $(a, b) \in A \times B$, with $A := \{-1, \dots, m_{\Gamma_1,i}\}$ and $B := \{-1, \dots, m_{\Gamma_2,i}\}$. We distinguish 5 cases for a fixed $a_0 \in \{0, \dots, m_{\Gamma_1,i} - 1\}$:

$\{b \mid (a_0, b) \in L\} = \emptyset$**:** Indicates, that there exists no continuous path on S_i whose closure connects (p_1, z_{p_1,i,a_0}) with some $(p_2, z_{p_2,i,b}), b \in B$.

$\{b \mid (a_0, b) \in L\} = \{b_0\} \wedge b_0 \notin \{-1, m_{\Gamma_2,i}\}$**:** The pair (a_0, b_0) then denotes the existence of a continuous path on S_i, lying over Γ_1, whose closure connects (p_1, z_{p_1,i,a_0}) with (p_2, z_{p_2,i,b_0}).

$\{b \mid (a_0, b) \in L\} = \{b_0\} \wedge b_0 = -1$ **(or** $b_0 = m_{\Gamma_2,i}$**):** The pair (a_0, b_0) denotes the existence of a continuous path, lying over Γ_1, whose closure connects (p_1, z_{p_1,i,a_0}) with the infinite "point" $(p_2, -\infty)$ (or $(p_2, +\infty)$), that is, S_i has a vertical asymptote with respect to z at p_2.

$|\{(a_0, b) \in L\}| = 2$**:** Let (a_0, b_0) and (a_0, b_1) be these pairs. They denote the existence of an infinite number of continuous paths on S_i, lying over Γ_1, such that exactly all points $(p_2, z), z \in [z_{p_2,i,b_0}, z_{p_2,i,b_1}]$ are connected with (p_1, z_{p_1,i,a_0}) by considering the closure of a path. In case that $b_0 = -1$ or $b_1 = m_{\Gamma_2,i}$, the interval is meant to be open at that end.

For an illustration we refer to Figure 5.2. The case distinction is analogue for fixed $b_0 \in \{0, \dots, m_{\Gamma_2,i} - 1\}$. Note that we only compute adjacencies between zero-, one-, and tow-dimensional cells. The adjacencies to three-dimensional open cells are given implicitly by them and the projection technique.

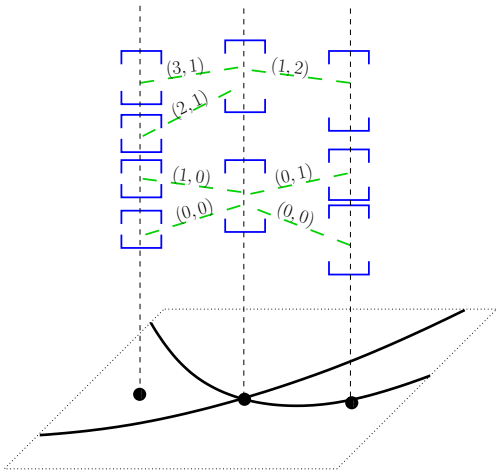


Figure 5.2. Compute adjacency relation of incident z-fibers

We next turn to consider more than one surface and we already state the first condition that characterizes the surfaces' intersections. In what follows, let always $S_i, S_j \in \mathcal{S}$ with $i \neq j$.

Condition 5.13 (One-dimensional intersection). $\dim(S_i \cap S_j) \leq 1$.

Similar to the single-surface case, we introduce an abstract container:

Definition 5.14 (Multi-surface z-fiber). Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ be a set of surfaces and $p = (p_x, p_y) \in \mathbb{R}^2$. A finite subset $\mathcal{Z}_{p,\mathcal{S}} \subset \{z \in \mathbb{R} \mid \exists i \in \{1, \ldots, n\} : (p_x, p_y, z) \in S_i\} \cup \{\pm\infty\}$ is called *multi-surface z-fiber of $\mathcal{S}$ at p*. We sort the entries of $\mathcal{Z}_{p,\mathcal{S}}$:

$$-\infty = z_{p,\mathcal{S},-1} < z_{p,\mathcal{S},0} < \ldots < z_{p,\mathcal{S},m_{\mathcal{S}}-1} < z_{p,\mathcal{S},m_{\mathcal{S}}} = +\infty$$

Its purpose is to store the intersections of $\mathcal{S}$ with ℓ_p if $p \notin \mathcal{V}$. In case that $p \in \mathcal{V}$, we want to store interesting z-coordinates that decompose ℓ_p into a finite number of open intervals. The value $m_{p,\mathcal{S}}$ denotes the number of finite entries of a multi-surface z-fiber. In Definition 5.26 we also introduce multi-surface lifts, which pose a central tool for our intended cell decomposition. But before, we remark that such a fiber can be related to single-surface z-fibers, in particular for two given surfaces:

Definition 5.15 ($m_{p,i,j}$). Let $S_i, S_j \in \mathcal{S}, i \neq j$ and let $p \in \mathbb{R}^2 \setminus (V_i \cup V_j)$. Then $m_{p,i,j} := |\{z \in \mathcal{Z}_{p,\mathcal{S}} \mid z \in Z_{p,i} \wedge z \in Z_{p,j}\}|$. A connected set of points Γ is called *$m_{i,j}$-invariant* if $m_{p_1,i,j} = m_{p_2,i,j}$ for $p_1, p_2 \in \Gamma$ with $p_1 \neq p_2$. We define $m_{\Gamma,i,j} := m_{p,i,j}$ for some $p \in \Gamma$.

Again, we cannot compute multi-surface z-fibers for an infinite number of points. This fact founds another condition on two surfaces (and thus on any number of surfaces). Similar to the single-surface case, we want to group points into sets:

Condition 5.16 (Finite two-surface arrangement). Given surfaces $S_i, S_j \in \mathcal{S}, i \neq j$. An arrangement $\mathcal{A}_{\{S_i,S_j\}}$ exists, with:

- $\mathcal{A}_{\{S_i,S_j\}}$ consists of a *finite* number of cells and is induced by a *finite* number of continuous curves and a *finite* number of isolated points.
- $\mathcal{A}_{\{S_i,S_j\}}$ contains every point of $V_i \cup V_j$ as a vertex.
- For each cell Γ of $\mathcal{A}_{\{S_i,S_j\}} \setminus V_i \cup V_j$, the following equations hold: $\forall p_1, p_2 \in \Gamma : m_{p_1,i} = m_{p_2,i}, m_{p_1,j} = m_{p_2,j}, m_{p_1,i,j} = m_{p_2,i,j}$.

Such an arrangement is called $m_{i,j}$*-invariant*.

Remark. Observe that $\mathcal{A}_{\{S_i,S_j\}}$ is an m_i- and m_j-invariant arrangement.

Again, it suffices to choose a sample point p_Γ, in case, one is only interested in the cell-related information $m_{\Gamma,i,j}$. In addition, it holds that $\forall p_1, p_2 \in \Gamma : W_{\{S_i,S_j\},p_1} = W_{\{S_i,S_j\},p_2}$. Note that geometric information can be deduced from the individual fibers $Z_{p,i}$ and $Z_{p,j}$, for any $p \in \Gamma$, but typically we use $p = p_\Gamma$.

Condition 5.17 (Continuation for two surfaces). Let $\mathcal{S} = \{S_i, S_j\}$, $\mathcal{A}_{\{S_i,S_j\}}$ an $m_{i,j}$-invariant arrangement and Γ_1, Γ_2 being two cells of it with $\dim(\Gamma_1) > 0$.

1. Let $p_t \in \Gamma_1$ be a sequence of points with $\lim_{t\to\infty} p_t = p \in \Gamma_2$, and $\mathcal{Z}_{p_t,\mathcal{S}} = \left\{z_{p_t,-1}, \ldots, z_{p_t,\mathcal{S},m_{\Gamma_1,\mathcal{S}}}\right\}$. For any $l \in \{0, \ldots, m_{\Gamma_1,\mathcal{S}} - 1\}$ we have $\{\lim_{t\to\infty} z_{p_t,\mathcal{S},l} \mid p_t \in \Gamma_1 \text{ with } p_t \to p\} = [z_{p,\mathcal{S},v_l^-}, z_{p,\mathcal{S},v_l^+}] =: I_{p,\mathcal{S},l}$ with $z_{p,\mathcal{S},v_l^\pm} \in \mathcal{Z}_{p,\mathcal{S}}$. In addition, $\lim_{t\to\infty} z_{p_t,\mathcal{S},l_1} \leq \lim_{t\to\infty} z_{p_t,\mathcal{S},l_2}$ for $-1 \leq l_1 < l_2 \leq m_{\Gamma_1,\mathcal{S}}$.

2. For $p \notin V_i \cup V_j$ each interval $[z_{p,\mathcal{S},v_l^-}, z_{p,\mathcal{S},v_l^+}]$ consists of exactly one point, that is, $z_{p,\mathcal{S},v_l^-} = z_{p,\mathcal{S},v_l^+}$.

Note again that the number of intervals $[z_{p,\mathcal{S},v_l^-}, z_{p,\mathcal{S},v_l^+}]$ must be finite, as $m_{\Gamma_1,\mathcal{S}}$ is finite.

It is no surprise that we next want to define the actual content of a multi-surface z-fiber, followed by some remarks on the adjacency computation. We can assume that $\mathcal{S}$ consists of two surfaces S_i and S_j. The extension to any number is straightforward.

An implication of Condition 5.13 and Condition 5.17 is that each non-vertical part $S_i \cap S_j$ is expected to have a unique end-point (p, p_z), even if $p \in V_i \cup V_j$. In addition, we have: If $p \notin V_i$ then $p_z = z_{p,i,l_i'}$ for some l_i' and if $p \notin V_j$ then $p_z = z_{p,j,l_j'}$ for some l_j'. Following, $(Z_{p,i} \cup Z_{p,j}) \setminus \{\pm\infty\}$ comprises all z-coordinates of $S_i \cap S_j \cap \ell_p$ for $p \notin V_i \cup V_j$. Thus, for such p we can define $\mathcal{Z}_{p,\{S_i,S_j\}} := Z_{p,i} \cup Z_{p,j}$, which constitutes the easy case of Definition 5.20. In contrast, if $p \in V_i \cup V_j$, there is no such direct solution. We define another set, which also implies a problem to solve:

Definition 5.18 ($Z^{|}_{p,i,j}$). Consider the setting as in Condition 5.17, that is, $\mathcal{S} = \{S_i, S_j\}$. Let $p \in V_i \cup V_j$. Then $Z^{|}_{p,i,j} := \{z \in \mathbb{R} \mid \exists \Gamma \in \mathcal{A}_\mathcal{S}, l \in \{-1, \ldots, m_{\Gamma,\mathcal{S}}\}$ such that z is an endpoint of some $I_{\Gamma,\mathcal{S},l}\}$.

Remark. Actually, it is valid to also define $Z^{|}_{p,i,j}$ for any $p \in \mathbb{R}^2$. But nothing is won by doing so, as, by Condition 5.9, we have $Z^{|}_{p,i,j} \subset Z_{p,i} \cup Z_{p,j}$.

Problem 5.19 (Compute $Z^{|}_{p,i,j}$). For given S_i, S_j with $i \neq j$, and $p \in V_i \cup V_j$, compute $Z^{|}_{p,i,j}$.

Definition 5.20 (Content of multi-surface z-fiber). Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ be a set of surfaces and $p \in \mathbb{R}^2$. Then

$$\mathcal{Z}_{p,\mathcal{S}} = \left(\bigcup_{i=\{1,\ldots,n\}} Z_{p,i} \right) \cup \left(\bigcup_{p \in V_i \cup V_j, i \neq j} Z^{|}_{p,i,j} \right)$$

is called the *multi-surface z-fiber of $\mathcal{S}$ at p*.

Problem 5.21 (Compute $\mathcal{Z}_{p,\mathcal{S}}$). For given $\mathcal{S}$ and $p \in \mathbb{R}^2$, compute $\mathcal{Z}_{p,\mathcal{S}}$.

Observe that the z-pattern for $p \in \mathbb{R}^2 \setminus \mathcal{V}$ can be defined in terms of $\mathcal{Z}_{p,\mathcal{S}}$. We have $\forall 0 \leq l \leq m_{\mathcal{S}} - 1$:

$$w_{p,l} \quad := \quad \{i \in \{1, \ldots, n\} \mid z_{p,\mathcal{S},l} \in \mathcal{Z}_{p,\mathcal{S}} : z_{p,\mathcal{S},l} \in Z_{p,i}\} \tag{5.1}$$

The z-pattern for $p \in \mathcal{V}$ requires to also consider $Z^{|}_{p,i,j}$. To actually compute $\mathcal{Z}_{p,\mathcal{S}}$ in both cases, for example, with a multi-way merge algorithm, we must be able to decide the following problem:

Problem 5.22 (Compare entries of z-fibers). Let $S_i, S_j \in \mathcal{S}, i \neq j$. Given a point p and $z_{p,i,l_i} \in Z_{p,i}$ and $z_{p,j,l_j} \in Z_{p,j}$ decide whether $z_{p,i,l_i} < z_{p,j,l_j}$, $z_{p,i,l_i} = z_{p,j,l_j}$, or $z_{p,i,l_i} > z_{p,j,l_j}$. A similar comparison is required for $z_{p,i,l_i} \in Z_{p,i}$ and $z^{|}_{p,i',j',l_{i',j'}} \in Z^{|}_{p,i',j'}$, for $i' \neq j'$. This problem is illustrated in Figure 5.3.

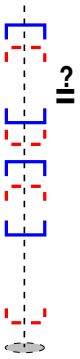


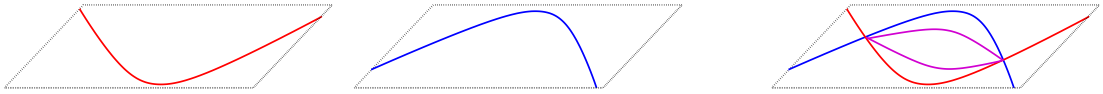
Figure 5.3. Check whether two z-fiber entries have equal z-coordinate

Note that this constitutes a solution to Problem 5.1. Actually, we learn in §5.3.1 that the equality decision is sufficient to compute the multi-surface z-fiber and the corresponding z-pattern. The reason is that we represent each $\pm\infty \neq z_{p,i,l} \in Z_{p,i}$ (and so the entries of $Z^{|}_{p,i,j}$) with a refineable interval approximation. If sufficiently refined, it is easy to decide $<$ and $>$, while for $=$ the refinement would never stop. Thus, we need to decide it externally.

Let us finally collect the missing tasks. Note that Problem 5.24 constitutes the central objective of our work.

Problem 5.23 (Compute planar arrangements). For given surfaces $S_i, S_j \in \mathcal{S}, i \neq j$, compute $\mathcal{A}_{\{S_i\}}$, $\mathcal{A}_{\{S_j\}}$, and $\mathcal{A}_{\{S_i,S_j\}}$. Figure 5.4 shows the different cases.

Figure 5.4. Compute planar arrangements



(a) Compute $\mathcal{A}_{\{S_i\}}$ and $\mathcal{A}_{\{S_j\}}$ **(b)** Compute $\mathcal{A}_{\{S_i,S_j\}}$

Problem 5.24 (Compute $\mathcal{A}_{\mathcal{S}}$, compute z-fibers and their adjacency relation). Given a set of surfaces $\mathcal{S}$ fulfilling the listed conditions. Compute a finite planar arrangement $\mathcal{A}_{\mathcal{S}}$ with the property that for each of its cells Γ it holds: $\forall p_1, p_2 \in \Gamma : W_{\mathcal{S},p_1} = W_{\mathcal{S},p_2}$. In addition, we want to solve Problem 5.25, that is, to compute the adjacency relation of entries of multi-surface z-fibers; see also Figure 5.5.

Problem 5.25 (Multi-surface adjacency). Given $\mathcal{A}_{\mathcal{S}}$ for a set of surface $\mathcal{S}$. Let Γ_1, Γ_2 denote incident cells of $\mathcal{A}_{\mathcal{S}}$ and p_1, p_2 their respective planar sample points. Then, we are interested in how an entry of $\mathcal{Z}_{\Gamma_1,\mathcal{S}}$ is connected with the intervals defined by the entries of $Z_{\Gamma_2,\mathcal{S}}$. The output is identical to Problem 5.12.

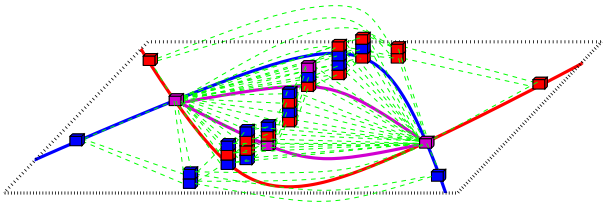


Figure 5.5. Decompose $\mathcal{S}$ into a finite number of lifted cells and compute their adjacency relation

It is clear that lifting the *overlay* of all arrangements $\mathcal{A}_{\{S_i\}}$ and $\mathcal{A}_{\{S_i,S_j\}}$ and computing the adjacency relation of their lifts constitutes a solution to Problem 5.24. We claim that the framework that we present in §5.3 implements this solution using subalgorithms for Problems 5.23 (planar arrangements), 5.11 and 5.19 (z-fibers), 5.22 (compare entries of z-fibers), and 5.12 (adjacency). The last is used to derive the desired connectivity of entries of z-patters from the connectivities of z-fibers of single surfaces. Only in case that some cell contains $p \in \mathcal{V}$ we have to solve a special subcase of Problem 5.25 as well.

Cell decompositions

We next introduce decompositions of surfaces into connected cells of dimension 0, 1, and 2 using planar arrangements and z-fibers. However, one definition is missing for this purpose:

Definition 5.26 (Multi-surface lift). Let $\mathcal{S}$ be a set of surfaces, and Γ be a set with constant z-pattern $W_{\Gamma,\mathcal{S}}$ for all points $p \in \Gamma$. For $l = 0, \ldots, m_{\Gamma,\mathcal{S}} - 1$, the *$l$-th multi-surface lift of $\mathcal{S}$ over Γ* is given by

$$\Gamma^{(l,\mathcal{S})} := \{(p_x, p_y, z_{p,\mathcal{S},l}) \in \Gamma \times \mathbb{R} \mid z_{p,\mathcal{S},l} \in \mathcal{Z}_{p,\mathcal{S}}\}$$

Note that for fixed l_0 it holds that $\Gamma^{(l_0,\mathcal{S})} = \Gamma^{(l_i,i)}$ with $l_i \in w_{\Gamma,l_0}$ and that in case of an intersection we have $|w_{\Gamma,l_0}| > 1$. Multi-surface lifts are essential for our decomposition. In addition, special diligence is required for vertical lines contained in a surface. But, we start with a simple case:

Definition 5.27 (Cell decomposition of S_i without vertical line). Let S_i be a surface, with $V_i = \emptyset$ and $\mathcal{A}_{\{S_i\}}$ fulfilling Condition 5.7. Let $\Gamma \in \mathcal{A}_{\{S_i\}}$, $Z_{\Gamma,i}$ its z-fiber, and $m_{\Gamma,i}$ the number of finite elements in $Z_{\Gamma,i}$. The *cell decomposition* $\Omega_{\{S_i\}}$ is defined as

$$\Omega_{\{S_i\}} := \bigcup_{\Gamma \in \mathcal{A}_{\{S_i\}}} \left(\bigcup_{l=0,\ldots,m_\Gamma-1} \{\Gamma^{(l,i)}\} \right)$$

For given S_i and S_j, $j \neq i$, we can also use $\mathcal{A}_{\{S_i,S_j\}}$ (instead of $\mathcal{A}_{\{S_i\}}$) to support $\Omega_{\{S_i\}}$, as $\mathcal{A}_{\{S_i,S_j\}}$ also fulfills Condition 5.7. However, this typically results in a larger number of cells.

We next extend Definition 5.27 to give a cell decomposition for a surface S_i that also comprises vertical lines. Remember that the set of vertical lines is finite. The idea is to also decompose each ℓ_p with $p \in V_i$ into segments and rays respecting the intervals boundaries arising from Condition 5.9.

Definition 5.28 (Cell decomposition of S_i with vertical line(s)). Let S_i be a surface with an arrangement $\mathcal{A}_{\{S_i\}}$ fulfilling Condition 5.7. For $p \in V_i$, let ω_p denote the partition of ℓ_p into elements of $Z_{p,i}$ and their induced intervals of $\mathbb{R}$. We define

$$\Omega_{\{S_i\}} := \bigcup_{\Gamma \in \mathcal{A}_{\{S_i\}} \setminus V_i} \left(\bigcup_{l=0,\ldots,m_\Gamma-1} \{\Gamma^{(l,i)}\} \right) \cup \bigcup_{p \in V_i} \omega_p$$

We turn to the case of multiple surfaces contained in a set $\mathcal{S}$. For this objective, we base the definition of $\Omega_{\mathcal{S}}$ on the planar arrangement $\mathcal{A}_{\mathcal{S}}$.

Definition 5.29 (Cell decomposition of $\mathcal{S}$). Let $\mathcal{S}$ be a set of n surfaces and $\mathcal{A}_{\mathcal{S}}$ as computed by Problem 5.24. For $p \in \mathcal{V}$, let ω_p denote the partition of ℓ_p into elements of $\mathcal{Z}_{p,\mathcal{S}}$ and their induced intervals of $\mathbb{R}$.

$$\Omega_{\mathcal{S}} := \bigcup_{\Gamma \in \mathcal{A}_{\mathcal{S}} \setminus \mathcal{V}} \left(\bigcup_{l=0,\ldots,m_{\Gamma,\mathcal{S}}-1} \{\Gamma^{(l,\mathcal{S})}\} \right) \cup \bigcup_{p \in \mathcal{V}} \omega_p$$

In §5.2.4 we show that these decompositions constitutes stratifications of algebraic surfaces and also state bounds on the stratifications' complexities.

The cells of a decomposition Ω with respect to Definition 5.27, 5.28, and 5.29 are, by construction, connected. However, sometimes it might be advantageous to achieve simply connected cells. Remember that a cell is simply connected if each cycle in a cell is contractible to a point. Thus, we show how Ω can be transformed into a decomposition Ω' consisting of simply connected cells only. Remember that in order to obtain Ω we homeomorphically lift a planar arrangement $\mathcal{A}$. Thus, the main idea is Algorithm 5.1 that refines some arrangement $\mathcal{A}$ into an arrangement $\mathcal{A}'$ of simply connected cells. We show in Proposition 5.31 that $\mathcal{A}'$ and has the same complexity as $\mathcal{A}$. Notice that only cells of dimension 1 and 2 of an arrangement $\mathcal{A}$ can be non-simply connected.

Algorithm 5.1. Refine $\mathcal{A}$ into simply connected cells

INPUT: Planar arrangement $\mathcal{A}$

OUTPUT: $\mathcal{A}'$ consisting of simply connected cells only

- Transform $\mathcal{A}$ into a planar graph $\mathcal{G}$ by mapping its zero-dimensional cells to nodes, and its one-dimensional connected cells to edges.
- A one-dimensional circular edge is made simply connected by adding a new vertex; see the squared vertices in the Figure 5.6.
- We are left with non-simply connected faces. While $\mathcal{G}$ contains a bounded connected component:
  - Choose such a component and connect its y-minimal point downwards using a vertical arc until it reaches another component of $\mathcal{G}$ (or if this does not happen, the arc goes to $-\infty$); see dashed lines in Figure 5.6.

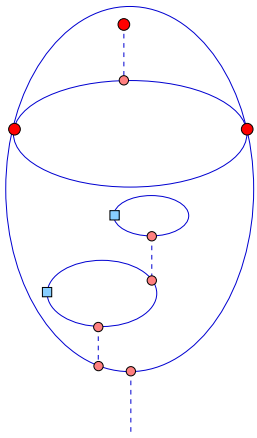


Figure 5.6. How to make cells of an arrangement $\mathcal{A}$ simply connected? Break one-dimensional circles (squares) and add vertical arcs (dashed). Each resulting face is simply connected.

Observe that each such arc either merges two connected components, or turns one of them unbounded. Thus, it is clear that the algorithm terminates, and produces a graph without bounded connected components. Some properties and results of the algorithm:

Proposition 5.30. *Each cell of $\mathcal{A}'$ is simply connected.*

Proof. Assume for a contradiction that there is a cell Γ of $\mathcal{A}'$ which is not simply connected. Clearly, Γ cannot be one-dimensional as we split all cycles. So assume that Γ is a face. Since it is not simply connected, there is a cycle C that is not contractible. Hence, its interior contains a connected component, which must be bounded. That contradicts the fact that there is no bounded connected component. □

Proposition 5.31. *The complexity of $\mathcal{A}'$ is the same as $\mathcal{A}$.*

Proof. Notice that for each connected component of $\mathcal{G}$, we introduce at most one edge and two vertices and split at most one face. The number of connected components is upper bounded by the number of faces of $\mathcal{A}$. We add at most 4 cells for each face of $\mathcal{A}$. This proves that we do not increase the complexity. □

Remarks.

- In the terminology of CGAL's `Arrangement_2` (see §2.4.3), we introduce a finite number of new (vertical) edges that such each inner CCB gets connected to the outer CCB of the face it belongs to. This means, that no face has an inner CCB and only one outer CCB. In addition, each isolated vertex is also connected to the outer CCB of the face that contains the point by a new (vertical) edge. An unbounded face is connected with the implicit fictitious outer CCB (see Figure 4.9 (a)). As a result, each (non-fictitious) face has neither an inner CCB nor an isolated vertex. By definition, such a face in a planar arrangement is simply connected.
- The computed graph induces a refined arrangement $\mathcal{A}'$ of $\mathcal{A}$. If the cells of $\mathcal{A}$ comprise data, the newly added cells obviously inherit the attached data of the cell they are included.

In what follows we only consider the single-surface case, as the multi-surface case is its natural extension and the corresponding adaptions for multi-surface cell-decompositions Ω are straightforward.

The arrangement $\mathcal{A}'_{\{S\}}$ implies a cell decomposition $\Omega'_{\{S\}}$ by lifting the cells of $\mathcal{A}'_{\{S\}}$; similar to Definition 5.27.

Proposition 5.32. *Each cell of $\Omega'_{\{S\}}$ is simply connected.*

Proof. Each cell ω' of $\Omega'_{\{S\}}$ is the homeomorphic image of a (simply connected) cell Γ' of $\mathcal{A}'_{\{S\}}$. It follows that ω' is simply connected as well. □

We mention that this refinement into simply connected cells has not yet been integrated into our implementation that we present in §5.3.

5.2. Operating algebraic surfaces

We next concentrate on algebraic surfaces. Such a surface S_i is defined by a trivariate polynomial $f_i \in \mathbb{Q}[x, z, y]$ of total degree D_i. We refer to $\deg_z(f_i)$ as $D_{z,i}$. We can assume that f_i is square-free and primitive, that is, S_i contains no irreducible component twice, and has no two-dimensional vertical component. In addition, each pair S_i, S_j, with $i \neq j$ is defined

by coprime polynomials. If the input does not fulfill these conditions, we can decompose pairs of non-coprime surfaces into (up to three) coprime ones and apply a square-free factorization as in §2.1.1. In other words: We treat vertical and multiple parts of each input surface separately. The intersection of two surfaces is at most one-dimensional. Note that for a fixed i we sometime use $S_i = V_{\mathbb{R}}(f_i) = V_{\mathbb{R}}(a_{D_{z,i}} z^{D_{z,i}} + , \ldots + , a_0 z^0)$.

We first, as in §5.2.1, consider z-fibers of a single surface and remember algebraic entities derived related to z-fibers that help to construct the desired planar arrangements $\mathcal{A}_{\{S_i\}}$ and $\mathcal{A}_{\{S_i,S_j\}}$ in §5.2.2 without the need to actually compute the fibers. But we require them for the actual lifting. §5.2.3 shows that the continuation conditions are fulfilled, while §5.2.4 revives the cell decompositions from §5.1 for algebraic surfaces. We also show that algebraic surfaces completely fulfill the conditions raised in §5.1. At the end of this section we give a short link to semi-algebraic surfaces.

5.2.1. z-fibers

Definition 5.33. Let $S_i \in \mathcal{S}$ be an algebraic surface defined by the vanishing set of f_i. The z-fiber of a point $p := (p_x, p_y) \in \mathbb{R}^2 \setminus V_i$ is

$$Z_{p,i} := \{z \in \mathbb{R} \mid f_i(p_x, p_y, z) = 0\}$$

Note that this definition omits to define z-fibers for $p \in V_i$, as for such points $\{z \in \mathbb{R} \mid f_i(p_x, p_y, z) = 0\} = \mathbb{R}$. This contrasts Definition 5.6 that expects $|Z_{p,i}|$ to be finite. To tackle this task, we below introduce three polynomials whose roots define the desired entries. Thus, the formal specification of such a z-fiber is postponed to Definition 5.61 on page 232 of §5.4.2. For now, we only rely on the fact that $Z_{p,i}$ with $p \in V_i$ decomposes ℓ_p into a finite number of pieces.

To compute $Z_{p,i}$ we require a method that is able to isolate the real roots of the polynomial $f_i(p) := f_i(p_x, p_y, z) \in \mathbb{R}[z]$, where p's coordinates are algebraically defined, which constitutes the first problem: $f_i(p) \in \mathbb{R}[z]$ has algebraic coefficients for many z-fibers computed by our method. A second problem is that $f_i(p)$ might have multiple roots.

Theorem 5.34 (Complexity of z-fiber for $p \notin V_i$). *Let S be an algebraic surface of degree D and $p \notin V_i$. Then, $|Z_{p,i}| \leq D$.*

Proof. $f_i(p) := f_i(p_x, p_y, z) \in \mathbb{R}[z]$ defines $Z_{p,i}$ and $\deg(f_i(p)) \leq D_{z,i} \leq D$. □

Let us derive additional exact values on $f_i(p)$ in order to simplify the desired computation.

Definition 5.35 (Local degrees). Let p be as above. The *local degree* $d_{p,i}$ is the degree of $f_i(p)$ in z. We also say that p is $d_{p,i}$*-regular*. In case that $f_i(p) \equiv 0$, $d_{p,i} = -\infty$. The *local gcd degree* $k_{p,i}$ is the degree of $\gcd(f_i(p), \frac{\partial}{\partial z} f_i(p))$. We also say that p has *degradation* $k_{p,i}$. The *local real degree* $m_{p,i}$ is the number of distinct real roots of $f_i(p)$.

How to compute these values? We start with the local degree $d_{p,i}$. Remember that $f_i = \sum_{d=0}^{D_{z,i}} a_d(x,y) z^d$. If f_i is z-regular, we are done. This can be checked by determining whether $\deg_z(f_i) = D$, that is, whether $\forall x, y \in \mathbb{R}$ we have $a_{D_{z,i}}(x,y) = c \neq 0$. Otherwise, we compute $d_{p,i} := \max\{d \mid a_d(p_x, p_y) \neq 0\}$ by starting with $d = D_{z,i}$ and stopping as soon as $a_d(p_x, p_y) \neq 0$. If even $a_0(p_x, p_y) = 0$, then $p \in V_i$ (remember that we excluded that S is completely vertical). In this case $d_{p,i} := -\infty$.

For computing $m_{p,i}$ and $k_{p,i}$ we refer the reader to Algorithm 2.10 on page 36 that is a specialized form of Algorithm 2.3. The algorithm relies on Sturm-Habicht sequences (see Definition 2.11) to obtain the number of real roots of $f_i(p)$. It is important to use a proper reductum of f_i if $d_{p,i} \neq D_{z,i}$; see also Lemma 2.19. Sturm-Habicht sequences are similar to signed subresultants. Thus, we encourage to follow the remark and invitation on page 28 to benefit from computed sequences when aiming for $k_{p,i}$; see also Lemma 2.13.

It must be said, that the specialization property (see Theorem 2.10) is central to this computation, as, in particular, we only know restricted information on p: We will know refineable interval approximations p_x and p_y and we will be able to check whether p lies on some planar curve. This is a perfect setting for the bitstream Descartes method.

We describe in detail how to use this method in combination with the computed values in §5.2.1. That part also discusses the missing case of how to compute the entries of $Z_{p,i}$ (and $\mathcal{Z}_{p,\mathcal{S}}$) for $p \in V_i$.

5.2.2. Planar arrangements

We next present (constructive) definitions for the desired arrangements $\mathcal{A}_{\{S_i\}}$ and $\mathcal{A}_{\{S_i,S_j\}}$ for algebraic surfaces $S_i, S_j \in \mathcal{S}, S_i \neq S_j$. We do not only prove that such arrangements exist, but also try to keep their sizes almost minimal with respect to the number of faces, edges, and vertices.

Constructing $\mathcal{A}_{\{S_i\}}$

Remember the local degrees from Definition 5.35 that give additional information on z-fibers of a single surface S_i. In this part we construct an arrangement $\mathcal{A}_{\{S_i\}}$, whose cells have invariant d and k. Following §5.2.1 the points of a cell also share the same m. As an arrangement consists of finitely many cells this construction shows that Condition 5.7 is fulfilled for an algebraic surface S_i. Implicitly, we also show Condition 5.5. Consequently, we are able to construct z-fibers over any point of the plane, since all algebraic information (local degrees) can be stored along each cell of $\mathcal{A}_{\{S_i\}}$ and is valid for each of the cell's points.

Definition 5.36 ((d,k)-invariance). A connected set $\Gamma \subset \mathbb{R}^2$ is called *(d,k)-invariant with respect to a surface* $S_i = V(f_i)$ if the local degree $d_{\Gamma,i} := d_{p,i}$ and the local gcd degree $k_{\Gamma,i} := k_{p,i}$ of f_i are invariant for all $p \in \Gamma$. A (d,k)-arrangement for S_i is a planar arrangement whose vertices, edges, and faces are (d,k)-invariant with respect to S_i.

The delineability (see Definition 2.21) of f_i on any (d,k)-invariant set has also been shown by Collins in his seminal work on cylindrical algebraic decomposition [Col75]. Remember the implication: The (real) lift over the set is the union of $m_{p,i}$ disjoint function graphs (also known as sheets; see §2.1.5). A slightly weaker version is:

Theorem 5.37. *Let Γ be a (d,k)-invariant set for $V(f_i)$. Then, each $p \in \Gamma$ has the same local real degree $m_{\Gamma,i}$. Even more: For each $l = 0, \ldots, m_{\Gamma,i} - 1$, the l-th lift $\Gamma^{(l,i)}$ over Γ is connected.*

Proof. The number of distinct complex roots over a (d,k)-invariant set is constantly $d - k$. The roots of $f_i(p)$ continuously depend on p, thus, in an open neighborhood of any point on Γ the imaginary roots stay imaginary. As the total number of roots is preserved and

imaginary roots only appear together with its complex conjugate, the real roots also remain real; see [Col75, Theorem 1] for more details. □

The next construction also appears in Collins' work [Col75, Theorem 4]:

Theorem 5.38 (Existence of (d,k)-invariant). *For each algebraic surface S_i, there exists a (d,k)-arrangement.*

Proof. Our proof is constructive. Let p be an arbitrary point in the plane, and $f_i = \sum_{d=0}^{D_{z,i}} a_d(x,y)z^d$. The local degree of f_i at p simply depends on the coefficients a_d. Remember from above:

$$d_{p,i} = \deg_z(f_i(p)) \,[= \max \mathcal{D}]$$

with $\mathcal{D} := \{d = 0, \ldots, D_{z,i} \mid a_d(p) \neq 0\}$. Note that in case $d_{p,i} = -\infty$, it holds $\mathcal{D} = \emptyset$.

The same way, the local gcd degree depends on the principal Sturm-Habicht coefficients $\mathrm{stha}_k((f_i)_{(d_p)})$ by

$$k_{p,i} = \deg_z(\gcd(f_i(p), \tfrac{\partial}{\partial z} f_i(p))) \,[= \min \mathcal{K}]$$

with $\mathcal{K} := \{k = 0, \ldots, d_{p,i} - 1 \mid \mathrm{stha}_k((f_i)_{(d_p)})(p) \neq 0\}$. Note that in case $k_{p,i} = -\infty$ it holds $\mathcal{K} = \emptyset$.

The coefficients a_d and $\mathrm{stha}_k((f_i)_{(d)}$ define algebraic plane curves $\alpha_d = V(a_d)$ and $\sigma_{d,k} = V(\mathrm{stha}_k((f_i)_{(d)}))$, respectively, of degree at most $D(D-1)$. Then, $d_{p,i}$ and $k_{p,i}$ are determined by the curves p is part of. Thus, the arrangement induced by $\alpha_{D_{z,i}}, \ldots, \alpha_0$ and, for all $d = 1, \ldots, D_{z,i}$, $\sigma_{d,0}, \ldots, \sigma_{d,d}$ has only (d,k)-invariant cells. □

Note that the number of curves is finite, and as each curve induces a finite arrangement, also the overlay consisting of all curves induces a finite arrangement. In addition, $\mathcal{A}_{\{S_i\}}$ subdivides $\mathbb{R}^2$ into cells of points that have an invariant pattern of (multiple) roots of $f_i(p)(z)$ for all p in a cell. This implies, that the z-pattern $W_{\{S\},p}$ can only change upon switching to another cell. This shows Condition 5.7. In addition: As S_i does not contain two-dimensional vertical components, all α_i intersect in finitely many points, which results in an alternative definition of $V_i := \{p \in \mathbb{R}^2 \mid \forall d = 0, \ldots, D_{z,i} : \; a_d(p_x, p_y) = 0\}$. This shows Condition 5.5.

The proof of Theorem 5.38 gives a way to construct some (d,k)-arrangement for a surface. However, its number of cells might be larger than needed. We aim for a clustering into few (d,k)-cells.

Definition 5.39 (Projected silhouette). The *projected silhouette* τ_{S_i} of S_i is defined by $\mathrm{stha}_0(f_i) = \mathrm{Res}_z(f_i, \frac{\partial f_i}{\partial z})$

Lemma 5.40. *For any point, $(d_{p,i}, k_{p,i}) = (D_{z,i}, 0)$ if and only if p is not on τ_{S_i}. Consequently, all edges and vertices of a (d,k)-arrangement not belonging to τ_{S_i} can be removed and their incident cells can be merged to a larger (d,k)-invariant cell.*

Proof. Following [BPR06, Proposition 4.27], we have $\mathrm{Res}_z(f_i, \frac{\partial f_i}{\partial z}) = a_{D_{z,i}}\mathrm{Disc}(f_i)$ where $\mathrm{Disc}(f_i)$ denotes the discriminant of f_i. It is clear that $d_{p,i} = D_{z,i}$ for a point p if and only if $a_{D_{z,i}}(p) \neq 0$. From the definition of the discriminant, $k_{p,i} = 0$ for a regular point p if and only if $\mathrm{Disc}(f_i)(p) \neq 0$. □

This opens the door to apply a combinatorial minimization of any (d,k)-arrangement. For what follows, we assume that each cell Γ of a (d,k)-arrangement is equipped with its local degrees $d_{\Gamma,i}$ and $k_{\Gamma,i}$ as data. As post-processing, one can remove all edges and vertices away from τ_S, and remove vertices on τ_S that have exactly two adjacent edges, and both edges have the same local degree and local gcd degree as the vertex (and merge the adjacent edges). A similar idea of clustering a cad has been proposed by Arnon [Arn88], but, in contrast, (d,k)-invariance models a strictly weaker condition. Thus, it produces larger cells.

As we are aiming for an actual implementation using CGAL's `Arrangement_2` package to construct the (d,k)-arrangement, we present Algorithm 5.2 that turns the actual post-processing into a bottom-up construction of the (d,k)-arrangement, which lowers the size of intermediate arrangements.[48] The main tool for Algorithm 5.2 is CGAL's possibility to *overlay* arrangements. Given arrangements $\mathcal{A}_1$ and $\mathcal{A}_2$, the overlay is the union $\mathcal{A}_3$ of both arrangements. In addition, we can ensure that each cell of $\mathcal{A}_3$ knows from which cells of $\mathcal{A}_1$ and $\mathcal{A}_2$ it originates.

Remark (on Algorithm 5.2). There are two optimizations: First, to set the local gcd degree one only has to consider those degrees d that appear as the local degree of at least one cell. Second, the inner iteration over the k's is stopped as soon as all cells of degree d know their local gcd degree.

$\mathcal{A}_{\{S_i\}}$, as constructed with Algorithm 5.2, basically consists of the overlay of the leading coefficient curve and the discriminant curve of f_i (compare Lemma 5.40). However, this curve is subdivided by additional points in order to ensure (d,k)-invariance. We admit, that the approach is in similar spirit as the improved projection operators in cad computation; see the work of McCallum [McC] and the slight improvement by Brown [Bro01b]. Instead of (d,k)-invariance, they introduce *order-invariance* and show that such cells also ensure delineability. Consequently, the non-leading coefficients and the principal Sturm-Habicht coefficients are superfluous for Theorem 5.37. However, the knowledge about the local degree and the local gcd degree in our (d,k)-invariant decomposition enables fast methods in the lifting step, as we learn in §5.3.3.

We now consider a set $\mathcal{S} := \{S_1, \ldots, S_n\}$ of algebraic surfaces. The definition of local degrees naturally extends:

Definition 5.41 (Local multi-regularity). Given a point $p = (p_x, p_y) \in \mathbb{R}^2$. We call it $(d_1, \ldots, d_n)$-*regular* with respect to $\mathcal{S} := \{S_1, \ldots, S_n\}$ if and only if p is $d_{p,i}$-regular with respect to S_i. Note that having some $d_{p,i} = -\infty$ is allowed.

We first concentrate on $n = 2$, that is, we restrict to two surfaces only. Afterwards it is easy to define an arrangement for an arbitrary number of surfaces for our purposes. Let $S_1, S_2 \in \mathcal{S}$ be two surfaces and $\mathcal{A}^*_{\{S_1,S_2\}}$ be the overlay of the arrangements $\mathcal{A}_{\{S_1\}}$ and $\mathcal{A}_{\{S_2\}}$. Then, for each cell of $\mathcal{A}^*_{\{S_1,S_2\}}$ the regularity with respect to $\{S_1, S_2\}$ stays invariant.

We next show that there exists a refinement $\mathcal{A}_{\{S_1,S_2\}}$ of $\mathcal{A}^*_{\{S_1,S_2\}}$, such that on each component $\Gamma \in \mathcal{A}_{\{S_1,S_2\}}$ the z-patterns $W_{\{S_1\},p}$, $W_{\{S_2\},p}$, and $W_{\{S_1,S_2\},p}$ stay the same. For this purpose we have to introduce some further notation based on subresultant sequences.

[48] Arrangements of large size are usually more costly to construct than such with a small number of cells.

Algorithm 5.2. Construct clustered $\mathcal{A}_{\{S\}}$ with low-size intermediate arrangements

INPUT: Algebraic surface S of degree D
OUTPUT: $\mathcal{A}_{\{S\}}$ with minimal number of (d,k)-invariant cells

1. Computing the arrangement $\mathcal{A}$ induced by the projected silhouette τ_S only. Remember that τ_S may be not square-free. To handle this case, we typically apply a square-free factorization (see §2.1.1) and compute $\mathcal{A}_{\tau_S}$ by overlaying individual arrangements induced by the resulting square-free curves. In this case, each resulting edge of $\mathcal{A}$ can be assigned with the multiplicity of the corresponding factor of τ_S.

2. Each face of $\mathcal{A}$ receives the values $(D_z, 0)$ respecting Lemma 5.40.

3. Decompose $\mathcal{A}$ such that each resulting cell has invariant local degree by repeating the following steps for $d = D_z, \ldots, 0$:
   - Compute the arrangement $\mathcal{A}_d$ induced by α_d; as above, we actually consider the square-free factorization of a_d.
   - Overlay $\mathcal{A}$ with $\mathcal{A}_d$, the result is $\mathcal{A}'$.
   - Remove all vertices and edges of $\mathcal{A}'$ that originate from a face of $\mathcal{A}$.
   - Remove also all vertices of $\mathcal{A}'$ that originate from an edge of $\mathcal{A}$ whose local degree has already been set.
   - For each cell of $\mathcal{A}'$ that originates from a face of $\mathcal{A}_d$, and whose degree is not set yet, assign its local degree to d.
   - Set $\mathcal{A} \leftarrow \mathcal{A}'$ and proceed with the next iteration.

4. Set the local degree of all cells which are not yet set to $-\infty$, as S must be vertical above these cells (vertices).

5. It remains to decompose $\mathcal{A}$ into (d,k)-invariant cells. The strategy is similar: Iterate over the degrees and overlay with the corresponding principal Sturm-Habicht coefficient curves $\sigma_{d,k}$. Thus, repeat for $d = D_z, \ldots, 1$: Repeat for $k = 0, \ldots, d-1$:
   - Compute the arrangement $\mathcal{A}_{d,k}$ induced by $\sigma_{d,k}$; as above, we actually consider the square-free factorization of $\mathrm{stha}_k(f_{(d)})$.
   - Overlay $\mathcal{A}$ with $\mathcal{A}_{d,k}$, the result is $\mathcal{A}'$.
   - Remove all vertices and edges of $\mathcal{A}'$ that originate from a face of $\mathcal{A}$.
   - Remove all vertices of $\mathcal{A}'$ that originate from an edge of $\mathcal{A}$ whose local gcd degree has already been set, or whose local degree does not equal d.
   - For each cell of $\mathcal{A}$ that lies on a face of $\mathcal{A}_{d,k}$, whose local degree is d, and whose local gcd degree is not yet set, assign the local gcd degree to k.
   - Set $\mathcal{A} \leftarrow \mathcal{A}'$ and proceed with the next iteration.

Definition 5.42 (Local multi-degradation). Let $p = (p_x, p_y)$ be a (d_1, d_2)-regular point, with $d_1 \neq -\infty$ and $d_2 \neq -\infty$. We say that p has *degradation* $k_{p,1,2}$ with respect to $\{S_1, S_2\}$ if and only if

$$k_{p,1,2} := \deg_z(\gcd(f_1(p), f_2(p))) - 1 \,[= \min \mathcal{K}_{1,2}]$$

with $\mathcal{K}_{1,2} := \{k = 0, \ldots, \min\{d_1, d_2\} - 1 \mid \operatorname{sres}_k((f_1)_{(d_1)}(p,z), (f_2)_{(d_2)}(p,z), z) \neq 0\}$. Note that if $k_{p,1,2} = -\infty$ we have $\mathcal{K}_{1,2} = \emptyset$.

Let $\Gamma \in \mathcal{A}^*_{\{S_1,S_2\}}$ be a cell of regularity (d_1, d_2), then, there exist common minimal degradations $k_{1,\Gamma}$, $k_{2,\Gamma}$ for Γ. If Γ is a face, $k_{\Gamma,1} = k_{\Gamma,2} = 0$. Otherwise, $k_{\Gamma,i}$ is defined and ≥ 0 if $d_{\Gamma,i} \neq -\infty$. However, Γ may not yet be invariant with respect to common roots of $f_1(p,z)$ and $f_2(p,z)$. Remember from Proposition 2.7 that $f_1(p,z)$ and $f_2(p,z)$ only have a common (complex) root, if $\operatorname{Res}_z(f_1, f_2, z)$ vanishes. That is, the two surfaces may only intersect above some p if $k_{p,1,2} \geq 0$. Following, $p \in V_{\mathbb{R}}(\operatorname{Res}_z(f_1, f_2))$ is a necessary condition for having an intersection of the surfaces over p. Points having this condition are given by the following curve:

Definition 5.43 (Projected intersection). The *projected intersection* τ_{0,S_1,S_2} of $\{S_1, S_2\}$ is defined by $\operatorname{sres}_0(f_1, f_2, z) = \operatorname{Res}_z(f_1, f_2)$.

To overlay the (d_1, d_2)-regular arrangement $\mathcal{A}^*_{\{S_1,S_2\}}$ with the projected intersection is the main step in Algorithm 5.3. However, it still does not ensure that the obtained cells are invariant with respect to z-patterns. $p \in Res_z(f_1, f_2)$ only means that $g_p := \gcd(f_1(p,z), f_2(p,z))$ is non-trivial. Only if g_p has real roots then f_1 and f_2 have real intersections over p. As before, the number and distribution of (complex) roots of polynomials $f_1(p,z)$ and $f_2(p,z)$ continuously depend on p, and so the roots of g_p. The distribution of its roots (i. e., their number and multiplicities) changes only where the degrees or the factorizations of f_i or g alter; see Proposition 2.9 in combination with Theorem 2.10 and Theorem 5.37. Fortunately, the subresultant sequence gives a (deliberate) algebraic indication for a change in g_p's degree — and thus a possible change in the number of real intersections.

We already considered the individual degradations of f_1 and f_2. But, we still have to refine the cells induced by τ_{0,S_1,S_2} with respect to further degradations, that is, with respect to curves $\tau_{k,S_1,S_2} = V_{\mathbb{R}}(\operatorname{sres}_k(f_1, f_2, z)$. Note that if $k_{p,1,2} = 0$ implies that $\deg_z(g(p)) = 1$, that is, there is one real intersection of S_1 and S_2 over p (by degree there cannot be a multiple or complex one). Remember that $k_{p,1,2}$ constitutes an upper bound on the number real intersections of S_1 and S_2 over p. These observations finally lead to Algorithm 5.3.

Remark (on Algorithm 5.3). Observe that our construction of $\mathcal{A}_{\{S_1,S_2\}}$ is conservative in the sense that it might keep cells having the same number (and order) of real intersection over it as its neighbored cells. The reason is simply that the algebraic indication that we rely on does not ignore complex roots.

However, by construction it is ensured that the local degrees for each individual surface stays invariant in each cell, and the intersection pattern of two surfaces over a given cell also cannot change. We admit, that at this point the local real degrees are not yet determined, though they are theoretically fixed in terms of the others. Thus, Condition 5.16 is also fulfilled for algebraic surfaces.

Remark. As for a single surface, the construction is similar to what is done for a cylindrical

Algorithm 5.3. Construct clustered $\mathcal{A}_{\{S_1,S_2\}}$

INPUT: Algebraic surfaces S_1, S_2 of degree D_1, D_2
OUTPUT: $\mathcal{A}_{\{S_1,S_2\}}$ with minimal number of invariant cells with respect to local degrees and degradations

1. Compute overlay of $\mathcal{A}_{\{S_1\}}$ and $\mathcal{A}_{\{S_2\}}$ and call it $\mathcal{A}^*_{\{S_1,S_2\}}$.

2. Compute the arrangement $\mathcal{A}_{\tau_{0,S_1,S_2}}$ of τ_{0,S_1,S_2}. As above, remember that $\mathcal{A}_{\tau_{0,S_1,S_2}}$ can be composed of the overlay of τ_{0,S_1,S_2}'s square-free factors. Thus, each of $\mathcal{A}_{\tau_{0,S_1,S_2}}$'s edges can be assigned with the multiplicity of the corresponding factor.

3. Overlay $\mathcal{A}^*_{\{S_1,S_2\}}$ with $\mathcal{A}_{\tau_{0,S_1,S_2}}$. The result is $\mathcal{A}_{\{S_1,S_2\}}$. However, $k_{\Gamma,1,2}$ for $\Gamma \in \mathcal{A}_{\{S_1,S_2\}}$ is still unknown and some edges can even split further:

4. Set $k_{\Gamma,1,2} = 0$ for all cells Γ that originate from vertices and edges in $\mathcal{A}_{\tau_{0,S_1,S_2}}$, and $k_{\Gamma,1,2} = -\infty$ for all other cells (meaning invalid).
   If Γ is such a special vertex or edge let $d_{1,\Gamma}$ and $d_{2,\Gamma}$ denote its local degrees with respect to S_1 and S_2.

5. For such a vertex at point p we have to compute the correct (and maybe larger) $k_{p,1,2}$. For $1 \leq k < \min\{d_{p,1}, d_{p,2}\}$:
   - Check whether p lies on $\tau_{k,1,2}$ (or one of its square-free factors, and note that $\tau_{k,1,2}$ depends on $d_{p,1}$ and $d_{p,2}$). If so, continue, otherwise set $k_{p,1,2} = k$ and stop. Note that this point-on-curve test encodes whether $\mathrm{sres}_k(f_1, f_2, z)(p) = 0$.

6. For such an edge E two options are possible. Again, for $1 \leq k < \min\{d_{E,1}, d_{E,2}\}$:
   - Check if E is part of $\tau_{k,1,2}$, by testing whether the polynomial defining the curve that supports E has a common factor with $\mathrm{sres}_k(f_1, f_2, z)$. If so, set $k_{E,1,2} = k$ and continue with next k.
   - Otherwise, check if E has a finite number of intersections with $\tau_{k,1,2}$. Split E at them which creates new vertices. For each such vertex at point p, assign $k_{p,1,2} = k$ and proceed with the described handling for vertices.

algebraic decomposition; see §2.1.6. In contrast, we explicitly handle the projection as a planar arrangement and benefit from the possibilities to combinatorially cluster cells and to attach additional data, such as the multiplicities of τ_{0,S_1,S_2} or degradations $k_{1,2}$.

The extension to more than two surfaces is natural:

Definition 5.44 ($\mathcal{A}_\mathbb{S}$). Let $\mathbb{S} = \{S_1, \ldots, S_n\}$ be algebraic surfaces and let $\mathcal{A}_{\{S_i,S_j\}}$, $i \neq j$ the arrangement as constructed with Algorithm 5.3. Then, we define $\mathcal{A}_\mathbb{S}$ to be the overlay of all arrangements $\mathcal{A}_{\{S_i,S_j\}}$.

By how we constructed $\mathcal{A}_{\{S_i,S_j\}}$ it can be seen that $\mathcal{A}_\mathbb{S}$ consists of cells $\Gamma \in \mathcal{A}_\mathbb{S}$ such that $W_{\mathbb{S},p}$ is identical for all $p \in \Gamma$. Thus, $\mathcal{A}_\mathbb{S}$ is m_i-invariant for $1 \leq i \leq n$ and $m_{i,j}$-invariant for any $1 \leq i, j \leq n$, $i \neq j$.

5.2.3. Continuation

In this section we show that Condition 5.9 is fulfilled for an algebraic surface S_i. Additionally, we learn that a lifted face F of some $\mathcal{A}_{\{S_i\}}$ can be incident to a whole interval along some ℓ_p for $p \in \overline{F}$ and $p \in V_i$, which helps in §5.2.4 to decompose S_i and to proof the boundary property of the decomposition.

As we already know that Condition 5.16 holds for $p \notin V_i$, Condition 5.9 can be verified by the fact that the roots of a polynomial continuously depend on its coefficients: Remember that in this case $Z_{p,i} = \{z \in \mathbb{R} \mid 0 = f_i(p_x, p_y, z) \in \mathbb{R}[z]\}$. Thus, for some $\Gamma \in \mathcal{A}_{\{S_i\}}$, $p \in \partial\Gamma$ and the sequence of points $p_t \in \Gamma$ with $\lim_{t\to\infty} p_t = p$, we must get $\left\{\lim_{t\to\infty} z_{p_t,i,-1}, \ldots, \lim_{t\to\infty} z_{p_t,i,m_{\Gamma,i}}\right\} \subset Z_{p,i}$ and $\lim_{t\to\infty} z_{p_t,i,l_1} \leq \lim_{t\to\infty} z_{p_t,i,l_2}$ for $l_1 < l_2$. The same argument applies to Condition 5.17 for $p \notin V_i \cup V_j$ and $\mathcal{A}_{\{S_i,S_j\}}$.

The situation is different when we have $p \in V_i$. In what follows, it does not make a difference whether we have $\mathcal{A}_{\{S_i\}}$, $\mathcal{A}_{\{S_i,S_j\}}$, or even $\mathcal{A}_\mathbb{S}$. We only fix an arbitrary surface S_i. The process is identical for any other surface.

We assume that p is a vertex in $\mathcal{A}_\mathbb{S}$ and for a neighborhood of p, none of the surfaces S_j, $j \neq i$ contains a vertical line, except at p. Now we consider a sequence of points $p_t \in \Gamma \subset \mathcal{A}_\mathbb{S}$ that converges against p. Then, we have to determine possible limits of their l-th lifts $(p_t, z_{p_t,i,l}) \subset p_t \times Z_{p_t,i} \subset p_t \times \mathcal{Z}_{p_t,\mathbb{S}}$. If all p_t lie on an edge E, then the limit is uniquely given as endpoint (above p) of the l-th lift of E with respect to S_i.

For a face $F \in \mathcal{A}_\mathbb{S}$, adjacent to p, it can happen that the limits of the l-th lifts of two different sequences $p_t, p'_t \in F$ are distinct.

Theorem 5.45. *Given a surface $S_i \in \mathbb{S}$, $\Gamma \in \mathcal{A}_\mathbb{S}$, and two sequences $p_t, p'_t \in \Gamma$ with $p = \lim_{t\to\infty} p_t = \lim_{t\to\infty} p'_t$ and for some $0 \leq l < m_{\Gamma,i}$ we have $z_0 = \lim_{t\to\infty} z_{p_t,i,l}$, $z_1 := z_{p'_t,i,l}$. Then, for any z^* in between z_0 and z_1 there exists a sequence $p^*_t \in \Gamma$ with $p = \lim_{t\to\infty} p^*_t$ and $z^* = \lim_{t\to\infty} z_{p^*_t,i,l}$.*

Proof. If $z_0 = z_1$ there is nothing to prove, thus we can assume $z_0 < z_1$. We can further assume that $|p_t - p|$ and $|p'_t - p|$ are monotone. As $\mathcal{A}_\mathbb{S}$ is expected to be m_i-invariant it follows that there exists an ε_0 such that $U_\varepsilon \cap \Gamma$ is connected for all $\varepsilon < \varepsilon_0$ and $U_\varepsilon := \{q \in \mathbb{R}^2 : |q - p| \leq \varepsilon\}$. Thus, we can assume that $U_t \cap \Gamma$ is connected, where $U_t := \{q \in \mathbb{R}^2 \,|\, |q - p| \leq 2\max\{|p_t - p|, |p'_t - p|\}\}$.

Now we consider a continuous path $\Pi_t \subset (\Gamma \cap U_t)$, that connects p_t and p'_t. As the roots of $f_i(q,z)$ continuously depend on the point $q \in \Gamma$, for each z_t^* in between $z_{p_t,i,l}$ and $z_{p'_t,i,l}$ we can choose a $p_t^* \in \Pi_t$ that lifts to $z_{p_t^*,i,l} = z_t^*$. As $z_0 = \lim_{t\to\infty} z_{p_t,i,l}$ and $z_1 := z_{p'_t,i,l}$ there exists a $t_0 \in \mathbb{N}$ such that $z^* \in [z_{p_t,i,l}, z_{p'_t,i,l}]$ for all $t > t_0$. Thus, we can choose $z_t^* = z^*$ from which it follows that $p_t^* \in \Gamma$ converges against p and fulfills $z^* = \lim_{t\to\infty} z_{p_t^*,i,l}$. □

Theorem 5.45 shows that for any element $\Gamma \in \mathcal{A}_\mathbb{S}$, adjacent to p, and any $l \in \{0,\dots,m_{\Gamma,i}-1\}$, the set of limits $\lim_{t\to\infty} z_{p_t,i,l}$ ($p_t \in \Gamma$ a sequence that converge against p) is an interval of $I_{\Gamma,i,l} \subset \mathbb{R}$. Thus, algebraic surfaces fulfill Condition 5.9 and following the specifications for $Z_{p,i}$ constituted in Definition 5.10.

Concerning Condition 5.17, we can apply the same proof idea. Actually, Theorem 5.45 in combination with Condition 5.13 show the desired result for algebraic surfaces. The reason is that lifts of a face $F \in \mathcal{A}_\mathbb{S}$ uniquely belong to a single surface.

However, it is not yet discussed, how to compute the entries of $Z_{p,i}$ for $p \in V_i$, and $Z^|_{p,i,j}$ for $p \in V_i \cup V_j$. We give the details on this task in §5.4.2. The construction described there provides also the proofs for the following theorems. We require them to bound the complexity of our cell decomposition that we introduce next in §5.2.4.

Theorem 5.46 (Complexity of z-fiber for $p \in V_i$). *Let S_i be an algebraic surface of degree D and $p \in V_i$. Then $|Z_{p,i}| \in O(D^3)$.*

Theorem 5.47 (Complexity of $Z^|_{p,i,j}$). *Let S_i, S_j be two algebraic surfaces of degree D and $p \in V_i \cup V_j$. Then $|Z^|_{p,i,j}| \in O(D^3)$.*

Corollary 5.48 (Complexity of multi-surface z-fiber for $p \notin \mathcal{V}$). *Let $\mathbb{S}$ be a set of n algebraic surfaces with maximal degree D and $p \notin \mathcal{V}$. Then, $|\mathcal{Z}_{p,\mathbb{S}}| \in O(nD)$.*

Proof. Remember the definition of $\mathcal{Z}_{p,\mathbb{S}}$ for such p. Theorem 5.34 gives $|Z_{p,i}| \leq D$. There are n surfaces. □

Corollary 5.49 (Complexity of multi-surface z-fiber for $p \in \mathcal{V}$). *Let $\mathbb{S}$ be a set of n algebraic surfaces with maximal degree D and $p \in \mathcal{V}$. Then, $|\mathcal{Z}_{p,\mathbb{S}}| \in O(n^2D^3)$.*

Proof. Remember the definition of $\mathcal{Z}_{p,\mathbb{S}}$. Theorem 5.46 gives $|Z_{p,i}| \in O(D^3)$. There are n surfaces. Theorem 5.47 gives $Z^|_{p,i,j} \in O(D^3)$. There are $O(n^2)$ different pairs of surfaces. □

5.2.4. Stratifications and their complexities

We next show that the cell decompositions we introduced in §5.1 constitute stratifications of algebraic surfaces.

Definition 5.50 (Stratification; see [BPR06, §5.5]). Let S_i be a surface. A *stratification* of S_i is a decomposition of S_i into cells such that

- each cell is a smooth subvariety of S_i of dimension 0, 1, or 2, and
- the boundary of a cell is given by the finite union of other cells; also known as the *boundary property*.

The cells of a stratification are also called strata.

Compare also the similar notion of a CW complex; see §2.1.7 and [Mas67], [Bre95]. In addition, we give complexity bounds for each of these decomposition in terms of the number of surfaces and their algebraic degree.

As previously mentioned $\Omega_{\{S_i\}}$ should fulfill the *boundary property*. An equivalent statement is, that for any two cells M_1, M_2 with $\dim(M_1) < \dim(M_2)$, we must have $M_1 \cap \overline{M_2} = \emptyset$ or $M_1 \subset \overline{M_2}$. In the previous case, the two cells of S_i are not related, while in the latter case, we call them *adjacent*. To check whether M_1 is adjacent to M_2 can be expressed with respect to an arbitrary point $p \in M_1$: Two cells are adjacent if and only if $p \in \overline{M_2}$. We first assume that $V_i = \emptyset$.

Theorem 5.51. *Let $M_1, M_2 \in \Omega_{S_i}$ with $\dim(M_1) < \dim(M_2)$ and $\Gamma_1, \Gamma_2 \in \mathcal{A}_{\{S_i\}}$ their corresponding projections onto the plane. If Γ_1 has local degree $d_{\Gamma_1,i} \neq -\infty$ and $M_1 \cap \overline{M_2} \neq \emptyset$, then $M_1 = \overline{M_2} \cap (\Gamma_1 \times \mathbb{R})$.*

Proof. Let M_2 be the l_2-th lift of Γ_2 and $p = (p^*, z_0) \in \overline{M_2} \cap (\Gamma_1 \times \mathbb{R})$ an arbitrary point, contained in a lift $\Gamma_1^{(l_1,i)}$ of Γ_1. For the lifts $p^{*(l,i)}$ of p^* we choose a box neighborhood B_{p^*} of p^* and also disjoint boxes $B_1, \ldots, B_{m_{\Gamma_1,1}}$ lying above B_{p^*} with $B_i = B_{p^*} \times \left[p^{*(l,i)} - \delta, p^{*(l,i)} + \delta\right]$ and a $\delta > 0$. We can assume that B_{p^*} and δ are chosen such that the l-th lift of $\Gamma_1 \cap B_{p^*}$ is contained in B_l. For B_{p^*} small enough, it follows that the l_2-th lift of $B_{p^*} \cap \Gamma_2$ is also contained in B_{l_1} as $p \in B_{l_1} \cap \overline{M_2}$. As a direct consequence $((B_{p^*} \cap \Gamma_1) \times \mathbb{R}) \cap \overline{M_2}$ is the l_1-th lift of $(B_{p^*} \cap \Gamma_1)$. Now for any two points p_1^* and p_2^* on Γ_1 there exists a compact path Π on Γ_1, which connects them. Then, we consider an open covering of Π with local neighborhoods $B_{p'}$, $p' \in \Gamma$, such that $((B_{p'} \cap \Gamma_1) \times \mathbb{R}) \cap \overline{M_2}$ is the $l_{p'}$-th lift of Γ_1. Then, from restricting to a finite partial covering it follows that $l_{p'} = l_1$ for all p', thus $\Gamma_1^{(l_1),i} = \overline{M_2} \cap (\Gamma_1 \times \mathbb{R})$. Now $M_1 \cap \overline{M_2} \neq \emptyset$ exactly if $M_1 = \Gamma_1^{(l_1,i)}$. □

Theorem 5.51 assumed that S_i does not contain a vertical line. Thus, we turn to surfaces that include vertical lines and decompose them into cells according to Definition 5.28. It also must be shown that this extended decomposition $\Omega_{\{S_i\}}$ is still a stratification, that is, the boundary property is fulfilled. Remember that S_i only contains a finite set of vertical lines. Observe that not splitting vertical lines is insufficient as such an ℓ_p, in general, constitutes a superset of a boundary of a lifted face. However, $\Omega_{\{S_i\}}$ as in Definition 5.28 splits ℓ_p according to the interval boundaries defined by the following implication of Theorem 5.45:

Corollary 5.52. *Let S_i contain the vertical line ℓ_p and $F \in \mathcal{A}_{\{S_i\}}$ be a face, which is adjacent to p. Then for any surface patch $F^{(l,i)}$ there exists an interval $I(F^{(l,i)}) \subset \mathbb{R}$, such that $p \times I(F^{(l,i)}) = \overline{F^{(l,i)}} \cap \ell_p$.*

Thus, the boundary property is ensured by how we constructed $Z_{p,i}$ according to Condition 5.9 which is fulfilled by Corollary 5.52.

The boundary property of $\Omega_{\mathcal{S}}$ is given by the facts, that with respect to a single surface S_i the arrangement $\mathcal{A}_{\mathcal{S}}$ is a refinement of $\mathcal{A}_{\{S_i\}}$ and a lift of any curve τ_{S_j} or τ_{0,S_i,S_j} for any $j \neq i$ only splits a cell M of $\Omega_{\{S_i\}}$. Note that M already has the boundary property. If $V_i = \emptyset$, the boundary property follows by Theorem 5.51 applied to S_i and $\mathcal{A}_{\mathcal{S}}$. Otherwise, we have to consider the case that M can be adjacent to a vertical line at p with $I(M)$ being non-degenerate and that M is split by a lift of some τ_{S_j} or τ_{0,S_i,S_j} for any

$j \neq i$ into M_1 and M_2. Note that by Condition 5.17 the z-coordinate of the endpoint of such a curve's lift is included in $\mathcal{Z}_{p,\{S_i,S_j\}}$. Thus, the way Ω_S splits ℓ_p ensures that $I(M_1)$ and $I(M_2)$ are reflected by the decomposition and thus, the boundary property is fulfilled.

As promised we also analyze the complexities of the cell decompositions. Again, we start with a single surface S_i defined by f_i having total degree D and its (d,k)-arrangement $\mathcal{A}_{\{S_i\}}$ as constructed by Algorithm 5.2. We first show, that the complexity of $\mathcal{A}_{\{S_i\}}$ is not greater than that for τ_S.

Theorem 5.53. *The number of cells of $\mathcal{A}_{\{S_i\}}$ is $O(D^4)$.*

Proof. First remark that $D = D_i$. It suffices to count the number of vertices, as arrangements form planar graphs. For such graphs, the number of edges and faces linearly depend on them by the Euler formula. First observe that the projected silhouette τ_{S_i} is of degree at most D^2. By Bézout's theorem it has at most D^4 critical points. It remains to show that its decomposition with respect to α_d and $\sigma_{d,k}$ does not create more than D^4 new vertices.

Consider the decomposition of τ_{S_i} into irreducible components γ_j with degree ν_j, and fix one $\gamma = \gamma_j$ of degree ν. In the execution of the algorithm, we only introduced new vertices for γ (that are not removed in the same iteration) in two iteration steps:

First, when a coefficient curve α_d does not contain the whole curve γ. This introduces at most $\nu \cdot D$ many vertices. All further coefficient curves $\alpha_{d-1}, \ldots, \alpha_0$ do not introduce new vertices on γ, since the local degree of all edges for γ is set to d.

Second, new vertices are introduced when a Sturm-Habicht polynomial $\mathrm{stha}_k(f_{(d)})$ does not contain the whole curve γ. This introduces at most $\nu \cdot D^2$ many new vertices. All further Sturm-Habicht curves $\mathrm{stha}_{k-1}(f_{(d)}, \ldots, \mathrm{stha}_0(f_{(d)})$ do not introduce new vertices on γ, since the local gcd degree of all edges for γ is set to k.

Finally, each γ gets at most $O(\nu_j \cdot D^2)$ many new vertices, and the ν_j sum up to D^2. □

Corollary 5.54. *For a surface S_i without vertical line, the number of cells in $\Omega_{\{S_i\}}$ is $O(D^5)$.*

The proof is given by Theorem 5.34 and Theorem 5.53. An implication is that we achieve a topological description of the surface using $O(D^5)$ many sample points. A corresponding cad consists of $\Omega(D^7)$ cells, due to its vertical decomposition strategy in the plane. However, the advantage of having a small number of cells implies less topological information, for example, to replace (lifted) edges by straight-lines requires further processing. How to compute the adjacency relation of $\Omega_{\{S_i\}}$'s cells is presented in §5.4.2.

The complexity of $\Omega_{\{S_i\}}$ with $V_i \neq \emptyset$ can also be stated. Note that we need to compute $Z_{p,i}$ for $p \in V$. We learn in §5.4.2 that we actually compute a superset of $Z_{p,i}$ that is algebraically defined by the real roots of three polynomials. That is, for $p \in V_i$, it holds that $\Omega_{\{S_i\}}$'s decomposition of ℓ_p is finer than technically required by Condition 5.9. However, Theorem 5.46 still holds to finally proof that $\Omega_{\{S_i\}}$ stays with the known worst-case complexity:

Theorem 5.55. *The number of cells of $\Omega_{\{S_i\}}$ is $O(D^5)$.*

Proof. For lifts not related to $p \in V_i$, Corollary 5.54 still applies. It only remains to bound the number of cells introduced for vertical lines by $O(D^5)$: Observe that $\deg(f_i) \leq D$ implies $|V_i| \leq D^2$. It remains to apply Theorem 5.46. □

We turn to the case of multiple surfaces. Note that $\forall D_i : D_i \leq D$. We already learned in §5.2.2 that $\mathcal{A}_{\mathcal{S}}$ composed of all $\mathcal{A}_{\{S_i,S_j\}}$ for two surfaces S_i, S_j, $i \neq j$ forms the basis for $\Omega_{\mathcal{S}}$. In order to derive $\Omega_{\mathcal{S}}$'s complexity, we first have to proof a result on $\mathcal{A}_{\mathcal{S}}$'s complexity:

Theorem 5.56. *The number of cells of $\mathcal{A}_{\mathcal{S}}$ is $O(n^4D^4)$.*

Proof. Again, it suffices to count the number of vertices, for the same reason as in the proof of Theorem 5.53.

We perform the counting in two steps: First, we consider the vertices of the individual arrangements that occur. Second, we analyze how many vertices occur during overlays.

For a single surface S_i, the complexity of $\mathcal{A}_{\{S_i\}}$ is mainly driven by τ_{S_i}, that is, an algebraic curve of degree at most D^2. The arrangement has, following Theorem 5.53, $O(D^4)$ vertices. When constructing $\mathcal{A}_{\{S_i,S_j\}}$ for two surfaces S_i, S_j, $i \neq j$, we additionally consider τ_{0,S_i,S_j}. This curve is also of degree at most D^2 and thus has at most D^4 critical points. During the execution of Algorithm 5.3, τ_{0,S_i,S_j} gets segmented by some $\mathrm{sres}_k(f_i, f_j, z)$ with $k \geq 1$. The maximal number of segmentation vertices occurs if $\mathrm{sres}_0(f_i, f_j, z)$ gets segmented by $\mathrm{sres}_1(f_i, f_j, z)$. Bounding $\deg(\mathrm{sres}_1(f_i, f_j, z))$ by D^2 is sufficient (though not very tight) as is allows to conclude that the number of segmentation vertices of τ_{0,S_i,S_j} is upper bounded by D^4. Thus, both the refined τ_{S_i} and the refined τ_{0,S_i,S_j} have at most $O(D^4)$ critical points.

It remains to give a bound on the number of vertices that are introduced with respect to overlays of arrangements. We only have to consider how often curves τ_{S_i} and τ_{0,S_i,S_j} can intersect.[49] Remember that $\deg(\mathrm{sres}_0(f_i, \frac{\partial f_i}{\partial z}, z)) \leq D^2$ and also $\deg(\mathrm{sres}_0(f_i, f_j, z)) \leq D^2$. Thus, two such curves intersect by Bézout's theorem in at most D^4 points. There are n projected silhouettes, and $O(n^2)$ projected intersection curves. As we overlay all of them, we have to consider each pair and thus get up to $O(n^4D^4)$ new intersection points.

In total, $\mathcal{A}_{\mathcal{S}}$ has complexity $O(nD^4) + O(n^2D^4) + O(n^4D^4) = O(n^4D^4)$. □

The next corollaries are simple implications of Theorem 5.56, Corollaries 5.48 and 5.49, and the fact that $|\mathcal{V}| \leq nD^2$.

Corollary 5.57. *For a set $\mathcal{S} = \{S_1, \ldots, S_n\}$ of algebraic surfaces of total degree D, the number of cells in $\Omega_{\mathcal{S}}$ is $O(n^5D^5)$.*

Corollary 5.58. *For algebraic surfaces S_1, S_2 of total degree D, the number of cells of $\mathcal{A}_{\{S_1,S_2\}}$ is $O(D^4)$ and the number of cells in $\Omega_{\{S_1,S_2\}}$ is $O(D^5)$.*

Assume we replace $\mathcal{A}_{\mathcal{S}}$ in Definition 5.29 by $\mathcal{A}'_{\mathcal{S}}$ that results from applying Algorithm 5.1 on $\mathcal{A}_{\mathcal{S}}$. By Proposition 5.32 we obtain a cell decomposition $\Omega'_{\mathcal{S}}$ consisting of simply connected cells, whose complexity can also be bounded:

Corollary 5.59. *For $S_i \in \mathcal{S}$, $|\Omega'_{\{S_i\}}| \in O(D^5)$ and $|\Omega'_{\mathcal{S}}| \in O(n^5D^5)$.*

5.2.5. Semi-algebraic surfaces

Let us also shortly refer to semi-algebraic surfaces, that constitute possible input. A *semi-algebraic surface* $S_{\geq}$ is defined by a polynomial equation $f = 0$ that is refined by

[49] Mind that the coefficients curves α_l only segment τ_{S_i}.

a sequence of polynomial inequalities $g_j \geq 0$, with $1 \leq j \leq r$ for some r. Such a semi-algebraic surface is closed, and thus fulfills the conditions expected by the framework. In particular, the arrangement $\mathcal{A}_{\{V_{\mathbb{R}}(f),V_{\mathbb{R}}(g_1),\ldots,V_{\mathbb{R}}(g_r)\}}$ constitutes an arrangement $\mathcal{A}_{\{S_{\geq}\}}$ fulfilling Condition 5.7 for $S_{\geq}$. However, this decomposition of the plane is far from being optimal, as most of its cells are redundant due to the fact, that $\mathcal{A}_{\{V_{\mathbb{R}}(f),V_{\mathbb{R}}(f_1),\ldots,V_{\mathbb{R}}(f_k)\}}$ considers the pair-wise combination of all involved surfaces defined by polynomials. As for $S_{\geq}$ we are only interested in changes of the z-fiber of $S = V_{\mathbb{R}}(f)$ with respect to a single inequality g_j, the following incremental strategy is encouraged to construct $\mathcal{A}_{\{S_{\geq}\}}$. Start with $\mathcal{A}_{\{V_{\mathbb{R}}(f)\}}$ and refine it with respect to the projected intersections of $V_{\mathbb{R}}(f)$ with $V_{\mathbb{R}}(g_j)$ for increasing j (mind local degrees and degradations). It is still possible to simplify the obtained $\mathcal{A}_{\{S_{\geq}\}}$, for example, by merging neighboring cells that do not comprise a projected point of $S_{\geq}$, or by removing a projected intersections of $V_{\mathbb{R}}(f)$ and some $V_{\mathbb{R}}(g_j)$ if it lies in a face of $\mathcal{A}_{\{V_{\mathbb{R}}(f)\}}$ and its incident higher-dimensional cells carry the same z-pattern. We omit further details and refer in the further discussion only in exceptional cases again to semi-algebraic surfaces.

5.3. Implementation in a framework

It is common, that algorithms in this area of research are lacking their implementation, or that certain degeneracies are excluded, such as vertical lines or singularities. We do not join this queue. In contrast, we provide a C++-implementation for the tasks listed in §5.1. It is part of the software projects CGAL [3] and EXACUS [6]; see also §2.2.3 and §2.2.4. We admit that the implementation has become feasible by relying on existing code of the projects. Our C++-implementation consists of two related components, that split combinatorial parts from surface-specific geometric tasks using generic programming; see §2.2.1.

The Framework maintains planar arrangements, computes sample points, executes the efficient construction of (multi-surface) z-fibers (with filters), and is responsible to store the adjacency relation of multi-surface z-fibers. In other words, it implements the surface-independent tasks from §5.1; see also §5.3.2 and §5.3.3. In order to do so, it defines a *concept* that expects basic geometric types (such as the one for a surface) and basic operations on them; the concept is described in §5.3.1.

Additionally, the framework provides classes that rewrite or use the obtained combinatorial results to enable other geometric algorithms on surfaces; main examples are given in §5.5.

Models provide concrete implementations for the concept. That is, the model for a certain family of surfaces implements the surface-specific sets of tasks listed in §5.1; such as to provide surface related projected curves, to create single-surface z-fibers, to detect the equality of their entries, or to obtain the single-surface adjacency.

This part of the thesis concentrates on the framework. In §5.4 we present two models for algebraic surfaces. The framework implement in an experimental package of CGAL, that is planned for future submission to the project's editorial board. All framework-specific code consists of about 10,000 lines of templated C++. This number is not counting required basic classes of the libraries or CGAL's matured `Arrangement_2` package, on which the

framework's implementation is mainly based. In particular, we rely on the `Arrangement_2`'s capabilities to extend DCEL-cells with data and to overlay (such) arrangements; see §2.4.3 and remember that in order to support a certain family of curves, a proper model of *ArrangementTraits_2* must be provided. This and other requirements are listed next, when we present the *SurfaceTraits_3* concept.

5.3.1. The *SurfaceTraits_3* concept

The *SurfaceTraits_3* concept specifies geometric types operations on them to finally support the computation of Problem 5.24. As each concept, it is a collection of syntactic and semantic premises. No assumptions on how to implement them are stated, as long as the demanded functionality is ensured and supported by the formal parameters. We next introduce the concept in its details and show in §5.3.2 and §5.3.3 how the framework interacts with a model of the concept to reach the goal. As in §4.3 and §4.5, the description of the concept lacks the precision of a reference manual. The reason is that such we can emphasize the simplicity of the tasks. In §5.4 we present concrete models fulfilling the *SurfaceTraits_3* concept.

A model of the *SurfaceTraits_3* is supposed to provide the following types:

- `Surface_3`

  An instance of this type should represent a surface $S_i \in \mathcal{S}$. How an instance is constructed is not specified. It depends on the surface the type represents.

- `Boundary`

  A type to represent lower and upper approximations of coordinates.

- `Kernel_2`

  This type determines the geometric properties of the planar arrangements we are going to construct. As we will rely on CGAL's `Arrangement_2` package for this purpose `Kernel_2` must be a model of CGAL's *ArrangementTraits_2* concept. Thus, it provides types `Curve_2`, `X_monotone_curve_2`, and `Point_2` and the operations on them as presented in §2.4.3. We use it to construct $\mathcal{A}_{\{S_i\}}$ and $\mathcal{A}_{\{S_i,S_j\}}$ and their overlays. It depends on the family of surfaces which model is sufficient. The embedded type `Curve_2` is used to represent the corresponding projected curves, that can be decomposed into zero- and (weakly) x-monotone one-dimensional components with `Make_x_monotone_2`.

  In addition to the *ArrangementTraits_2* concept, we require more specific functionality with respect to the nested types:

  - `Construct_interior_vertex_2`

    An instance of this functor is expected to provide the following operator:

    ```
    Point_2 operator()(
      X_monotone_curve_2 xcv
    )
    ```

    which should return a point in the interior of `xcv`, best with coordinates constructed from type `Boundary`.

– `Lower_boundary_x_2`
(and also: `Upper_boundary_x_2`, `Lower_boundary_y_2`, and `Upper_boundary_y_2`)
An instance of this functor is expected to provide the following operator:

```
Boundary operator()(
  Point_2 pt
)
```

which should return a lower approximation of the `pt`'s x-coordinate as instance of type `Boundary`; similar for the upper approximation of x, and analog also for the point's y-coordinate. Each approximation must be unequal to the actual coordinate.

– `Refine_x_2` (and also: `Refine_y_2`)
An instance of this functor is expected to provide the following operator:

```
void operator()(
  Point_2 pt
)
```

whose purpose is to refine the interval defined by lower and upper approximations of `pt`'s x-coordinate (y-coordinate, respectively).

- `Z_at_xy_isolator`

An instance of this type computes, represents, and approximates the set $Z_{p,i} \setminus \{\pm\infty\}$ for a given surface S_i at a given `Point_2` p as refineable intervals. Similar for $Z^{|}_{p,i,j}$ and two surfaces. Its member `size()` gives their number, that is, encodes $m_{p,i}$. The values $z = \{\pm\infty\}$ are implicitly handled.

Refineable means that the z-coordinate with index $0 \leq$ `l` $<$ `size()` might not be known exactly, but at least a lower and an upper boundary is accessible by `lower_boundary(int l)` and `upper_boundary(int l)`. This approximation can be improved by `refine_interval(int l)`. The type of such an interval-end is given by `Boundary`.

Besides these types, the concept also demands for functors related to the projection as presented in §5.1:

- `Construct_silhouette_2`

This functor has to provide three `operator()`s that compute different planar curves emanating from a given surface S. The output is returned as `std::pair< Curve_2, unsigned int >` through an `OutputIterator` (`OI`). The `unsigned int` defines the multiplicity of a curve, if possible to compute, else -1 is chosen. For example, bivariate polynomials defining algebraic curves can be factorized by multiplicity; see §2.1.1.

```
OI operator()(
  Surface_3 s, OI oi
)
```

This first operator returns all curves that belong to the projected silhouette of s.

```
OI operator()(
  Surface_3 s, int d, OI oi
)
```

The second computes for given d all curves whose points can decrease the regularity of a planar point with respect to s to d.

```
OI operator()(
  Surface_3 s, int d, int k, OI oi
)
```

This last operator computes for given regularity d and given $0 \leq k < d$ all curves whose points can increase the degradation of a planar point of regularity d to k with respect to the given surface s.

- `Construct_intersection_2`

  This functor is very similar to the previous one. Its output iterator relies on the same value-type, but the signature of the two demanded operators now expects two surfaces S_1 and S_2.

```
OI operator()(
  Surface_3 s1, Surface_3 s2, OI oi
)
```

  This first operator returns all curves belonging to the projected intersection of the two surfaces. Note that we assumed surfaces to intersect at most one-dimensional.

```
OI operator()(
  Surface_3 s1, Surface_3 s2, int d1, int d2, int k, OI oi
)
```

  This second operator returns for given regularities d_1 and d_2 and given $0 \leq k < \min\{d1, d2\}$ all curves whose points can increase the degradation of a planar point with $(d1, d2)$-regularity to k with respect to the two surfaces.

Finally, the concept expects functors supporting the lifting and adjacency phase:

- `Construct_isolator`

  An instance of this functor is expected to provide the following operator(s):

```
Z_at_xy_isolator_2 operator()(
  Point_2 pt, Surface_3 s, Cell_info1 ci
)
```

  which constructs for given `pt` and S the correct instance of `Z_at_xy_isolator` type, that is, it computes $Z_{\mathtt{pt},i}$ for $S_i = S$; even if S has a vertical line at `pt`. For an illustration see Figure 5.1.

  The given point is included in a cell Γ of $\mathcal{A}_{\{S\}}$. To trigger a special or more efficient implementation, the cell-info `ci` comprises integral values for $\dim(\Gamma)$, d_Γ, k_Γ, and, in case that $\dim(\Gamma) = 1$ (i. e., Γ is an edge), the multiplicity of τ_S's factor that contains `pt`. In addition, it carries boolean values indicating whether S consists of a two-dimensional vertical component, or whether S is vertical locally at `pt`.

```
Z_at_xy_isolator_2 operator()(
  Point_2 pt, Surface_3 s1, Surface_3 s2,
  Cell_info1 ci1, Cell_info1 ci2, Cell_info2 ci12
)
```

This operator is expected to compute $Z^{|}_{p,i,j}$. As precondition we have that S_1 or S_2 has a vertical line at `pt`. Similar to the previous operator the provided `Cell_info1` containers give information for the cell of $\mathcal{A}_{\{S_1\}}$ and $\mathcal{A}_{\{S_2\}}$ that contain `pt`. The `Cell_info2` container collects information on $\mathcal{A}_{\{S_1,S_2\}}$'s cell $\Gamma_{1,2}$ containing `pt`: $\dim(\Gamma_{1,2})$, $k_{\Gamma_{1,2},1,2}$, and, in case that $\dim(\Gamma_{1,2}) = 1$ (i. e., $\Gamma_{1,2}$ is an edge), the multiplicity of τ_{0,S_1,S_2}'s factor that contains `pt`.

- `Adjacency`

  An instance of this functor is expected to compute for a surface `s` the adjacency relation of two incident cells of $\mathcal{A}_{\{S\}}$ as depicted in Problem 5.12 by the following operator:

```
OI operator()(
  Surface_3 s,
  Point_2 pt1, Cell_info1 ci1, Z_at_xy_isolator iso1,
  Point_2 pt2, Cell_info1 ci2, Z_at_xy_isolator iso2,
  OI oi
)
```

  The cells are interfaced in terms of their sample points `pt1` and `pt2`. As before, the `Cell_info` container collect information on these cells Γ_1 and Γ_2 of $\mathcal{A}_{\{S\}}$. The interfaced instances of `Z_at_xy_isolator` represent $Z_{\texttt{pt1}}$ and $Z_{\texttt{pt2}}$. The value-type of the `OutputIterator` (`OI`) is `std::pair< int, int >`, reflecting an entry L as defined in Problem 5.12; see Figure 5.2 for an illustration. For $p \in \mathcal{V}$, it also can be the case that `iso1` is identical to $\mathcal{Z}_{p,\mathbb{S}}$. But note that $\mathcal{Z}_{p,\mathbb{S}} \supset Z_{p,i}$, that is, we only consider a finer decomposition of ℓ_p.

- `Equal_z`

  An instance of this functor has to provide the following operator:

```
bool operator()(
  Surface_3 s, Point_2 pt
  Z_at_xy_isolator iso1, int l1, Cell_info1 ci1,
  Z_at_xy_isolator iso2, int l2, Cell_info1 ci2,
  Cell_info2 ci12,
)
```

  It checks whether the `l1`'s entry of `iso1` is supposed to be equal to `l2`'s entry of `iso2`. Both isolators belong to `pt`. Remember that they are only required to store refineable approximations of the entries. Even in simple cases this information is insufficient, as their equality cannot be finally deduced from iterated refinements of the isolating intervals. If the isolators have access to an exact representation the detection of equality can just be forwarded. However, in general, we do not expect this case. Thus, having this functor keeps the chance that the equality decision is achieved less directly; for example, using information provided by the interfaced cell-info containers. In addition, such information may even improve `Equal_z`'s performance by filters. Mind that `Equal_z` usually implements costly computations, for example, unavoidable symbolic evaluations in some cases of algebraic surfaces; see §5.4.2.

  However, the functor has not to deal with all cases. Before it is triggered, we apply a set of filters; see Algorithm 5.5 for details. In particular, we know, when called, that

all intervals of the two given isolators are already refined such, that each interval overlaps with at most one interval of the other isolator. Thus, the set of overlaps forms a candidate list of real intersections. It is the functors tasks to decide for the queried (still undecided) candidate, whether there is really an intersection or whether the isolating intervals will separate after a finite number of further refinements. An example is given in Figure 5.3.

This concludes the discussion of the *SurfaceTraits_3* concept. It is our goal for the future to further abstract the implementation from algebraic components. Finally, it is strongly encouraged to deploy an extensive caching strategy when implementing these functors to avoid unnecessary re-computations of usually costly tasks.

5.3.2. Planar arrangements and attached data

The central class of our framework is called `Projection_2`. It is a reference-counted version [Ket06] of CGAL's `Arrangement_2`. We instantiate with `Kernel_2` as geometric-traits class, and the topology-traits for the unbounded plane, provided by CGAL 3.4. That is, there is a special fictitious rectangle at infinity (as in Figure 4.9 (a)) to distinguish several several unbounded faces.

We enhance the arrangement's DCEL by using CGAL's `Arr_extended_dcel` to attach an internal data class `P_dcel_data` to each vertex, each edge, and each face. An instance of type `P_dcel_data` for a cell Γ comprises the following data:

- the id of the `Projected_2` instance it belongs to
- an enumeration reflecting $\dim(\Gamma)$
- a `CGAL::Object` that encapsulate a handle to access Γ
- a list of surfaces whose projected silhouette or projected intersections are involved in Γ
- a list of surfaces with a two-dimensional vertical component over Γ
- a list of surfaces that have a vertical line over Γ (only if Γ is a vertex)
- a map that assigns a surface S whose projected silhouette participates in Γ to a `Cell_info1` container. The container collects information such as the cell's regularity, degradation with respect to S or, if Γ is an edge, the multiplicity of the factor of the projected silhouette that supports Γ. It also stores a DCEL-handle to the cell of $\mathcal{A}_{\{S_i\}}$ from which Γ might originate (after an overlay)
- a map that assigns pairs of surfaces S_i, S_j whose projected intersection participates in Γ to a `Cell_info2` container. This container collects information such as $k_{\Gamma,i,j}$, a DCEL-handle to the cell of $\mathcal{A}_{\tau_{0,S_i,S_j}}$ from which Γ might originate (after an overlay), or, if Γ is an edge, the multiplicity of τ_{0,S_i,S_j}'s factor that supports the edge
- an instance of type `Point_2`, that is, a sample point in Γ's relative interior
- an instance of type `Z_fiber` (see §5.3.3 for details on this type)

The stored list of data helps in two directions: First, it provides the data expected by the functors required by the *SurfaceTraits_3* concept. Examples are `Construct_isolator` or `Equal_z`. They can benefit from this data for good reasons: The global computation of regularities and degradations for all cells of an arrangement saves repeated local computations within the functors. In addition, the best algorithm according to the given data can be triggered directly in a functor. Second, the list constitutes combinatorial information that enable to filter tasks; for examples see §5.3.3 or §5.5.

Public members of `Projection_2` also provide access to the stored information for potential users, such as the applications we present in §5.5. We exemplary mention `.has_silhouette(Dcel_handle h)` and `.has_intersection(Dcel_handle h)`, where the template `Dcel_handle` corresponds to either a vertex-, an edge-, or a face-handle. In addition, `Projection_2` forwards iterators to traverse all vertices, edges, and faces. Unfortunately, CGAL's `Arrangement_2` forces us to split curves into x-monotone pieces. Thus, these traversal do not reflect if incident cell share the same attached data. For that reason, we provide special traversals that reflect this property. Consider a single surface: We are able to combine vertices and edges to maximal (d,k)-constant paths, that is, a vertex is filtered out if its degree is 2 and the vertex and its two incident edges all share the same (d,k)-values.

Having this enhanced arrangement we are now able to tackle Problem 5.23 and the projection step of Problem 5.24 in terms of software. The framework provides the functor `Construct_projection_2` that includes exactly three operators. Each is either constructing a new arrangement or overlaying existing ones. We present implementation details, while common subtasks are postponed.

- ```
  Projection_2 operator()(
    Surface_3 s
  )
  ```

  constructs $\mathcal{A}_{\{\mathtt{s}\}}$ for given `s`. It implements Algorithm 5.2: First construct $\mathcal{A}_{\tau_{\mathtt{s}}}$, set (d,k)-values for faces, and refine edges respect to other arrangements $\mathcal{A}_{\alpha_{\mathtt{s},d}}$ and $\mathcal{A}_{\sigma_{\mathtt{s},d,k}}$ by overlays. We introduce for each (refined) cell a map-entry from `s` to a new cell-info container and update its information (regularity, degradation) accordingly. Of course, we tuned the implementation not to run all iterations, but to stop as soon as all cells know their d-k-values. This saves the costly construction of new arrangements (and curve-analyses) and overlays with the existing ones. Note that all required curves are provided by the *SurfaceTraits_3*'s functor `Construct_silhouette_2`. We finally fill missing fields in each cell's `P_dcel_data` container: id of computed `Projection_2`, handle to cell it belongs to, list of involved surfaces (just add `s`).

- ```
  Projection_2 operator()(
    Surface_3 s1, Surface_3 s2
  )
  ```

  constructs $\mathcal{A}_{\{\mathtt{s1},\mathtt{s2}\}}$ for given $\mathtt{s1} \neq \mathtt{s2}$. It implements Algorithm 5.3: First we overlay $\mathcal{A}_{\mathtt{s1}}$ and $\mathcal{A}_{\mathtt{s2}}$. Then, we construct $\mathcal{A}_{\tau_{0,\mathtt{s1},\mathtt{s2}}}$ and overlay it with the previous overlay. Finally, refinements of edges with respect to $\mathcal{A}_{\tau_{k},\mathtt{s1},\mathtt{s2}}$ to set the k-values are performed. Similar, we introduce for each cell a map-entry from the pair `s1,s2` to a new cell-info container and update its information (degradation) accordingly. Again, the implementation stops further refinements, as soon as all k-values are known. Note that all required curves are provided by the *SurfaceTraits_3*'s functor `Construct_intersection_2`. Again, missing fields in each cell's `P_dcel_data` container are set at the end: id of computed `Projection_2`, handle to cell it belongs to, list of involved surfaces (add `s1` and `s2`).

- ```
  template < class InputIterator >
  Projection_2 operator()(
    InputIterator begin, InputIterator end
  )
  ```
  ```

constructs $\mathcal{A}_{\mathcal{S}}$, where $\mathcal{S}$ is attained by the input range [`begin`,`end`). The operator implements an overlay of all pairs $\mathcal{A}_{\{S_i,S_j\}}$, $i \neq j$. This is feasible by CGAL's arrangements. Concerning the attached data, note that the cell-info container for a point with respect to a given surface (or a pair of surfaces) must be equal, even if stored in different arrangements. As overlaying such arrangements only refines cells (with attached data), it suffices to merge the originating key-value-pairs of proper maps. The same holds for the list of involved surfaces and list of surfaces with vertical components. At any point, no deletion of an entry in a list or map is required. Finally, we again assign the id of the resulting arrangement and a cell handle to each cell. As each $\mathcal{A}_{\{S_i\}}$ appears up to n times, we remark that there is room for further improvements, using a more direct overlay.

Remark. The functor exploits an internal caching strategy to avoid repeated constructions. This means that for a given surface S_i, there will be exactly one `Projected_2` instance that represents $\mathcal{A}_{\{S_i\}}$, and for each pair $S_i, S_j, i \neq j$, there will be exactly one `Projected_2` instance that represents $\mathcal{A}_{\{S_i,S_j\}}$. Each such instance has a unique id in memory. The functor, again, is responsible to correctly assign this id to each resulting DCEL-cell (for later look-ups).

As promised, some remarks on subtasks:

- A first subtask is to compute an arrangement for a set of planar curves. Remember that each curve reported by a projection-functor of the *SurfaceTraits_3* concept, is enhanced with a multiplicity. In this substep we split each curve into its isolated points and (weakly) x-monotone curves, compute the induced arrangement, and assign the corresponding multiplicity to each edge. Finally, these arrangements are overlaid, while propagating the multiplicity information for edges.

  This substep is used when computing $\mathcal{A}_{\tau_{S_i}}$ from curves reported by the simplest operator of `Construct_silhouette_2` and $\mathcal{A}_{\tau_0,S_i,S_j}$ from curves reported by the simplest operator of `Construct_intersection_2`. We already remark that $\mathcal{A}_{\tau_S}$ is central in an application that we present later in §5.5.3 on page 248 ff.

  As `Kernel_2` is a model of CGAL's *ArrangementTraits_2* concept, the constructions and overlays of arrangements can be handled by CGAL's `Arrangement_2` package; see also §2.4.3.

- Although the refinements in Algorithm 5.2 and Algorithm 5.3 involve different values, they share common abstract steps:
  - compute an overlay of two arrangements
  - detect the cells whose values gets set
  - compute the value from the information available in the current iteration
  - remove unnecessary cells

  Our implementation exactly follows these generic steps, while code specializes for the refinement of an arrangement with respect to multiplicities, regularities, and degradations. Actually, the ultimate goal is to abstract further and to iteratively compute the *property* (such as regularity or degradation) for each cell in a sequence of overlays: Each overlay step adds a new *attribute* value (here, the existence of a curve), while after each overlay, it is checked whether the property can already be

computed from the the available attributes. However, this generalization is beyond the scope of this thesis.

For the lifting of surfaces, a sample point for each cell is required. As a vertex of an arrangement is zero-dimensional, there is no choice. The sample point of a vertex is simply the attached `Point_2`. An edge is one-dimensional, so there is some choice. Note that each edge stores an `X_monotone_curve_2`. A point in its interior can be computed by `Kernel_2::Construct_interior_vertex_2`, even with a x- or y-coordinate of type `Boundary`. To compute the sample point of face, remember that we can access an approximation of a point that represents a rectangle. Thus, we choose a point p on a CCB of a face. Let B be the rectangle defined by p's approximation. Pick a point p' on a part of B that is intersected by the desired face. In case, the boundary of the rectangle does not intersect with the face, we refine the point's approximation until its boundary has an intersection with the face. Note that following this strategy, the complexity of sample points for edges and faces depends on the provided planar kernel. We actually try to compute such with rational coordinates of low bit-size, if possible.

Consider now a cell that originates from the overlay of two arrangements. We can simply compute a new sample point for this cell. However, as the sample point is also the base of the lifting, which we explain next, we do not want to have too many different sample points. Thus, it is first checked, whether one of the sample points of the originating cells fits for the resulting cell. If so, this one is chosen.

5.3.3. `Z_fiber`

Once the planar arrangements enhanced with combinatorial data and sample points for each cell are computed, we can lift them to the third dimension in order to achieve a cell decomposition; see §5.1. Concerning the implementation we have to represent a z-pattern for each cell along with geometric information on the surfaces' z-coordinates. Thus, we present the class `Z_fiber` that serves both goals.

In what follows we fix a single cell $\Gamma \in \mathcal{A}_{\mathbb{S}}$, where the case $|\mathbb{S}| = 1$ is special and requires only trivial processing. Let $p \in \Gamma$. For our purpose, we typically have $p = p_\Gamma$ where p_Γ is the sample point of Γ. However, if desired, any point is selectable; we only detect changes in the surfaces' z-coordinates, when moving p within Γ. So, assume $p = p_\Gamma$.

Let $\mathbb{S}_\Gamma = S_{\Gamma,1}, \ldots, S_{\Gamma,r}$ be the set of surfaces involved in Γ. We know this information. In particular, by available combinatorial information, we can even partition $\mathbb{S}_\Gamma$ into $\mathbb{S}_\Gamma^{|} \uplus \mathbb{S}_\Gamma^{*}$ such that for $S \in \mathbb{S}_\Gamma^{|}$ we have $\ell_p \subset S$ and for $S \in \mathbb{S}_\Gamma^{*}$ we have $\ell_p \not\subset S$. Thus, an instance of type `Z_fiber` maintains a list for surfaces being vertical over p. We are missing to achieve geometric information for $S \in \mathbb{S}_\Gamma^{*}$. Thus, for each such S we call `Construct_isolator` interfacing the available cell-info as expected, which returns a `Z_at_xy_isolator` instance providing the desired (approximative) z-coordinates for S at p. The `Z_fiber` maintains a map that assigns S to its respective isolator. This completes the part of an `Z_fiber` instance dealing with geometric information.

We next turn to compute the sequence $W_{p,\mathbb{S}_\Gamma^{*}} = w_{p,1}, \ldots, w_{p,k}$ representing (together with $\mathbb{S}_\Gamma^{|} =: w_p^{|}$) the z-pattern over Γ. The `Z_fiber` class maintains a sorted list of surface-sets. Each set is called a `Z_cell` and stores instances of `std::pair< Surface_3, int >`. Such a pair denotes a *surface lift* over Γ. Note that the `int` corresponds to the sheet

number of the `Surface_3` instance at p; see also Definition 2.29. Observe that the `Z_fiber` decouples the combinatorial z-pattern from the geometric information (i. e., $Z_{p,i}$). But as a `Z_cell` can store a lift for each $S \in \mathcal{S}_\Gamma$, we are able to reassemble them: It is easy to refine the intervals of the stored isolators such that all intervals belonging to one surface lift are isolating with respect to the intervals belonging to surface lifts of the neighbored (below/above) `Z_cell`, if existing. That is, their convex hulls are isolating to each other. We remark, that cells for $z = \pm\infty$ are not explicitly stored.

Theoretically, Equation (5.1) defines $w_{p,l}$ and thus the entries of a `Z_cell`. In practice we still have to determine each. In case that $|\mathcal{S}_\Gamma| = 1$, this task is obvious. Computing $W_{p,\mathcal{S}^*_\Gamma}$ with $|\mathcal{S}_\Gamma| > 1$ is implemented via a multi-way merging. That is, for a set of z-coordinates $Z := \{z_{p,i,l_i} \mid 1 \leq i \leq r\}$ we have to compute $Z_{\min} := \{i \mid z_{p,i,l_i} = \min(Z)\}$. This requires to compare the z-coordinates as stated in Problem 5.22. The isolators stored for the surfaces do not provide sufficient information to determine $Z_{\min}$, actually to determine if $|Z_{\min}| > 1$. The reason is, that an isolator only provides refineable approximation for all z_{p,i,l_i}. At this point, the *SurfaceTraits_3*'s functor `Equal_z` enters the stage. The subsequent discussion assumes that $|\mathcal{S}_\Gamma| = 2$; the extension to $|\mathcal{S}_\Gamma| > 2$ is straightforward.

In order to enable a two-way merge, our task is to compute the order of $z_{p,1,l_1} \in Z_{p,1}$ and $z_{p,2,l_2} \in Z_{p,2}$ for surfaces S_1 and S_2. The direct solution is given by Algorithm 5.4.

Algorithm 5.4. Compare entries of z-fibers of two surfaces

INPUT: $z_{p,1,l_1} \in Z_{p,1}$, $z_{p,2,l_2} \in Z_{p,2}$
OUTPUT: Their order

1. Refine intervals of isolators representing $Z_{p,1}$ and $Z_{p,2}$ such that each interval overlaps with at most one interval of the other isolator.
2. The overlapping intervals form a candidate list for possible intersections of S_1 and S_2 along ℓ_p. If no candidate is found, proceed with (5).
3. Check if the intervals approximating $z_{p,1,l_1}$ and $z_{p,2,l_2}$ overlap. This can be done in terms of indices l_1 and l_2. If not, proceed with 5.
4. Call `Equal_z` for $z_{p,1,l_1}$ and $z_{p,2,l_2}$. If it returns `true`, return `EQUAL`.
5. Reaching here indicates that $z_{p,1,l_1}$ and $z_{p,2,l_2}$ are not equal, that is, their approximative intervals can be refined until they do not overlap any more, which gives the correct order, that is, `SMALLER` or `GREATER`.

Algorithm 5.4 fully relies on the *SurfaceTraits_3*'s functor `Equal_z` to decide the equality. However, this strategy ignores available combinatorial information attached to Γ and continuations we expect from surfaces. Thus, we present Algorithm 5.5 that exploits these data in order to avoid, usually costly, calls to `Equal_z`. One of the filters (highlighted) detects that $z_{p,1,l_1} = z_{p,2,l_2}$, while most of them decide that $z_{p,1,l_1} \neq z_{p,2,l_2}$.

For reasons of efficiency, the filters are active by default. When we discuss algebraic surfaces in §5.4, they help to avoid costly equality test, for example, at points with high algebraic degrees. Note that the equality over vertices is only explicitly checked, if there exists an isolated point (a degenerate case). However, the coordinates of vertices are usually the ones with the highest algebraic degrees. Thus, it is beneficial to filter such cases with combinatorics. For the future, we hope to develop further filters.

We want to remark, that the lifting follows the lazy evaluation scheme. This means that sample points for DCEL-components and their z-fibers are only computed on demand. Further requests for them are served by cached versions. Of course, `Projection_2`

Algorithm 5.5. Compare entries of z-fibers of two surfaces, with filters

INPUT: $z_{p,1,l_1} \in Z_{p,1}$, $z_{p,2,l_2} \in Z_{p,2}$
OUTPUT: Their order

1. If $\dim(\Gamma) = 2$ (face), proceed with (10).
2. If τ_{0,S_1,S_2} is not involved in Γ, proceed with (10).
3. Refine intervals of isolators representing $Z_{p,1}$ and $Z_{p,2}$ such that each interval overlaps with at most one interval of the other isolator.
4. The overlapping intervals form a candidate list for possible intersections of S_1 and S_2 along ℓ_p. If no candidate is found, proceed with 10.
5. Check if the intervals approximating $z_{p,1,l_1}$ and $z_{p,2,l_2}$ overlap. This can be done in terms of indices l_1 and l_2. If not, proceed with (10).
6. If there is exactly one overlap, check if $\dim(\Gamma) = 1$ (edge) and if it stores multiplicity 1 for τ_{0,S_1,S_2}. If so return `EQUAL`, if not, proceed with (10).
7. If $\dim(\Gamma) = 0$ (vertex), select incident edges of $\Gamma \in \mathcal{A}_{\{S_1,S_2\}}$ whose `Z_fiber` indicate an intersection of S_1 and S_2. Compute for each `Z_cell` containing an intersection the adjacencies of S_1 and S_2 towards given vertex (using *SurfaceTraits_3* `Adjacency` functor). For each we obtain a pair of indices. If one pair matches (l_1, l_2), return `EQUAL`, which follows by Condition 5.9. Otherwise proceed with (10).
8. If $\dim(\Gamma) = 0$ (vertex), check which τ_{S_1} and τ_{S_2} are involved in Γ. If none, proceed with (10), as only isolated points remain for possible intersections, but an isolated point is indicated by the existence of a projected silhouette.
9. Finally, call `Equal_z` for $z_{p,1,l_1}$ and $z_{p,2,l_2}$. If it returns `true`, return `EQUAL`.
10. Reaching here indicates that $z_{p,1,l_1}$ and $z_{p,2,l_2}$ are not equal, that is, their approximative intervals can be refined until they do not overlap any more, which gives the correct order, that is, `SMALLER` or `GREATER`.

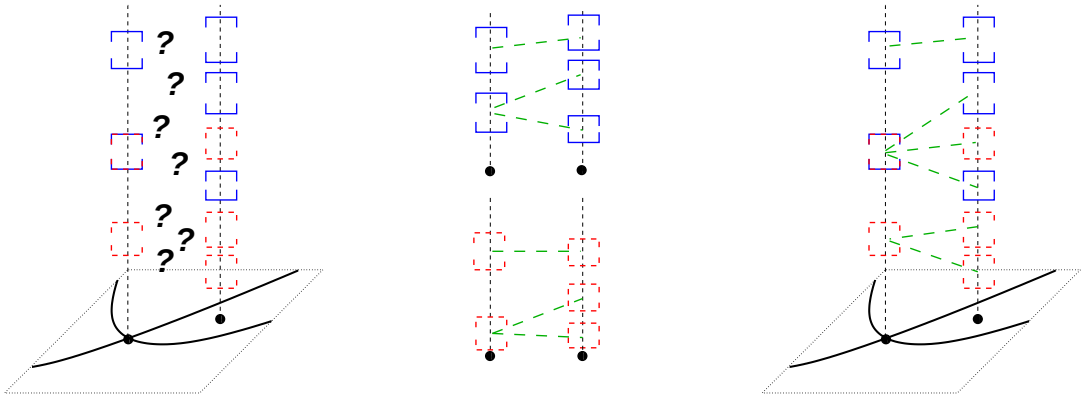
offers public members to access sample points (`.sample_point(Dcel_handle h)`) and z-fibers (`.z_fiber(Dcel_handle h)`) for given DCEL-handles. When merging attached data due to an overlay, we already mentioned that our code always tries to reuse already computed sample points. Obviously, the same idea is possible for `Z_fiber` instances attached to a cell, especially for the stored isolators.

In general, we have seen how to efficiently construct the z-pattern for a cell Γ, that also implicitly defines the multi-surface z-fiber of surfaces involved in Γ. The computations highly benefit from precomputed combinatorial data attached to Γ.

Remark. In case that $p \in \mathcal{V}$, Algorithm 5.4 is also used to compute $\mathcal{Z}_{p,\mathbb{S}}$ by merging the entries of $Z_{p,i}$, $Z_{p,j}$ and $Z^{|}_{p,i,j}$.

As last step, it remains to connect `Z_cell` instances with respect to the adjacency relation(s). For that reason, each such cell maintains a list storing handles to adjacent cells. If $\mathbb{S}$ consists of a single surface, the lists can directly be filled with information provided by querying *SurfaceTraits_3*'s `Adjacency` functor for all pairs of incident cells of $\mathcal{A}_{\mathbb{S}}$. In principle, the same idea is applicable if $|\mathbb{S}| > 1$. The difference is now that the indices of the z-pattern $w_{\Gamma,l}$ are not identical to the surface lifts. To correctly maintain the lists of adjacent `Z_cell` instances, we have to locate w_{Γ_1,l_1} and w_{Γ_2,l_2} that contains the reported index-pairs L of z-fibers to link them; see also Equation 5.1, Problem 5.12 and, for an illustration, Figure 5.7.

Figure 5.7. Propagate single-surface adjacencies to multi-surface fibers



(a) What is the adjacency relation between cells of multi-surface fibers?

(b) Compute single-surface adjacencies and ...

(c) ... propagate them using sheet numbers.

Remark. Note that this propagation only works if none of the cells Γ_1, Γ_2 contains a vertical ℓ_p of some surface. Otherwise, we have by Condition 5.17 that more than one surface influences the decomposition of ℓ_p. Let us assume that $\Gamma_1 = \{p\}$ with $p \in \mathcal{V}$. In this case $Z_{p,i}$ must be replaced by $\mathcal{Z}_{p,\mathbb{S}}$ and we compute the adjacency relation of each S_i between its lifts over Γ_1 and all lifts of S_i over Γ_2. Note that we only have to match correct indices for lifts of Γ_2, while the indices for Γ_1 are already reported with respect to $\mathcal{Z}_{p,\mathbb{S}}$.

5.4. Models for algebraic surfaces

In this section, we finally present details on two models that we provide for the new *SurfaceTraits_3* concept. Both deal with algebraic surfaces.

`Quadric_3_traits` Supports algebraic surfaces of degree at most 2. It was our initial model, and allows combinatorial filters for the functors related to lifting.

`Algebraic_surface_3_traits` This model supports algebraic surfaces of any degree.

The models have in common, that none expects to shear the three-dimensional coordinate system in order to avoid degeneracies. In the recent implementation, the quadrical model is a refinement of the other. But let us present the details step by step. Algebraic surfaces have already been touched in §5.2. Thus, we mainly concentrate on implementation details. We start with the basic types and switch in §5.4.1 to the projection tasks. Then, §5.4.2 covers the details on the lifting phase. There, we also give the missing computation of $Z_{p,i}$ and $Z^{|}_{p,i,j}$ for algebraic surface S_i and S_j.

Algebraic surfaces are represented by the class template `Algebraic_surface_3`. It is based on CGAL's `Polynomial` class, but adds surface-specific functions. An object of this type is constructed from a trivariate polynomial. We typedef `Surface_3` to this type.

For our quadrical model, EXACUS' class `Quadric_3` derives from the `Algebraic_surface_3` and adds constructors (e. g., from ten coefficients defining the quadric) and other specific members: for example, to compute the quadric's matrix representation, or the quadric's inertia (not required here). Both classes are templated by an `Arithmetic_kernel` that provides coherent types for integer, rational, and bigfloat numbers; see §2.3.1. We typedef `Boundary` to `Arithmetic_kernel::Rational`.[50]

5.4.1. Projections for algebraic surfaces

As seen in §5.2.2, the projection for algebraic surfaces requires to construct and overlay arrangements of algebraic curves. Their degree is bounded by D^2, where D is the maximum (total) degree of any input surfaces. Thus, for quadrics we need a model of `Kernel_2` that can deal with algebraic curves of degree at most 4, while the any-degree model, requires a model that supports algebraic curves without restrictions on their degree. Such a model has become available recently with CGAL's `Curved_kernel_via_analysis_2` if instantiated with CGAL's bivariate `Algebraic_curve_kernel_2`; see §2.4.4, [BE08], and §2.3.3, [EK08a], [EKW07] for more details. In fact, the `Curved_kernel_via_analysis_2` also provides the additional functors (interior vertex, approximations for points) as listed in §5.3.1. Similar to the geometric predicates and constructions expected by CGAL's *ArrangementTraits_2* concept, they are implemented relying on the provided algebraic kernel. Thus, we are able to finally typedef `Kernel_2` to `CKvA_2< Algebraic_curve_kernel_2>`. Note that we do not specialize for quadrics.

The same holds for the functors related to the projection, that is, they serve both models. For simplicity, we abuse notation and identify surface and defining polynomial. Remember that $f = \sum_{d=0}^{D} a_d z^d$ and for $D_0 < D$: $f_{(D_0)} := \sum_{d=0}^{D_0} a_d z^d$. We require to decompose the polynomials $\mathrm{Res}_z(f, \frac{\partial f}{\partial z})$, a_d, and $\mathrm{stha}_k(f_{(d)})$ into square-free factors

[50] In a future version, it is an objective to use `Arithmetic_kernel::Bigfloat` as `Boundary` type.

and construct corresponding curve instances. We utilize CGAL's `Polynomial` [Hem07c] and `Algebraic_kernel_d` [BHKT08] package, that provide all required operations, such as square-free factorization, resultants and their sequences. It allows to provide a straightforward implementation of `Construct_silhouette_2` and `Construct_intersection_2`. The value-type of the functor's `OutputIterator` is `std::pair< Curve_2, int >`, where `Curve_2` is actually a `Curve_analysis_2` provided by the `Algebraic_curve_kernel_2`. The reported `int` represents the corresponding multiplicity of the square-free factor.

The functor `Construct_silhouette_2` has to provide three operators for a given surface:

`OI operator()(Surface_3 f, OI oi)`

we compute and report the square-free factorization of $\mathrm{Res}_z(f, f_z)$,

`OI operator()(Surface_3 f, int d, OI oi)`

we compute and report the square-free factorization of a_d with $0 \leq d \leq D$

`OI operator()(Surface_3 f, int d, int k, OI oi)`

we compute and report the square-free factorization of $\mathrm{stha}_k(f_{(d)}, \frac{\partial}{\partial z} f_{(d)}, z)$ with $0 \leq k < d \leq D$.

For the `Construct_intersect_2` functor, exactly the same approach is taken, with the difference that the desired polynomials are expressed with respect to two given surfaces.

`OI operator()(Surface_3 f1, Surface_3 f2, OI oi)`

we compute and report the square-free factorization of $\mathrm{Res}_z(f_1, f_2)$

`OI operator()(Surface_3 f1, Surface_3 f2, int d1, int d2, int k, OI oi)`

we compute and report the square-free factorization of $\mathrm{sres}_k((f_1)_{(d_1)}, (f_2)_{(d_2)}, z)$, with $0 \leq k < \min(d_1, d_2) \leq D$.

Remark. We compute Sturm-Habicht sequences with cofactors as given by [BPR06, Algorithm 8.22]. This algorithm relies on polynomial remainder sequences [Loo82b]. In practical setting this is more efficient than computing the Sturm-Habicht sequence via determinantal expressions.

Note that the actual construction of the desired arrangements is implemented using exactly the output of these functors; see §5.3.2 and Algorithms 5.2 and 5.3.

5.4.2. Lifting for algebraic surfaces

In the lifting phase, we have three tasks to achieve. Namely, to construct isolators representing $Z_{p,i}$ and $Z^{|}_{p,i,j}$, to decide equality for two entries of such isolators for different surfaces, and to compute the adjacency relation between the entries of two isolators belonging to the same surface. We first discuss these tasks for algebraic surfaces of any degree, and finally present how to combinatorially filter the quadrical case.

Isolator

For all constructions of $Z_{p,i}$ we rely on the bitstream Descartes method that has been presented with its details in §2.3.4. Remember that the method isolates the real roots of a polynomial whose coefficients are given as possible infinite bitstreams, that is, the approximation of its coefficients can be improved to arbitrary precision. Thus, we typedef `Z_at_xy_isolator` to CGAL's type `Bitstream_descartes`.

For our purposes, we require a new model fulfilling the *BitstreamDescartesRndlTreeTraits* concept, which we call the `Bitstream_z_at_xy_traits`. There are three constructors for this traits:

square-free-construction

```
Bitstream_z_at_xy_traits(
  Polynomial_3 f, Point_2 pt
)
```

which supports to isolate the roots of $f(\mathtt{pt}) := f(p_x, p_y, z) = \sum_{d=0}^{D} a_d(\mathtt{pt}) z^d \in \mathbb{R}[z]$ with the bitstream Descartes method. Remember that $a_k \in \mathbb{Q}[x, y]$. The constructor requires that $f(\mathtt{pt})$ is square-free.

m-k-construction

```
Bitstream_z_at_xy_traits(
  Polynomial_3 f, Point_2 pt,
  int m, int k
)
```

which supports to isolate the roots of $f(\mathtt{pt}) := f(p_x, p_y, z) = \sum_{d=0}^{D} a_d(\mathtt{pt}) z^d \in \mathbb{R}[z]$ with the m-k-bitstream method, where `m` represent the local real degree of $f(\mathtt{pt})$ and `k` the local gcd degree of $f(\mathtt{pt})$. It is successful, if $f(\mathtt{pt})$ has at most one multiple root, otherwise an exception is thrown; see also §2.1.2.

vertical-line-construction

```
template < class InputIterator >
Bitstream_z_at_xy_traits(
  InputIterator begin, InputIterator end
)
```

which supports a simulated isolation. It only forwards the input range `[begin,end)` of handles to already isolated intervals, that is, to entries of isolators constructed with the square-free or m-k-variant. We use it to represent the isolator for $Z_{p,i}$ for $p \in V_i$, or for $Z^{|}_{p,i,j}$. Below, we see that such sequences consists of links to roots of a small number of polynomials.

The first two constructors rely on the possibility to refine `pt`'s coordinates to arbitrary precision; see §5.3.1 and §5.4.1. This directly supports the computation of the approximations as `Bigfloat` intervals as expected by the `Bitstream_coefficient_kernel`. Additionally, for $c \in \mathbb{Q}[x, y]$ (as a_d, or stha-coefficients) we can even determine $\mathrm{sign}(c(p_x, p_y))$ using `Algebraic_curve_kernel_2`'s `Sign_at_2` functor. It internally uses a clever combination of analyses of curves and interval arithmetic. Note that this enables the zero-test that is expected to obtain a better initial interval; see `Bitstream_coefficient_kernel` in §2.3.4. Even more, the m-k-variant relies on the functor to compute a sequence of signs; see below.

The different variants (square-free-constructor, m-k-constructor, sequence-constructor) are interfaced through the common `Bitstream_descartes` class; see also §2.3.4. This allows that a user (as, e. g., Algorithm 5.5), is not aware of the various details required in each variant. Its main objectives with respect to some isolator are:

- How many entries does some isolator have?
- Give me an interval approximation of $z_{p,l}$ for given l.
- Refine the interval approximation of $z_{p,l}$ for given l.
- Which l belongs to the multiple root? (Only for the m-k-variant!)

It remains to discuss how `Construct_isolator` combines the different traits constructions in order to correctly provide the desired isolator for $Z_{p,i}$. Note that the interface of the functor receives via a cell-info the local degrees $d_{p,i}$, $k_{p,i}$, $m_{p,i}$ (see §5.2.1), and information on whether $\Gamma \in \mathcal{A}_{\{S_i\}}$ with $p \in \Gamma$ is a vertex, an edge, or a face. In case Γ is an edge, the multiplicity of τ_{S_i}'s factor that supports the edge is also provided.

We first consider the non-vertical case, that is $p \notin V_i$. If $k_{p,i} = 0$, then $f_i(p)$ is square-free; this triggers the standard construction of the `Bitstream_z_at_xy_traits` from f_i and p only. The traits itself ensures iterated and coherent refinements of interval approximations for p_x and p_y to serve the actual isolation; actually it demands for them from the algebraic kernel.

Otherwise, if $k_{p,i} > 0$, we first try to run the m-k-Bitstream Descartes method (see also [EKW07, Section 5]) on $f_i(p)$. This extension exploits our knowledge on the local real degree and the local gcd degree, and isolates the real roots using numerical approximations even if $f_i(p)$ has at most one multiple root. However, we are required to compute m. This can be done, for example, using a modified version of Algorithm 2.3 that can deal with specialized polynomials. For computing the signs of stha_i we rely on the algebraic kernel's functor `Sign_at_2`.

However, it is not ensured, that the m-k-variants exists with success. So we are left with the case, that $f_i(p)$ has more than one multiple root. In this case, we compute the square-free part $f_i^*(p)$ of $f_i(p)$ using Algorithm 2.4 and apply the Bitstream Descartes method on $f = f_i(p)^*$ using the first constructor. As f is square-free, termination is ensured. Observe that in all cases, we simplify by ignoring $\pm\infty$ being part of a z-fiber.

It is essential that the algebraic kernel models the planar points' coordinates in algebraic interval represention; see Definition 2.17. Following, all obtained z-coordinates can be expressed as algebraic interval representations of dimension 3.

However, it is open, and promised in §5.2, to compute the entries of $Z_{p,i}$ if $p \in V_i$. Remember from Definition 5.10 how the entries of $Z_{p,i}$ are characterized, namely as the endpoints of intervals of lifted faces that are incident to p; see also Theorem 5.45 and Corollary 5.52. The computation of a superset $Z_{p,i}^*$ for $Z_{p,i}$ is shown in [BKS08]. As we are focussing on the algorithmic part of the objective, we only review its main ideas and present the central result; for the (lengthy) proofs we refer to the original work.

Let $F \in \mathcal{A}_{\{S_i\}}$ be a face incident to p and let $I(F^{(l,i)})$ be a non-degenerate adjacency interval. Choose an arbitrary interior point $(p, z_0) \in I(F^{(l,i)})$, that is, $z_0 \notin Z_{p,i}$. It is an implication of Theorem 5.45 that the planar curve $C_{z_0} = \{(x, y) \in \mathbb{R}^2 | f_i(x, y, z_0) = 0\}$, embedded into the arrangement $\mathcal{A}_{\{S_i\}}$, contains at least one arc that leaves p and passes face F. Vice versa, each arc of C_{z_0} starting in p corresponds uniquely to a lifted surface patch above F which is adjacent to (p, z_0). This observation is the basis for the computation

of possible interval endpoints by a conceptual sweep along ℓ_p: We keep track of the special arrangement $\mathcal{A}_{\{S_i\},z_0}$ that is induced by the overlay of $\mathcal{A}_{\{S_i\}}$ and $\mathcal{A}_{C_{z_0}}$, while moving with z_0 from $-\infty$ to $+\infty$. It is the objective to detect possible topological changes of $\mathcal{A}_{\{S_i\},z_0}$ local to p. To be more precise: Compute all $z \in \mathbb{R}$ where for any face F in $\mathcal{A}_{\{S_i\}}$ the number of arcs of C_z leaving p and passing F changes. Observe that for most z_0, a slight perturbation of z_0 deforms C_{z_0} in such a way, that the local topology of $\mathcal{A}_{\{S_i\},z_0}$ at p is preserved. Hence, an arc of C_{z_0} contained in F still lifts to the same $F^{(l,i)}$. In contrast, perturbing an z_0 that belongs to an endpoint of an interval $I(F^{(l,i)})$ results in either loosing an arc that passes p or in an arc that switches to another face F'; see Figure 5.8 where loosing arcs happens at $z_0 = \pm\frac{1}{2}$, and switching arcs happen at $z_0 = 0$.

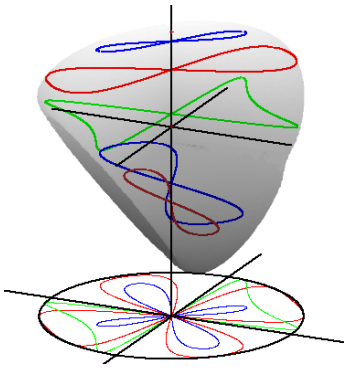


Figure 5.8. Steiner Roman surface with horizontal intersections at $z = \frac{1}{2}, \frac{2}{5}, \frac{3}{10}, \frac{1}{10}, -\frac{3}{10}, \frac{2}{5}$ taken from [BKS08]

The following theorem from [BKS08] algebraically describes the non-generic z-values that respect local topology changes of $\mathcal{A}_{\{S_i\},z_0}$ at $p \in V_i$.

Theorem 5.60. *Let S_i be an algebraic surface without two-dimensional vertical component, defined by $f_i \in \mathbb{Q}[x,y,z]$ being square-free and let $p \in V_i$ as above. Let*

$$\begin{aligned} r(x,z) &:= \operatorname{Res}_y(f_i, \frac{\partial f_i}{\partial y}) = (x-p_x)^{r_0}\tilde{r}(x,z), \\ t(x,z) &:= \operatorname{Res}_y(f_i, \operatorname{Res}_z(f_i, \frac{\partial f_i}{\partial z})) = (x-p_x)^{t_0}\tilde{t}(x,z) \end{aligned}$$

with the following definitions of exponents

$$\begin{aligned} r_0 &:= \max\{r' \,:\, (x-p_x)^{r'} | r(x,z)\}, \\ s_0 &:= \min\{s' \,:\, \tfrac{\partial^{s'}}{\partial y^{s'}} f_i(p_x,p_y,z) \not\equiv 0\} \\ t_0 &:= \max\{t' \,:\, (x-p_x)^{t'} | t(x,z)\}. \end{aligned}$$

Then for $z_0 \notin Z^*_{p,i} := \{z | \tilde{r}(p_x,z) = 0 \vee \frac{\partial^{s_0}}{\partial y^{s_0}} f_i(p_x,p_y,z) = 0 \vee \tilde{t}(p_x,z) = 0\}$ the local topology of $\mathcal{A}_{\{S_i\},z_0}$ at p is preserved for any sufficiently small perturbation of z_0. Additionally, $Z_{p,i} \subset Z^*_{p,i}$.

By assumption C_z is square-free and does not share a common component with τ_S for all but finitely many $z \in \mathbb{R}$. Such *degenerate* z-values are exactly given by $\operatorname{Res}_y(f_i, \frac{\partial f_i}{\partial y})(x,z) \equiv$

0 or $\mathrm{Res}_y(f_i, \mathrm{Res}_z(f_i, \frac{\partial f_i}{\partial z})) \equiv 0$. An implication is that the factorization of $r(x,z)$ and $t(x,z)$ as well as s_0 is well defined. In particular for each $z_0 \in Z_{p,i}$, the curve C_{z_0} is square-free and it neither contains the vertical line $L := V_{\mathbb{R}}(x - p_x) \subset \mathbb{R}^2$ nor any component of τ_{S_i}. Note that $Z^*_{p,i}$ defines a superset of $Z_{p,i}$. For the full proof and more details we refer to [BKS08]. As an implication, we can precisely define the content of S_i's z-fiber for $p \in V_i$.

Definition 5.61 (z-fiber for $p \in V$). Consider the polynomials

$$R(z) := \tilde{r}(p_x, z) \qquad S(z) := \tfrac{\partial^{s_0}}{\partial y^{s_0}} f_i(p_x, p_y, z) \qquad T(z) := \tilde{t}(p_x, z)$$

contained in $\mathbb{R}[z]$. We relax Definition 5.10, and allow also a superset of the interval boundaries as z-fiber. Thus, we now define

$$Z_{p,i} := \{z \in \mathbb{R} \mid R(z) = 0 \vee S(z) = 0 \vee T(z) = 0\}$$

To compute this set, we isolate the real roots of $R(z)$, $S(z)$, and $T(z)$ with the bitstream Descartes method; remember that the polynomial's coefficients are expressed with respect to p's coordinates, that are, in general, algebraic. However, we know, as in the non-vertical case, approximations for them and how to refine them. Actually, we can obtain a list of polynomials $P_1(z), \ldots, P_l(z)$ representing the square-free and coprime counterparts of $R(z)$, $S(z)$, and $T(z)$ using Algorithms 2.4 and 2.8 (pages 29 and 32). This treatment simplifies two steps: First, each polynomial is square-free. Thus, we can directly apply the square-free-variant of the bitstream Descartes method. Second, as no two polynomials P_{l_1}, P_{l_2} with $l_1 \neq l_2$ share a common root, the merge of the obtained sequences of isolating (and refineable) intervals is simple.

The actual implementation reduces the computation of such a $Z_{p,i}$ to curve- and curve-pair analyses. This trick can be seen as keeping x a little bit longer indeterminate. In addition, filters developed for planar curves do now apply also for this task, which are not accessible in the direct approach as presented above. Doing it this way, also helps to remove factors of $(x - p_x)$ from the original polynomials, as the bivariate polynomials defining curves can be decomposed into "vertical lines" and "non-vertical curves"; see §2.1.4. Thus, we ignore the vertical lines, and only process three non-vertical curves at $x = p_x$.

Finally, we just report the merged sequence of obtained isolating intervals as input range to the third constructor of the bitstream traits model.

Remark. It is an open question, whether there is a more strict definition of $Z_{p,i}$, best one that tightly defines the boundaries of all $I(F^{(l,i)})$. The conjecture is: For given $z_0 \in \mathbb{R}$, we have $S(z_0) = 0 \Rightarrow R(z_0) = 0$.

The desired $O(D^5)$ complexity of the cell-decomposition $\Omega_{\{S\}}$ introduced in Definition 5.28 is fulfilled as we now can give the missing proof of Theorem 5.46.

Proof. **(of Theorem 5.46)** Observe that we only have to show that $|Z_{p,i}| \in O(D^3)$ according to Definition 5.61 of $Z_{p,i}$. Consider the polynomials $R(z), S(z), T(z)$ whose roots define $Z_{p,i}$. Each is of degree at most $O(D^3)$. Thus, each can have up to $O(D^3)$ real roots, which implies the desired bound for the union of them. □

Following Condition 5.17, Definition 5.18, and finally desired by Problem 5.19, we also have to compute $Z^{|}_{p,i,j}$ for given $p \in V_i \cup V_j$ for two surfaces S_i, S_j, $i \neq j$. That is, we have

to explain how to implement the second operator expected by `Construct_isolator`. To solve it, a strategy similar to the one that defines $Z_{p,i}$ for $p \in V_i$ can be used. Analogously, we want to extract z-coordinates at which S_i and S_j induces intervals along ℓ_p. As in Theorem 5.60, we use a local projection onto the yz-plane.

Theorem 5.62. *Let S_i, S_j be algebraic surfaces without two-dimensional vertical component, defined by $f_i, f_j \in \mathbb{Q}[x,y,z]$ being square-free, coprime and let $p \in V_i \cup V_j$ as above. Let*

$$\begin{aligned}
u_i(x,z) &:= \mathrm{Res}_y(f_i, \mathrm{Res}_z(f_j, \frac{\partial f_j}{\partial z})) = (x-p_x)^{u_0^{(i)}}\,\tilde{u_i}(x,z)\\
u_j(x,z) &:= \mathrm{Res}_y(f_j, \mathrm{Res}_z(f_i, \frac{\partial f_i}{\partial z})) = (x-p_x)^{u_0^{(j)}}\,\tilde{u_j}(x,z)\\
v_i(x,z) &:= \mathrm{Res}_y(f_i, \mathrm{Res}_z(f_i, f_j)) = (x-p_x)^{v_0^{(i)}}\,\tilde{v_i}(x,z)\\
v_j(x,z) &:= \mathrm{Res}_y(f_j, \mathrm{Res}_z(f_i, f_j)) = (x-p_x)^{v_0^{(j)}}\,\tilde{v_j}(x,z)
\end{aligned}$$

and the following definitions of the exponents

$$\begin{aligned}
u_0^{(i)} &:= \max\{u' \,:\, (x-p_x)^{u'} | u_i(x,z)\}\\
u_0^{(j)} &:= \max\{u' \,:\, (x-p_x)^{u'} | u_j(x,z)\}\\
v_0^{(i)} &:= \max\{v' \,:\, (x-p_x)^{v'} | v_i(x,z)\}\\
v_0^{(j)} &:= \max\{v' \,:\, (x-p_x)^{v'} | v_j(x,z)\}
\end{aligned}$$

Define $Z^{|}_{p,i,j} := Z_{p,i} \cup Z_{p,j} \cup Z'_{p,i} \cup Z'_{p,j} \cup Z^*_{p,i} \cup Z^*_{p,j}$ with

$$Z'_{p,i} := \begin{cases} \{z \in \mathbb{R} | \tilde{u_i}(p_x, z) = 0\} & , \text{ if } p \in V_i \wedge \tau_{S_j}(p) = 0 \\ \emptyset & , \text{ otherwise} \end{cases}$$

$$Z^*_{p,i} := \begin{cases} \{z \in \mathbb{R} | \tilde{v_i}(p_x, z) = 0\} & , \text{ if } p \in V_i \wedge \tau_{0,S_i,S_j}(p) = 0 \\ \emptyset & , \text{ otherwise} \end{cases}$$

and similar for $Z'_{p,j}$ and $Z^*_{p,j}$

Then for $z_0 \notin Z^{|}_{p,i,j}$ the local topology of $\mathcal{A}_{\{S_i,S_j\},z_0}$ at p is preserved for any sufficiently small perturbation of z_0.

Intuitively, $Z^{|}_{p,i,j}$ decomposes ℓ_p into intervals such that each face F of $\mathcal{A}_{\{S_i,S_j\}}$ incident to p is adjacent to exactly one such interval. This ensures the boundary property for multi-surface z-lifts of a multi-surface arrangement. The proof of Theorem 5.62 is analog to Theorem 5.60. To actually compute $Z^{|}_{p,i,j}$, we again rely on the bitstream Descartes method for u_k and v_k (as we previously did for r, s, and t), while the final merging of sets into $\mathcal{Z}_{p,\{S_i,S_j\}}$ is analog to the merge presented for $Z_{p,i}$ with $p \in V_i$ using Algorithm 5.4. This construction also shows that Definition 5.20 is well-chosen for algebraic surfaces. It remains to proof the complexity of $\mathcal{Z}_{p,\{S_i,S_j\}}$

Proof. **(of Theorem 5.47)** Observe that we only have to show that $|Z^{|}_{p,i,j}| \in O(D^3)$ according to Definition 5.18. We already have $|Z_{p,i}| \in O(D^3)$ and $|Z_{p,j}| \in O(D^3)$. The remaining sets that define $Z^{|}_{p,i,j}$ are determined by roots of polynomials whose degrees in z are at most $O(D^3)$. Thus, each can have up to $O(D^3)$ real roots, which implies the desired bound for the union of them. □

Choosing the bitstream Descartes method to compute the isolators is not an arbitrary decision. First of all, the Descartes method is considered to be a practically efficient root isolation method, and using numerical approximations of the coefficients is experienced to speed up the computation further [Str06], [CJK02], [Bro02]. Thus, our choice for the Bitstream Descartes aims for practical efficiency, but it has another advantage: The algorithm guarantees a successful real root isolation for the square-free case by a randomized choice of subdivision points, and by its adaptive precision management — regardless of the polynomial's root separation. This implies, that we never have to switch to a symbolic root isolator. The same guarantee is given for the m-k-variant. Only if the polynomial is algebraically difficult, that is, it has several multiple roots, it must be made square-free by symbolic computation; see Algorithm 2.4. However, the obtained square-free part can again be tackled with the original version of the Bitstream Descartes method. In case of the vertical-line "isolation", our implementation relies on robust curve-analyses. For our purpose, they can be considered as a sophisticated variant of the bitstream Descartes method.

Remark (Semi-algebraic surface). If we consider a semi-algebraic surface, for example, a sphere with a removed cap, the functor has to modify its report. In particular, the inequalities $g_i \geq 0$ that restrict $f = 0$ also restrict $Z_{p,i}$ for a given p. That is, we first compute $Z_{p,f}$, but only report those $z_{p,f,l}$ that fulfill $\forall i : g_i(p, z_{p,f,l})) \geq 0$.

Equality

We next discuss how to implement `Equal_z` that should decide the equality of $z_{p,1,l_1} \in Z_{p,1}$ and $z_{p,2,l_2} \in Z_{p,2}$. Remember that we already filtered some cases; see Algorithm 5.5. However, sometimes we still need this *external* answer for algebraic surfaces. Our solution is to compute the local gcd $g_p := \gcd((f_1)_{(d_{p,1})}(p), (f_2)_{(d_{p,2})}(p))$ at p. This can be done using Algorithm 2.8. Even better, by Lemma 2.13, we can directly set $g_p := \mathrm{Sres}_{k_{p,1,2}}((f_1)_{(d_{p,1})}, (f_2)_{(d_{p,2})}, z)_p$, as $k_{p,1,2}$, $d_{p,1}$ and $d_{p,2}$ are known and interfaced for the cell $\Gamma \in \mathcal{A}_{\{S_1,S_2\}}$ that contains p.

To decide the equality, we only have to check whether the intervals for $z_{p,1,l_1}$ and $z_{p,2,l_2}$ are both isolating for g_p. In case that g_p only contains simple roots, this task can be solved by evaluating g_p at the boundaries of $z_{p,1,l_1}$'s available approximation (and similar for $z_{p,2,l_2}$) and to check whether they have different signs. The local gcd g_p is surely square-free if $k_{p,1,2} = 1$, or if $k_{p,1} = 0$ or $k_{p,2} = 0$. Otherwise, we isolate g_p's roots by interpreting $g_p := \mathrm{Sres}_{k_{p,1,2}}((f_1)_{(d_{p,1})}, (f_2)_{(d_{p,2})}, z)$ as algebraic surface and call `Construct_isolator`. Observe that this algebraically complex case (several intersections) implies that g_p must be made square-free using Algorithm 2.4.

Note that this functor is also used to decide equality of entries of $Z_{p,1}$, $Z_{p,2}$ and $Z^{|}_{p,1,2}$ when computing $\mathcal{Z}_{p,\{S_1,S_2\}}$ for $p \in V_1 \cup V_2$.

Adjacency

The final task expected from an algebraic surface S_i by the *SurfaceTraits_3* concept is to compute the adjacency relation of its z-patterns. As noticed, it suffices to compute it only for z-fibers of incident cells in $\mathcal{A}_{\{S_i\}}$. We next explain how to implement the `Adjacency` functor. Remember that we are basically given $Z_{p,1}$ and $Z_{p,2}$. Each of its entries has an index l_1 and l_2. We are aiming for the list L of pairs of indices that define the adjacency relation as in Problem 5.12. We make a case-distinction over the dimensionality of the planar cells.

Edge-face adjacencies: Let E be an edge of $\mathcal{A}_{\{S_i\}}$, and let F denote an adjacent face. The boundary property allows us to pick E's sample point p_E in its interior to proceed. We assume that E is non-vertical and p_E is chosen such that its x-coordinate is rational.[51] If $d_{p_E} = D_i$ and p has been lifted with the m-k-variant, then all but one roots of $f_i(p_E)$ are simple. The cells over E to which these simple roots belong have precisely one adjacent lift over F. The remaining lifts over F must be adjacent to the possibly multiple root over E. This strategy to obtain adjacencies has already been applied in [GVN02], [Ber04], and [EKW07].

Otherwise, the implementation is similar to the one in [ACM88]. Determine $q = (q_x, q_y)$ for F with $q_x = p_x$, and $q_x, q_y \in \mathbb{Q}$ and consider the planar curve $f_i|_{x=p_x} := f(p_x, y, z) \in \mathbb{Q}[y, z]$. The l_F-th lift $F^{(l_F,i)}$ of F is adjacent to the l_E-th lift $E^{(l_E,i)}$ of E if and only if there is an arc of the curve $V_{\mathbb{R}}(f_i|_{x=p_x})$ connecting the l_F-th point over q_y with the l_E-th point over p_y. To compute the adjacencies of $V_{\mathbb{R}}(f_i|_{x=p_x})$ we rely on CGAL's `Algebraic_curve_kernel_2`; see also [EKW07]. An illustration is given in Figure 5.9.

Adjacencies of a vertex Let p be the vertex V's point. Let us assume first, that $p \notin V_i$. We consider the other case separately below.

Note that the vertex has at least one incident face, and if there is more than one, there are also incident edges. Let F be such a face and E be such an edge. First observe, that if p has been successfully lifted with the m-k-variant, the same idea as for the edge-face-adjacencies applies for adjacencies between V and E and between V and F.

Second, due to Condition 5.9, the adjacencies between V and some E can often be derived by a transitivity argument: Let F_1 and F_2 be the faces to both sides of E. If every lift over V is adjacent to a lift over F_1 or F_2, knowing the adjacencies between V and F_1 and between F_1 and E, or between V and F_2 and between F_2 and E, directly gives the adjacencies between V and E as well; see also Figure 5.9.

In case that $f(p)$ has more than one multiple root, or some lift over V is connected to an isolated lift of an edge E (i. e., the lifted edge has no incident lifted face), we implement the following *bucketing* strategy:

Choose rational values $q_{-1}, \ldots, q_{m_{p,i}-1}$ such that $q_{l-1} < z_{p,i,l} < q_l$ for all $l = 0, \ldots, m_{p,i}-1$. The $m_{p,i}+1$ many planes $z = q_l$ divide the real space in $m_{p,i}+2$ many *buckets* that separate the lifts over V: One for each entry of $Z_{p,i}$; even $\pm\infty$. The

[51] Otherwise, p_E's y-coordinate is rational and we proceed analogously.

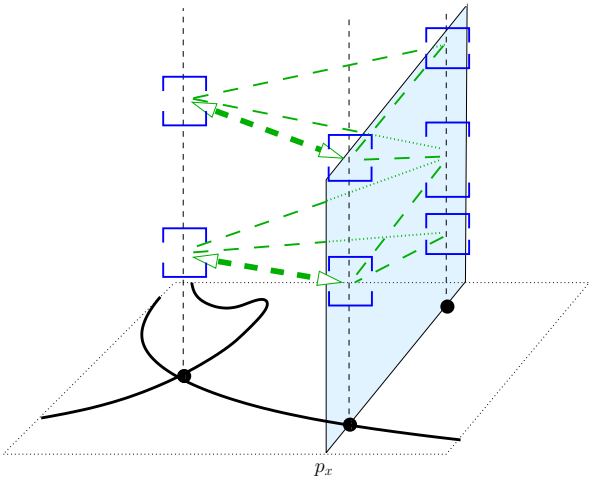


Figure 5.9. First: Edge-face adjacency is given by analyzing $V_{\mathbb{R}}(f_i|_{x=p_x})$. Second: Vertex-face adjacencies and face-edge adjacencies are known (without arrows). Thus, vertex-edge adjacency (with arrows) can be deduced by transitivity.

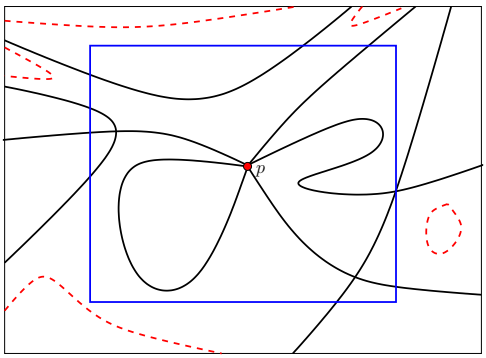
buckets help find points on incident faces and edges whose lifts uniquely determine the adjacency relation.

Definition 5.63 (Bucket-loyal). Let $\Gamma \in \mathcal{A}_S$ be incident to V. We say that $p' \in \Gamma$ is *bucket-loyal* if there is a path $\Pi \subset \Gamma$ from p' to p such that each lift $\Pi^{(l,i)}$ stays in the l-th bucket.

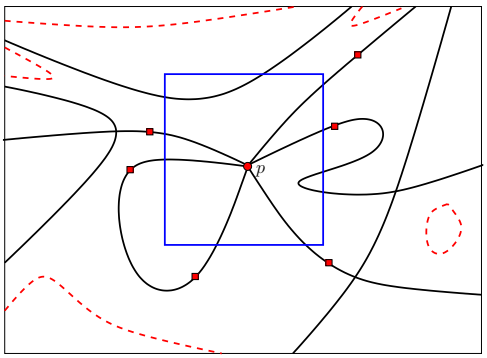
Thus, finding a bucket-loyal point p' for an incident DCEL-cell Γ of V gives a possibility to compute the adjacency between V's lifts and Γ's lifts: If the l'_p-th lift over p' lies in the bucket of $z_{p,i,l}$, then, the lifts $V^{(l,i)}$ and $\Gamma^{(l'_p,i)}$ are adjacent. Lifts of Γ that belong to the bottom-most and top-most bucket are special: The z-coordinate of Π's endpoint at p is $+\infty$ or $-\infty$, that is, they belong to asymptotic lifts. If $p \notin V_i$, Condition 5.9 implies that for each Γ incident to V, there exists a bucket-loyal path Π. However, we have to compute the z-fiber of f_i over p'. If p' is too close to p, then $f_i(p')$ has a bad root separation, which we want to avoid. Thus, we next propose a strategy to find good bucket-loyal points for the cells incident to V.

- The first crucial observation is that $p' \neq p$ is bucket-loyal if and only if Π does not intersect any of the *bucket curves* b_l defined by $f_i(x, y, q_l) \in \mathbb{Q}[x, y]$. As $\ell_p \not\subset S_i$ no bucket curve intersects p by construction. Following, we can define a *bucket box* B around p such that it does not contain any of the bucket curves; see the dashed red curves in Figure 5.10 (a). We exploit interval arithmetic to reach the goal: Approximate p's x- and y-coordinate as intervals $[p_x]$, $[p_y]$ and use them to evaluate $I_l := f([p_x], [p_y], q_l)$ for all $l = -1, \ldots, m_{p,i}$. As long as some resulting interval I_l contains zero, refine p's approximation and proceed. The final approximation of p defines B. Each point on B's boundary and inside B is bucket-loyal; see Definition 5.63.
- Next, we shrink B such that each incident cell of V has a bucket-loyal point on B's boundary. This is done by choosing a sample point for each edge E incident

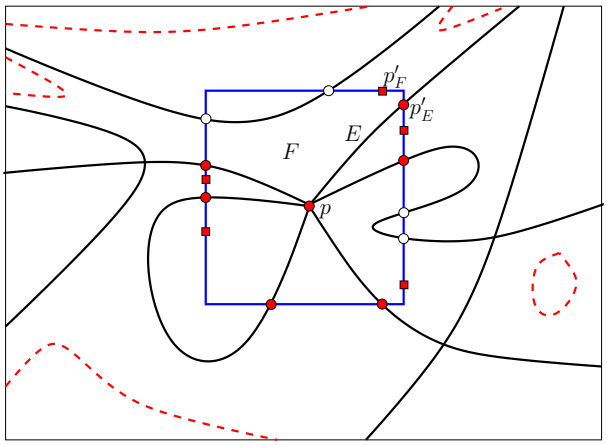
Figure 5.10. Computing bucket-local points around a vertex (schematic)



(a) Compute initial bucket box B that does not intersect any bucket curve



(b) Refine B with respect to sample points of incident edges



(c) Compute intersections of B with edges and determine bucket-loyal points for edges and incident faces

to V and refine B such that these sample points are outside B;[52] an illustration is given in Figure 5.10 (b).

- We finally compute all intersection points of $\mathcal{A}_{\{S_i\}}$ with B's boundary. By construction of B and Condition 5.9, the bucket-loyal point p'_E for an edge E incident to V is given by the first intersection of E with B's boundary, when traversing E starting in p. Consider next the face F that succeeds E in counter-clockwise order on the boundary of B. The intersection points of $\mathcal{A}_{\{S_i\}}$ and B's boundary are also ordered counter-clockwisely. Let p'_{next} the intersection that succeeds p'_E. The desired point p'_F is given by a point on B's boundary between p'_E and p'_{next}. Note that by construction the path between p and p'_F is bucket-loyal. Figure 5.10 (c) illustrates the two cases.

We have implemented this strategy, which gives us the desired adjacencies; see also Figure 5.11 which illustrates the bucketing in three dimensions. There is one missing case for $p \notin V_i$. Namely, the vertex V can be isolated in some F. In this case, we choose p'_F on the vertical line $x = p_x$ with $p'_F \in F$ and having a rational y-coordinate.

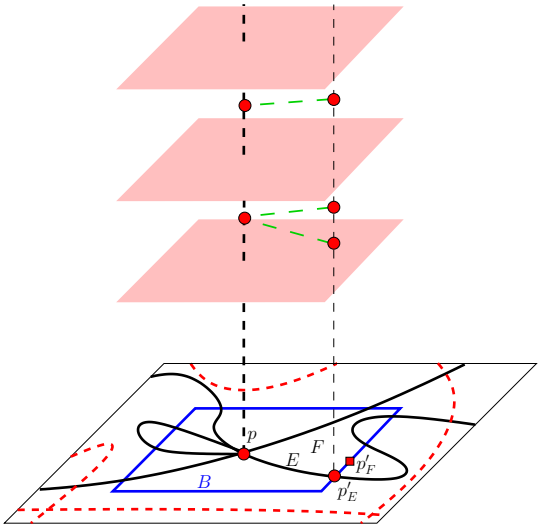


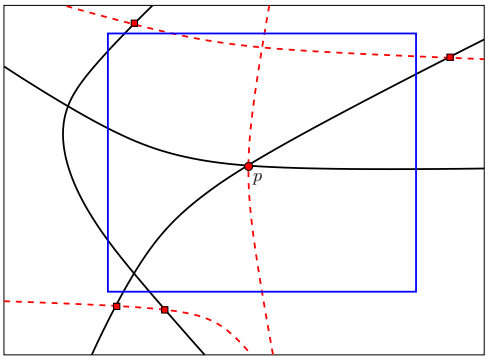
Figure 5.11. Computing adjacencies: here between vertex and edge

We admit, that the strategy exploits similar ideas as the local box algorithm by Collins and McCallum [MC02] for cylindrical algebraic decompositions. The main difference is that our bucket box construction only involves cheap interval arithmetic, and thus is expected to be more efficient. In addition, their local box algorithms requires to factorize polynomials, while we provide a purely geometric algorithm. On the other side, this complicates the actual construction, as we have to deal with incident cells that are not x-monotone.

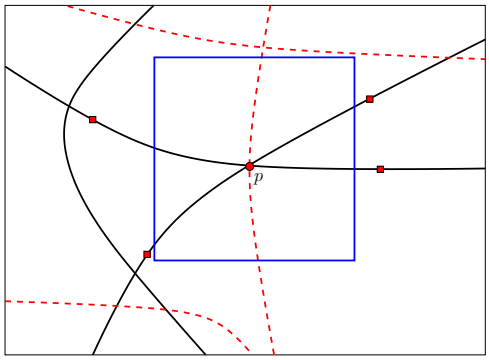
Adjacencies for vertical line cells: We turn to the case that V is defined by a point p with $p \in V_i$. In general, we proceed similar to the previous case. However, there

[52] Note that we consider E to be a maximal d-k-path emanating V and not an x-monotone curve incident to V actually maintained in the underlying arrangement of CGAL. Thus, E can be a self-loop.

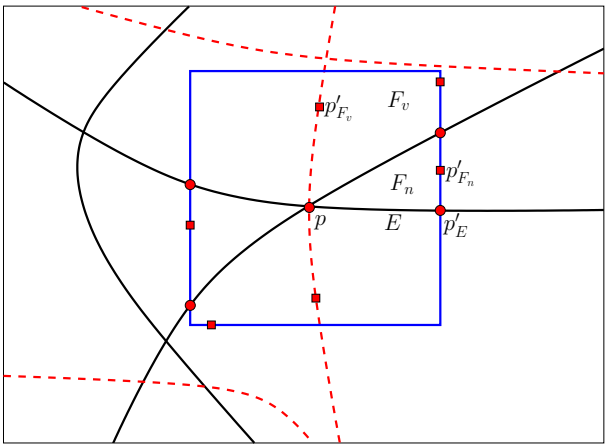
Figure 5.12. Computing bucket-loyal points around a vertex of a vertical line (schematic)



(a) Compute initial bucket box B such that no intersection of a bucket curve with $\mathcal{A}_{\{S_i\}}$ is inside B



(b) Refine B with respect to sample points of incident edges



(c) Locate points on bucket curves to detect adjacency of a lifted face to a vertical line; then, choose bucket-loyal points for remaining edges and faces

is one complication here: If V has an incident face F that is adjacent to a non-degenerate interval along ℓ_p, then, the corresponding bucket curve b_l (with q_l being in the interior of the interval) intersects p in the plane. Following, we need a fix for the construction of the bucket box, as interval arithmetic is not sufficient.

- To determine B's initial size, we now compute the overlay of $\mathcal{A}_{\{S_i\},\mathcal{B}}$ with all bucket curves. The initial B is chosen such around p that no intersection of B and $\mathcal{A}_{\{S_i\},\mathcal{B}}$ is contained in B, except the unavoidable intersections at p itself; see Figure 5.12 (a) for an illustration.
- As before, we refine B, such that the sample point p_E of each incident edge E is not contained in B anymore; see Figure 5.12 (b).
- The corresponding bucket-loyal point p'_E is again given by traversing E starting in p and choosing the first intersection of E with B. Lifting p'_E reveals by Condition 5.9 and how we defined $Z_{p,i}$ the desired adjacencies. Figure 5.12 (c) displays this step.
- The strategy for a face F is different from the previous handling. We start to detect lifts of faces that are adjacent to a line segment along ℓ_p. Each such segment has to contain an intermediate bucket value q_l. Following, the lift of $F^{(l_F,i)}$ that is adjacent to the interval results in an arc of the bucket curve b_l that lies in F (in $\mathcal{A}_{\{S_i\}}$) and ends in p. To find them we propose the following strategy: For each bucket curve b_l that leaves p and lies in F, let q_l be corresponding bucket value. Choose a sample point p_{b_l} on b_l but inside B (see Figure 5.12 (c)) and lift it. Note that its lifting corresponds to lifts of F. Determine which $F^{(l_F,i)}$ has the z-coordinate q_l (by interval arithmetic). Following Problem 5.12, we report the pairs (l_F, l) and $(l_F, l+1)$; an illustration in three dimensions is given in Figure 5.13.
- Finally, we are left with the lifts of F that are adjacent to a single point on ℓ_p. We simply compute a bucket-loyal point p'_F as in the $p \notin V_i$ case and determine the buckets of the remaining lifts analogously. This gives the full adjacency relation. The face adjacencies are also presented in Figure 5.12 (c).

Remark. Remember that we also have to compute special adjacencies between $\mathcal{Z}_{p_0,\mathcal{S}}$ and $\mathcal{Z}_{p_1,\mathcal{S}}$ with $p_0 \in \mathcal{V}$ and $p_1 \notin \mathcal{V}$ for a given S_i. The critical case is $p_1 \in F \in \mathcal{A}_{\mathcal{S}}$ with F being incident to p_0. Note that the ideas of the bucketing strategy also lead to a successful computation. The difference is that the number of buckets defined over p_0 has increased and we have to ignore other surfaces existing in the multi-surface z-fiber $\mathcal{Z}_{p_1,\mathcal{S}}$.

There is also the possibility of combinatorial filtering: We only have to consider those buckets of $\mathcal{S}_{p_0,\mathcal{S}}$ that comprise z-coordinates in the finite z-range of $Z_{p_0,i}$. This reduces the number of bucket curves in the plane. On the other side, we have to maintain a mapping between all and the selected buckets. If we are only interested in a single incident face F to p_0, we can even further restrict the z-range that must be considered by querying the single-surface adjacency of lifts of p_0 and p_1 first, which gives us $I(F^{(l_F)}) \subset \ell_{p_0}$ for the correct l_F.

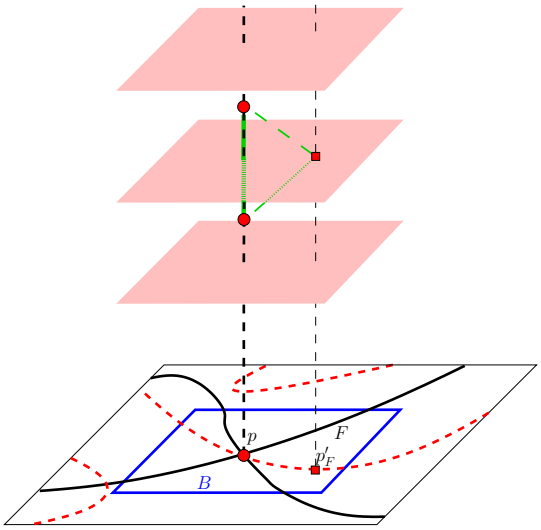


Figure 5.13. Computing adjacencies at a vertical line: a lifted face is incident to an interval along ℓ_p

Alternative idea The adjacency relation of a lifted vertex with its lifted incident cells can also be determined by analyzing a two-dimensional arrangement embedded on a vertical cylinder C around ℓ_p. The radius of C is chosen such that $c := C|_{z=0}$ fits in the box defined by B as above and the center of c should be equal to p — in theory. Below, we show that we can actually perturb c's center to rational coordinates.

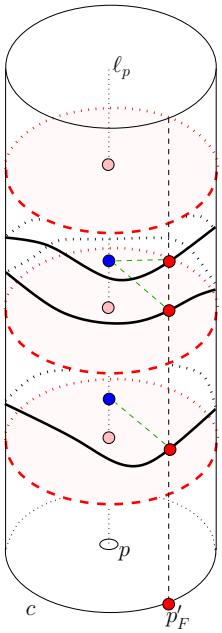
Consider the arrangement on C induced by $V_{\mathbb{R}}(f)$ and all $V_{\mathbb{R}}(z = q_l)$. The later induces a set of horizontal circles around the cylinder that split C again into *buckets*. $V_{\mathbb{R}}(f)$ also induces curves on C.

Let $\ell_{p'} \subset C$ be parallel to ℓ_p, that is, choose a point a point p' on c. If $p \notin V_i$, then, by construction of C the point p' is bucket-loyal for the incident cell Γ of V with $p' \in \Gamma$. This means, that along $\ell_{p'}$ the curves $V_{\mathbb{R}}(f)$ and $V_{\mathbb{R}}(z = q_l)$ do not intersect. The adjacency relation between lifts of V and lifts of Γ can be determined by the order of $V_{\mathbb{R}}(f)$ and $V_{\mathbb{R}}(z = q_l)$ along $\ell_{p'}$ — the status line at p'. We only have to find p' for all incident edges and faces: Compute the intersections of c with $\mathcal{A}_{\{S\}}$. Think of c being the boundary of B and proceed as above.

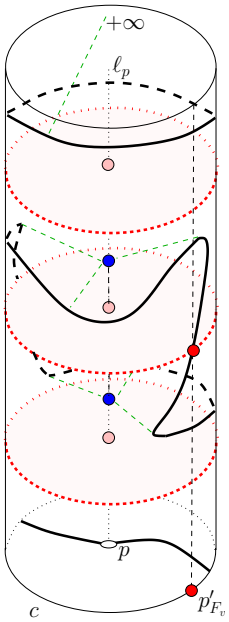
An interesting phenomena can be seen for the lift of face $F^{(l_F,i)}$ that is adjacent to an interval of ℓ_p. In this case, there is a p'_F on c defined by the c's intersection with a bucket curve b_l, where $V_{\mathbb{R}}(f)$ intersects the circle defined by $V_{\mathbb{R}}(z = q_l)$ along the line $\ell_{p'_F}$. The value l determines to which interval along ℓ_p the lift $F^{(l_F,i)}$ is adjacent. For all other faces (and edges), we proceed as before.

By now, the approach is identical to the box idea, with the difference that we chose a circle as planar base. But we can also analyze the arrangement on C itself which gives further information. For simplicity, we assume that $p = (0, 0)$ and $r = 1$. Then, C can be rationally parameterized by $\varphi_C(t, z) = (\frac{1-t^2}{1+t^2}, \frac{2t}{1+t^2}, z)$, for $t, z \in \mathbb{R}$. Homogenizing it, allows to us compute the arrangement on C with the help of a set of real algebraic plane curves; in fact, the topology is similar to the one we introduced in §4.6.1 for quadrics, while

Figure 5.14. Adjacencies by a two-dimensional arrangement on a cylinder



(a) All points of c are bucket-loyal. Thus, $V_{\mathbb{R}}(f)$ (black) on C stays within one bucket



(b) $V_{\mathbb{R}}(f)$ crosses some $V_{\mathbb{R}}(z = q_l)$. Thus, there is a lift of a face $F^{(l_F,i)}$ that is adjacent to $p \times [z_{p,i,l}, z_{p,i,l+1}]$.

concerning the geometry, we should adopt ideas similar to what we did for a Dupin cyclide in §4.6.2. Special care is only required for $t = \pm\infty$, that is, C's *curve of identification*. However, as $V_{\mathbb{R}}(x = p_x)$ is not part of S_i (by input assumption), the corresponding curve-ends for $t = \pm\infty$ have a unique limit. Or we simply consider $p' = (-1, 0)$ as an additional special case on the boundary. Let $\mathcal{A}_C$ be the arrangement on C induced by $V_{\mathbb{R}}(f)$ and $V_{\mathbb{R}}(z = q_l)$. The critical t-coordinates of $\mathcal{A}_C$ are given by the t-extremal points of $V_{\mathbb{R}}(f)$ and t-coordinates of $V_{\mathbb{R}}(f)$'s intersections with some $V_{\mathbb{R}}(z = q_l)$ only appear if some lift of an incident face is adjacent to an interval along ℓ_p. Note that neither $V_{\mathbb{R}}(z = q_l)$ nor $V_{\mathbb{R}}(f)$ have self-intersections on C. For $V_{\mathbb{R}}(z = q_l)$ this is clear by construction, for $V_{\mathbb{R}}(f)$ this follows from how we choose the radius of c: No target point of an edge E that leaves p is inside c. Observe that the t-critical points of $\mathcal{A}_C$ correspond to points on c where edges E or bucket curves b_l intersect c. This gives a more direct way to compute (bucket-loyal) points for edges and faces. Computing the adjacencies now reduces to find all such t-coordinates and to analyze the vertical lines of $\mathcal{A}_C$ in the tz-plane for such coordinates. We start with the intersections of b_l with c, if existing, to detect face lifts adjacent to vertical intervals. Then, we proceed with the t-extremal points of $V_{\mathbb{R}}(f)$, to determine adjacencies for lifted incident edges and finally analyze the status line slightly to the right of a t-extremal coordinate. This gives us the adjacencies for lifted incident faces that only meet a point along ℓ_p.

In terms of CGAL's `Arrangement_2` package, we can imagine a special visitor that only reports the adjacencies while sweeping over a set of annotated curves (i. e., whether each belongs to $V_{\mathbb{R}}(f)$ or $V_{\mathbb{R}}(z = q_l)$) on C.

In an actual implementation one would better choose a cylinder whose center line ℓ_C is slightly perturbed away from p, such that p_C has rational-coordinates. The perturbation must be chosen such that $C|_{z=0}$ still defines bucket-loyal points. Actually, every vertical cylinder inside $B \times \mathbb{R}$ that includes ℓ_p in its interior fulfills this condition.

5.4.3. Filters for lifting of quadrics

If we only consider quadrics, the functors related to lifting can benefit from combinatorial filters. We next present the details.

Construct_isolator We first characterize the entries of some $Z_{p,i}$.

Lemma 5.64. *Let $f_i = a_2 z^2 + a_1 z^1 + a_0 z^0$ be a polynomial defining a quadric S_i, that is, $\deg_{\text{total}}(f) \leq 2$. Let $p \in \mathbb{R}^2$. Then, $f_i(p_x, p_y, z) \in \mathbb{R}[z]$ has either no real root, a double real root, or two distinct real roots. If it has a double root, then $\text{Res}_z(f_i, \frac{\partial f_i}{\partial z})(p_x, p_y) = 0$. Contrary if $\text{Res}_z(f_i, \frac{\partial f_i}{\partial z})(p_x, p_y) = 0$ then, the z-fiber at p contains at most one finite point.*

Proof. The first assertion is rather trivial. If z_0 is a multiple root of $f_i(p_x, p_y, z)$ then it is also a root of $\frac{\partial}{\partial z} f_i(p_x, p_y, z)$, thus $\text{Res}_z(f_i, \frac{\partial f_i}{\partial z})(p_x, p_y) = 0$. For $a_2(p) = 0$ and $p \notin V_i$ the backward direction is trivial as in this case $f_i(x, y, z)$ is a polynomial of degree one or less in z for all (x, y). If $a_2(p) \neq 0$ and $\text{Res}_z(f_i, \frac{\partial f_i}{\partial z})(p_x, p_y) = 0$ the polynomials $f_i(p_x, p_y, z)$ and $\frac{\partial f_i}{\partial z}(p_x, p_y, z)$ must share a common root z_0, thus for $p \notin V_i$, the z-fiber $Z_{p,i}$ is given by $\{z_0\} \cup \{\pm\infty\}$. In case where S_i contains a vertical line at p, we refer to the paragraph about *vertical lines* below. □

Using Lemma 5.64 we isolate with the Bitstream Descartes method the real roots of $f_i(p_x, p_y, z)$ in case $k_{p,i} = 0$ or of $\frac{\partial f_i}{\partial z}(p_x, p_y, z)$ if $k_{p,i} > 0$. Observe that both polynomials fulfill the demanded property of being square-free, while still determining the S_i's intersection with ℓ_p. Note that we are able to combinatorially avoid to call the m-k-variant.

Equal_z Remember that we have to decide whether $z_{p,1,l_1} \in Z_{p,1}$ and $z_{p,2,l_2} \in Z_{p,2}$ are equal. However, we can benefit from previous information: When `Equal_z` is called, each $Z_{p,1}$ and $Z_{p,2}$ has to contain a positive number of finite entries. As they correspond to real roots of $f_1(p)$ and $f_2(p)$, Lemma 5.64 implies that we see one or two such. In addition, we know that the approximations of $Z_{p,1}$ and $Z_{p,2}$ have already been refined such that they overlap with at most one interval of the other. As τ_{0,S_1,S_2} exists at p and $Z_{p,1}$ and $Z_{p,2}$ contain finite entries, at least one of these candidates must correspond to a true equality; see also Algorithm 5.5. Thus, most cases are trivial to decide. Only if $|Z_{p,1}| = |Z_{p,2}| = 2$ we require further work. Two possibilities exists:

1. Both $f_1(p,z) = a_2(p)z^2 + a_1(p)z + a_0(p)$ and $f_2(p,z) = b_2(p)z^2 + b_1(p)z + b_0(p)$ have two distinct real roots and they are both equal at the given p. That is, there exists a constant $c \in \mathbb{R} \setminus \{0\}$ with $f_1(p,z) = c \cdot f_2(p,z)$. This is exactly the case if the two vectors
$$(a_2(p), a_1(p), a_0(p))^T$$
and
$$(b_2(p), b_1(p), b_0(p))^T$$
are linear equivalent, which can be checked by
$$\begin{aligned}(a_0b_1 - a_1b_0)(p) &= 0 \wedge\\ (a_0b_2 - a_2b_0)(p) &= 0 \wedge\\ (a_1b_2 - a_1b_2)(p) &= 0\end{aligned}$$
Note that
$$\begin{aligned}h_{0,1} &:= (a_0b_1 - a_1b_0) \in \mathbb{Q}[x,y],\ \deg_{\text{total}}(h_{01}) \leq 3\\ h_{0,2} &:= (a_0b_2 - a_2b_0) \in \mathbb{Q}[x,y],\ \deg_{\text{total}}(h_{02}) \leq 2\\ h_{1,2} &:= (a_1b_2 - a_1b_2) \in \mathbb{Q}[x,y],\ \deg_{\text{total}}(h_{12}) \leq 1\end{aligned}$$
and even $\deg_{\text{total}}(h_{i,j}) < \deg_{\text{total}}(\text{Res}_z(f_1, f_2))$ holds. Thus, we check whether the three conditions are fulfilled by interpreting $h_{i,j}$ as low-degree planar curves, and test whether p lies on them with the `Sign_at_2` functor provided by CGAL's planar `Algebraic_curve_kernel_2`. This functor even exploits interval arithmetic to quickly decide a non-zero sign. Note that the algebraic kernel is available anyhow, as we use it for the projection. For each pair of quadrics only one such set of curves is required, so we can cache them. Of course, we start testing with $h_{1,2}$ as it has lowest degree. We continue with the $h_{0,2}$ only if the test result is successful. Similar for $h_{0,2}$ and $h_{0,1}$. If all three conditions hold, then two common roots exists (out of two possible). Thus, return `true`. This case is illustrated in Figure 5.15.
2. Otherwise, two candidate overlaps remain for a single equality. We refine their approximations in parallel, until only one overlap is left. If the given indices l_1 and l_2 correspond to that overlap, return `true`, else return `false`.

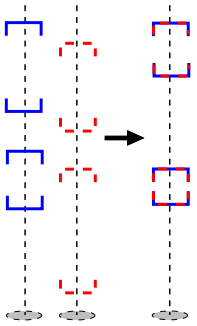


Figure 5.15. Illustration of two covertical intersections (case 1)

Adjacency This functor is implemented mostly combinatorially. First observe that by Lemma 5.64, the adjacencies between lifted vertices and lifted edges, if existing, are fixed. We always return $(0,0)$ as the unique lifts all lie in the same plane defined by $\frac{\partial f}{\partial z}$. The case that a vertex corresponds to a vertical line is discussed separately below.

Thus, we are left with edge-face-adjacencies. Let $0 \leq m_E \leq 1$ be the number of finite lifts of some E, and $0 \leq m_F \leq 2$ be the number of finite lifts of an incident face. Note that not both can be zero. If $m_E = 0$, then $m_F = 1$ and the $F^{(0)}$ must be asymptotic when "approaching" E. In this direction $F^{(0)}$ is monotonic increasing or decreasing, which can be determined by computing the sign of $\frac{\partial f_i}{\partial z}$ evaluated at F's sample point with interval arithmetic. Otherwise, $m_E = 1$ and we are left with two cases: $m_F = 1$ implies to return $(0,0)$, while $m_F = 2$ results in reporting $(0,0)$ and $(0,1)$ (by Lemma 5.64 and Condition 5.9).

Quadrics and vertical lines: A quadric S_i contains a vertical line ℓ_p at $p \in \mathbb{R}^2$ exactly if $a_2(p) = 0$ and p is an intersection point of the line $L = V_{\mathbb{R}}(a_1(p))$ and the conic $C = V_{\mathbb{R}}(a_0(p))$. Then, for each point $p \notin L$, there exists a unique lift $(p_x, p_y, z) \in S_i$ with $z = -\frac{a_0(p)}{a_1(p)}$. Furthermore, there exists no point on S_i above any $p \in L \setminus (L \cap C)$ and for each (of at most 2) intersection point $p \in L \cap C$ the quadric contains the vertical line ℓ_p. The arrangement $\mathcal{A}_{\{S_i\}}$ as defined in §5.2.2 is quite simple in this situation: The projected silhouette L divides $\mathbb{R}^2$ into two half-planes, which are the faces F_1 and F_2 of $\mathcal{A}_{\{S_i\}}$. The intersection points $L \cap C$ represents all vertices in $\mathcal{A}_{\{S_i\}}$ and they decompose L into at most 3 edges. As these edges cannot be lifted onto S_i, no adjacencies between them and vertices can be reported.

In the following steps we will show how to determine the fiber $Z_{p,i}$ for the vertices with $p \in V_i$ and how to get the adjacency information between vertices and faces. In Theorem 5.45 we have already proven that for each face $F = F_1$ or $F = F_2$ there exists a corresponding interval I_F such that for each $z^* \in I_v$ we have a sequence $p_t \in F$, converging against p, with $z^* = \lim_{t\to\infty} z_{p_t} = \lim_{t\to\infty} -\frac{a_0(p_t)}{a_1(p_t)}$.

From an affine change of coordinates we can assume that $L = V_{\mathbb{R}}(y)$, that is, L is the x-axis. Writing $a_0(x,y) = c_0x^2 + c_1y^2 + c_2xy + c_3x + c_4y + c_5$ with variable coefficients $c_i \in \mathbb{R}$, for the z-value of any $(x,y,z) \in S_i$ we have

$$z = \frac{c_0x^2 + c_1y^2 + c_2xy + c_3x + c_4y + c_5}{y}$$

For a fixed $y \neq 0$ the set of z-values is given by a parabola ($c_0 \neq 0$), which has its unique local extremum $z_{\max,y}$ at the point $x_{\max}$ with

$$2c_0 x_{\max,y} + c_2 y + c_3 = 0$$

Thus, we get $x_{\max,y} = \frac{-c_3 - c_2 y}{2c_0}$ and

$$z_{\max,y} = \frac{4c_0c_5 + 4yc_0c_4 - 2yc_2c_3 - c_3^2 + 4y^2c_0c_1 - y^2c_2^2}{4yc_0}$$

Now we distinguish three cases:

1. $c_0 = 0$: C and L intersect in a unique point $p = (-\frac{c_5}{c_3}, 0)$. Now given an arbitrary $z^* \in \mathbb{R}$, we have

   $$f(x_y, y, z^*) = 0 \Leftrightarrow x_y(c_2y + c_3) = yz^* - c_1y^2 - c4y - c5$$

   For $y \to 0$ we get $x_y \to -\frac{c_5}{c_3}$, thus the conic C_{z^*}, which is implicitly given by the equation $x(c_2y + c_3) = yz^* - c_1y^2 - c4y - c5$ passes the point p and the face F. It follows the existence of a sequence $p_t \to p \in V_{\mathbb{R}}(a_0) \cap V_{\mathbb{R}}(a_1)$ with $z_{p_t} \to z^*$.

2. $|V_{\mathbb{R}}(a_0) \cap V_{\mathbb{R}}(a_1)| = 0$ or 2 and $c_0 \neq 0$: For a fixed $z^* \in \mathbb{R}$, the conic $C_{z^*} = V_{\mathbb{R}}(f(x, y, z^*))$ has exactly two intersection points with L. Hence, C_{z^*} has only ordinary intersections with L. Thus, C_{z^*} contains an arc that passes $V_{\mathbb{R}}(a_0) \cap V_{\mathbb{R}}(a_1)$ and the face F. It follows the existence of a sequence $p_t \to p \in V_{\mathbb{R}}(a_0) \cap V_{\mathbb{R}}(a_1)$ with $z_{p_t} \to z^*$.

3. $|V_{\mathbb{R}}(a_0) \cap V_{\mathbb{R}}(a_1)| = 1$: In this case the quadratic polynomial $a_0(x, 0)$ has a multiple root and $p = V_{\mathbb{R}}(a_0) \cap V_{\mathbb{R}}(a_1) = (\frac{-c_3}{2c_0}, 0)$. Hence, we get $c_3^2 - 4c_5c_0 = 0$ and furthermore

   $$\lim_{y\to 0} z_{\max,y} = \lim_{y\to 0} \frac{4yc_0c_4 - 2yc_2c_3 + 4y^2c_0c_1 - y^2c_2^2}{4yc_0}$$

   $$= \frac{2c_0c_4 - c_2c_3}{2c_0}$$

   Now, the line $L_{\max}$, implicitly given by $2c_0x + c_2y + c_3 = 0$ passes the point p and contains a sequence of points $p_t \to p$ with $z_{p_t} \to \frac{2c_0c_4 - c_2c_3}{2c_0} =: z_{p,0}$ for $t \to \infty$. In the next step we will show that for any other sequence $p'_t \to p$ we must get $z_{p_t} \to z' \geq z_{p,0}$ or $\leq z_{p,0}$ depending on whether $c_0 > 0$ or $c_0 < 0$. W.l.o.g. we assume that $c_0 > 0$. Then, for a fixed y the parabola $\frac{c_0x^2 + c_1y^2 + c_2xy + c_3x + c_4y + c_5}{y}$ has a global minimum $z_{\max,y}$ at $x_{\max,y}$. Thus, for any point $p'_t = (x'_t, y'_t)$ we must have $z_{p'_t} = -\frac{a_0(x'_t, y'_t)}{a_1(x'_t, y'_t)} \geq z_{\max,y'_t}$. It follows that $\lim_{p'_t \to p} z_{p'_t} \geq \lim_{p'_t \to p} z_{\max,y'_t} = z_{p,0}$.

In the two cases (1) and (2) we have shown that the unique lifts of the two faces F_1 and F_2 are both adjacent to any point on the vertical line ℓ_p, that is, for any $(p, z^*) \in \ell_p$ there exist sequences $p_t^{(l,i)} \in F_j$ with $(p_t^{(l,i)}, z_{p_t^{(l,i)}}) \to (p, z^*)$ for $l = 1, 2$. Thus, it suffices that $Z_{p,i} = \{\pm\infty\}$, that is, `Construct_isolator` returns an empty instance and so `Equal_z` is trivial. It is clear that `Adjacency` returns for an incident lifted face $F^{(l,i)}$ towards p the pairs $(0, -1)$ and $(0, 0)$.

In the third case, for exactly all z^* in between $\frac{2c_0c_4-c_2c_3}{2c_0}$ and $\pm\infty$ (depending on whether $c_0 > 0$ or $c_0 < 0$, respectively) there exists a sequence $p_t^{(l,i)} \in F_j$ with $p_t^{(l,i)} \to p$ and $z_{p_t^{(l,i)}}) \to z^*$. As we can also pursue the affine coordinate transformation in case where f_i is given with arbitrary variable coefficients, it is possible to get formulas in terms of these coefficients to decide in which case we are and to determine the single non-infinity entry $z_{p,i,0}$ of $Z_{p,i}$. Observe that $p = (p_x, p_y)$ and $z_{p,i,0}$ are all rational. Thus, `Equal_z` can be implemented in terms of rational arithmetic. `Adjacency` returns for an incident face towards the vertical line at p, either the pairs $(0,-1),(0,0)$ or the pairs $(0,0),(0,1)$, depending on the sign of c_0 and the face.

5.5. Applications

The proposed design and its implementation provides three-dimensional information on a set of surfaces $\mathcal{S}$, that is, we compute a stratification $\Omega_{\mathcal{S}}$ enhanced with geometric information. The basic structure is a planar arrangement whose vertices, edges, and faces can be queried to obtain the third dimension. The adjacencies of lifts also provide connectivities. Although it is remarkable that the richness of computed information is rather complex, it is also abstract. In addition, we have seen that the complexity of $\Omega_{\mathcal{S}}$ is quite high due to the projection and the lifting; even for relatively small D.

On the other hand, based on this decomposition, it is possible to rely on the framework as a key ingredient when providing or supporting more concrete applications in geometric computing. In this section, we present a list of such. For some, we only give basic ideas; their details require further work. Other applications are illustrated more elaborate; for example, the computation of space curves (§5.5.3) and lower envelopes (§5.5.4).

5.5.1. Analysis of a single surface

Stratification Given a single surface S. First of all, we can simply report the stratification of S along with its full adjacency information. This, for example, supports the localization of a query point in the stratification. That is, we return the cell to which a point on S belongs. The cell decomposition can also be queried to detect three-dimensional cells induced by S. In fact, we can use adjacency information to cluster them into maximal sets. Note that this is a major steps towards the three-dimensional arrangements induced by S. This also paves the way to locate any $p \in \mathbb{R}^3$ with respect to S. However, we are mainly missing a data structure to efficiently store these clusters.

Sampling A big advantage of our method is that geometric information is kept with respect to the original coordinate system. Thus, we can sample S in arbitrary precision, including its critical points. This possibility can be exploited, for example, in a visualization: Once, $\mathcal{A}_{\{S\}}$ is computed, a dense grid of points is located and lifted using the adaptive bitstream Descartes method. It only requires to specify the grid-width and the maximal length of the intervals that approximate z-coordinates to define the desired precision of the sampling. The implementation consists in construction the grid and to refine the liftings. Note that lifting is a perfect tasks for a parallelized computations as we should usually have a much larger number of liftings than available processors. We mention this

possibility, as modern computers are usually equipped with multi-core architectures and thus constitute an ideal platform for this objective.

Meshing A desirable goal is to compute a simplified representation for S, for example, in the form of a mesh. That is, we aim for a simplicial complex that is isotopic to S and whose points are located on S. This complex cannot be directly extracted from $\Omega_{\{S\}}$. In order to maintain the topology of S further decompositions are required. We refer to [BKS] for on-going work of such a triangulation of an algebraic surface.

It should also be analyzed how many triangles are required to form a complex that is isotopic to S, but whose vertices are not required to lie on S. There is a gap for algebraic surfaces: A decomposition of S with degree D into non-singular cells requires $\Omega(D^3)$ cells [Bru81]. In contrast, using a cylindrical algebraic decomposition (cad) results in a complex with $O(D^7)$ cells. It is unknown where the true value is.

5.5.2. Analysis of two or more surface

Stratification As for a single surface, we also have seen how to compute the stratification for a set of surfaces $\mathcal{S}$, that is, respecting their intersections, too. The localization of a point in the set of strata is a task that is directly supported by the framework. Again, the adjacency relation for any two cells is available, which enables similar to the single-surface case, to identify induced three-dimensional cells, and to cluster them into maximal connected sets. Although the boundary of such a cell can be described, this only constitutes a restricted representation of the three-dimensional arrangement induced by $\mathcal{S}$. We omit details on point localization, sampling, and meshing as they are similar to the single-surface case.

Semi-algebraic sets We mentioned that the framework also supports semi-algebraic surfaces. But not only in their handling, but also for their representation: We can extract the decomposition of such a surface $S_{\geq}$ defined by $f = 0$ and a set of polynomial inequalities $g_i \geq 0$, $1 \leq i \leq r$ into connected zero-, one-, and two-dimensional cells having the boundary property. Note that all points of a single lift of a cell in $\mathcal{A}_{\{S_{\geq}\}}$ share the same signs with respect to all g_i. We only have to select those cells whose signs are all non-negative by choosing the stratum defined by the inequalities.

5.5.3. Space curves

The stratification of two surfaces also allows to extract the space curve defined by two surfaces:

Definition 5.65 (Space curve). A *space curve* is the intersection set of two surfaces S_1, S_2, if at most one-dimensional.

To represent a space curve, one usually decomposes it into its zero- and one-dimensional parts, where zero-dimensional parts form isolated points, while the one-dimensional arcs can have properties, like x- or xy-monotonicity. Our implementation provides C++ class templates called `Surface_point_3` and `Surface_arc_3`. The representation of a point is organized as a tuple `(Point_2, Surface_3, int)`, that is, a planar base point, a supporting surface, and its lift index (also known as sheet number). x- and y-coordinate are given

explicitly by the planar point, the z-coordinate is encoded implicitly by the other two types. A bounded one-dimensional arc in 3D is represented as a tuple `(Surface_point_3, Surface_point_3, X_monotone_curve_2, Surface_3, int, int, int)`, where the points encode the lexicographic smallest and largest point of the arc. The remaining entries lift the planar curve onto the given surface. The `int` instances encode sheet numbers at the lexicographic smallest and largest point, and in the interior of the arc, where the number must be constant. Note that all three can even be equal or different. The supporting surfaces of the arc and its minimal and maximal point are not required to be equal. Special constructions for unbounded and vertical arcs are implemented, but omitted in this description. Similar in Algorithm 5.6 that computes the decomposition of the space-curve defined by S_1 and S_2 into isolated vertices and one-dimensional arcs:

Algorithm 5.6. Decompose space curve into arcs and points

INPUT: Two surfaces S_1, S_2 with $\dim(S_1 \cap S_2) \leq 1$
OUTPUT: The decomposition of $S_1 \cap S_2$ into arcs of dimension 1 and isolated points of dimension 0.

1. Compute $\mathcal{A}_{\{S_1,S_2\}}$ and extract vertices and edges belonging to τ_{0,S_1,S_2}.
2. Obtain for each such vertex and each such edge its `Z_fiber`; identify their `Z_cell` instances that define an intersection.
3. Compute for each lift of an edge that forms an intersection of S_1 and S_2 to which lifts of vertices it is adjacent.
4. For an edge, the `Z_fiber` and the adjacencies give all information required to construct instances of type `Surface_arc_3`.
5. An isolated vertex in 3D is detected and constructed by checking whether there exists a `Z_cell` instance over a DCEL-vertex that is not adjacent to any `Z_cell` over edges supported by of τ_{0,S_1,S_2} and incident to the vertex. It remains to construct proper instances of type `Surface_point_3` using the available information.

We remark, that there a subtleties to consider. For example, a `Surface_point_3` instance for a lifted vertex should be computed only once, especially if several arcs of intersections are adjacent to it.

A careful reader might detect that this approach requires to compute both $\mathcal{A}_{\{S_1\}}$ and $\mathcal{A}_{\{S_2\}}$. Observe, that the output is not demanding for both surfaces at the same time. It suffices to express the decomposition of a space curve into points and arcs only in terms of the surface with *lower complexity*, for example, the degree of an algebraic surface. Let S_1 be the surface with lower complexity.

We next show how to avoid the computation of $\mathcal{A}_{\{S_2\}}$ and $\mathcal{A}_{\{S_1,S_2\}}$. However this requires to refine the *SurfaceTraits_3* concept by an additional functor called `Common_z`. It is expected to provide the following operator:

```
Z_at_xy_isolator operator()(
   Point_2 pt, Surface_3 s1, Surface_3 s2,
   Cell_info1 ci1
)
```

In contrast to `Equal_z`, which only checks the equality for given intervals, `Common_z` constructs a new instance of type `Z_at_xy_isolator` that represents the common interesting z-coordinates of `s1` and `s2` along a vertical line defined by the given `pt`. Due to lacking $\mathcal{A}_{\{S_2\}}$ and $\mathcal{A}_{\{S_1,S_2\}}$, we do not have access to full knowledge about multiplicities, regularities, and

degradations with respect to $\{S_2\}$ and $\{S_1, S_2\}$ (collected usually in cell-info instances). Thus, `Common_z` has to deal without these information. It depends on the family of surfaces, how to compute the desired isolator. For algebraic surfaces, we have seen that the roots of the local gcd define the required z-coordinates. Compare also with the implementation details of `Equal_z` in §5.4.2. However, there the degradation k is accessible from the planar arrangement. In our current setting, we have to compute it. We also need two adaptions. First, Algorithm 5.6 does not compute $\mathcal{A}_{\{S_1,S_2\}}$, but only the overlay of $\mathcal{A}_{\{S_1\}}$ and $\mathcal{A}_{\tau_{0,S_1,S_2}}$ and traverses its edges and vertices. Second, we require a new algorithm to construct the `Z_fiber`, replacing the usual two-way merge:

Algorithm 5.7. Compute `Z_fiber` for a DCEL-cell participating in τ_{0,S_1,S_2}

INPUT: $p \in \mathbb{R}^2$; surfaces S_1, S_2
OUTPUT: `Z_fiber` for S_1, S_2 over p

1. Construct `Z_at_xy_isolator iso1` for S_1 using `Construct_isolator`.
2. Construct `Z_at_xy_isolator iso12` for intersections of S_1 and S_2 using `Common_z`.
3. Refine intervals of `isolator12` until each is included in an interval of `isolator1`.
4. Create `Z_cell` for each interval of `isolator1` and add S_2 to a cell, if there is an interval of `iso12` that overlaps with an interval of `isolator1`.

Observe that the surface lifts of S_2 in the computed `Z_cell` instances cannot be enhanced with a sheet number. Fortunately, this is also not needed, as Algorithm 5.6 only wants to detect cells where S_2 exists, but its output is with respect to S_1's sheet numbers only.

We have implemented this output-sensitive strategy in a class-template called `Curve_3`. We consider it as a basic implementation that can be used whenever space curves are computed by relying on their projection into the xy-plane. In this light, this work can be seen as a prototypical implementation of a key ingredient for an upcoming `Curved_kernel_3` in CGAL.

5.5.4. Lower envelope

We can also regard the surfaces in $\mathcal{S}$ as functions in x and y that return for given $p = (p_x, p_y)$ the smallest z-coordinate of the surface's intersections with ℓ_p; requiring $V_i = \emptyset$ is a good assumption for this task. Taking for every point of the plane the set of surfaces that attain the minimum of these functions, we compute the *lower envelope of* $\mathcal{S}$; see also Chapter 3, where we present a specialized version for quadrics. Remember that CGAL provides a generic divide-and-conquer approach to compute lower envelopes [Mey06b]; see also Algorithm 3.1. One only has to provide a model of CGAL's *EnvelopeTraits_3* concept, which itself is a refinement of CGAL's *ArrangementTraits_2* concept; details on the tasks expected by the concept are given in [MWZ07] or §3.3. In this section, we present a generic implementation of such a model, called `Surface_3_envelope_traits`, that is based on `Projection_2` and attached instances of type `Z_fiber` provided by our new framework.

Let `Surface_traits_3` be the given model of the *SurfaceTraits_3* concept. The new `Surface_3_envelope_traits` class template is derived from `Surface_traits_3::Kernel_2` in order to be a model of CGAL's *ArrangementTraits_2* concept. We also have to define spatial types:

`Surface_3` and `Xy_monotone_surface_3` are expected. The former is trivial, the latter is mapped to lifted DCEL-cells, that is, a pair consisting of a DCEL-handle and an integer.

The integer corresponds to a sheet number. It can be assumed to be 0 if we only consider lower envelopes. For more sophisticated envelopes, other values are conceivable. The EnvelopeTraits_3 concept expects to decompose an instance of type `Surface_3` into its xy-monotone subsurfaces by a functor `Make_xy_monotone_3`. Our generic implementation traverses all Dcel-cells of the corresponding $\mathcal{A}_{\{S_i\}}$. Faces with non-empty z-pattern are reported, while for edges and vertices with non-empty z-pattern it first must be check whether no lift of an incident planar cell adjacent to the lowest lift over the edge and vertex, respectively, exists.

Two functors implement the required projections.

- `Construct_projected_boundary_2`

  Computes for a given xy-monotone subsurface its projected boundary. To provide this information for a subsurface over a face, we traverse the face's boundaries and distinguish whether the cycle that contains a boundary curve is oriented clockwise or counter-clockwise in order to decide to which side the xy-monotone subsurface exists. Subsurfaces that correspond to lifts of edges and vertices do not require this test. We simply report the adjoined geometric object.

- `Construct_projected_intersection_2`

  Computes the projected intersection curves of two xy-monotone subsurfaces supported by S_i and S_j. If $S_i = S_j$, we only have to return curves (points) if lifted faces (edges) are adjacent to the same lifted edge (vertex). Otherwise, we compute $\mathcal{A}_{\{S_i,S_j\}}$ and traverse all edges (and isolated vertices) in its cells that originate from the given Dcel-handles stored along with the subsurfaces. We discard those not participating in τ_{0,S_i,S_j}, those with an empty z-pattern, and those whose lowest z-cell does not contain S_i and S_j. The remaining edges and vertices are returned. The intersections tests for isolated lifted edges and vertices are similar.

The concept also requires to implement functors that compare the relative alignment of two `Xy_monotone_surface_3` instance in z-direction over a point, over a curve, or over a face incident to a projected intersection curve (i. e., a sub-face of the projected curve boundaries). Obviously, if their supporting surfaces S_i and S_j are equal, the stored sheet-numbers encode the desired order. Otherwise the vertical alignment can be read from a z-pattern of an appropriate cell of $\mathcal{A}_{\{S_i,S_j\}}$. We only have to pick the correct one, which is simple for the implementation of `Compare_z_at_xy_3`: Depending on the operator, we can directly take the `Z_fiber` for the given point, or take the one for the sample point of the given curve (or the single unbounded face). Computing the `Z_fiber` for the remaining functors `Compare_z_at_xy_below_3` and `Compare_z_at_xy_above_3` reduces to locate the sample point of the given curve in $\mathcal{A}_{\{S_i,S_j\}}$. This task is directly supported by Cgal's `Arrangement_2` package on which we rely our framework.

Remark. Using this generic model of the *EnvelopeTraits_3* concept, computing (lower) envelopes for a family of surfaces boils down to provide a model of the *SurfaceTraits_3* concept for that class of surfaces. We admit that a specialized model for lower envelopes might be more efficient, but obviously lacks of the possibility to support other applications that we introduced in this section. The reason is, that we compute more information on how two surfaces intersect than actually required for the lower envelope; compare also

with the construction of the Apollonius digram in §3.5, that is similarly "direct". However, our implementation is the first that follows the exact geometric computing paradigm to compute lower envelopes of algebraic surfaces.

As in §3.5, we can think of modifications on the `Surface_3_envelope_traits`, such as to compute upper envelopes, or sandwich regions. One should also check various dualities that allow to rewrite a geometric problem as a envelope computation of surfaces; see, for example, [dBvKOS00, §11.5].

5.6. Results

We also run experiments to check the efficiency of our implementation(s). In the following we report on various tests that present different aspects of the framework, but also show the limits of practicality. We distinguish between experiments on quadrics and such on algebraic surfaces of any degree.

5.6.1. Quadrics

We tested the performance of the framework instantiated with the `Quadric_3_traits` model by computing all z-fibers and all adjacencies for $\mathcal{A}_\mathcal{S}$, where $|\mathcal{S}|$ increases. We especially distinguish between arbitrary quadrics and ellipsoids. All experiments are executed on a Pentium IV CPU with 3.0 GHz clock-speed and 2 MB of cache. The executables are compiled with gnu's C++-compiler in version 3.3 with disabled debugging (`-DNDEBUG`) and enabled optimizations (`-O2`), and CGAL's `Algebraic_curve_kernel_2` in wrapping mode with the exact number types of LEDA. Table 5.1 lists example runs.

♯Surfaces	♯DCEL	♯z-cells	t	t/cell
2 ellipsoids	13	12	**0.1s**	5.7ms
4 ellipsoids	230	904	**2.8s**	3.4ms
6 ellipsoids	877	5942	**19.9s**	3.7ms
8 ellipsoids	2780	25220	**171.9s**	7.2ms
10 ellipsoids	4952	52788	**582.0s**	11.5ms
2 quadrics	53	160	**0.4**	2.7ms
4 quadrics	1099	7172	**19.7**	3.0ms
6 quadrics	3946	39254	**194.4**	5.4ms
8 quadrics	9983	132352	**2306.1**	18.1ms

Table 5.1. Performance measures for sets of ellipsoids and arbitrary quadrics

It cannot be hidden, that $|\mathcal{S}|$ seems quite small, but on the other hand, the size of the output grows rapidly. For 8 quadrics we already have to compute nearly 10.000 z-fibers containing more than 130.000 cells. However, these numbers match the analyzed complexities of $\Omega_\mathcal{S}$ in §5.1. On the contrary, the time spent per cell grows much slower. In fact, we have to see an increasing amount of time here, as by construction of the data, similar intervals along each ℓ_p are intersected by a growing number of quadrics, that is, it requires additional time to isolate the cells against each other. Anyhow, we can conclude that the implementation computes for a non-trivial set of quadrics, the correct stratification in reasonable time (per cell). Nevertheless, we recommend to use this in applications that

typically involve only a small number of surfaces at a time. An example is the computation of an arrangement on a reference quadric, as in §4.6.1: Besides the reference itself, we only have a second surface in focus, namely, when decomposing their intersection curve into (weakly) x-monotone arcs and isolated points. We can remark now, that the decomposition of these spaces curves in the reported implementation and experiments are actually realized using the ideas presented in §5.5.3. Previously, in Table 4.3 we mainly ignored the column dedicated to the splitting time. However, this column actually shows the performance of an aggregated space curve construction keeping one surface fixed. The required time simply grows linear with the number of quadrics intersecting the reference grows. This is sensible as we only consider two quadrics at a time.

We are finally able to compare the new approach, that is, using the tools presented in this thesis, with our former implementation [BFH+07]. As example we chose increasing sets of random ellipsoids. Figure 5.16 shows an overall improvement of about 30%. For general quadrics the ratio is similar. Let us pick a concrete set: An arrangement induced by 400 ellipsoids intersecting the reference ellipsoid consists of about 38.000 vertices and 74.000 edges, which is now possible to compute in around 180s instead of 287s.

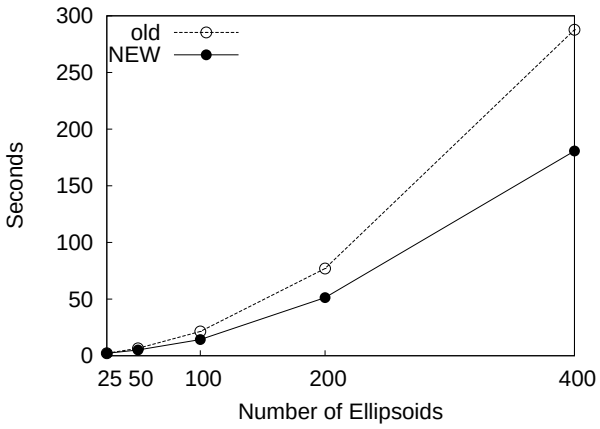


Figure 5.16. Running times to compute arrangements on an ellipsoid: We compare the implementation from [BFH+07] (2005) with the one based on ideas in this thesis (2008).

5.6.2. Algebraic surfaces

We also run experiments on algebraic surfaces, that is, we compute their decomposition as presented in §5.1. As input we have chosen well-known examples from algebraic geometry,[53] random and interpolated instances, and also a generic projection of two quadrics in 4D. All experiments are executed on an AMD Dual-Core Opteron(tm) 8218 (1 GHz) multi-processor platform. Each processor has an internal cache of 1 MB and the total memory consists of 32 GB. The system runs Debian Etch. We compiled using `g++-4.1.2` with flags `-O2 -DNDEBUG` and use the exact number types of CORE [KLPY99]. For planar arrangements of algebraic curves, we relied on CGAL's internal `Algebraic_curve_kernel_2`

[53] Subsets of the tested example surfaces are provided courtesy of INRIA by the AIM@SHAPE Shape Repository, by `www.singsurf.org`, by `www.freigeist.cc`, and by [PV07]

Surface S	$\deg_{x,y,z}$	(#V,#E,#F)	$\lvert\Omega_S\rvert$	t
steiner-roman	2,2,2	(5,12,8)	28	**0.73**
cayley-cubic	2,2,2	(3,10,8)	31	**0.74**
dupin-cyclide	4,4,4	(3,4,4)	10	**0.19**
tangle-cube	4,4,4	(0,6,7)	28	**0.61**
bohemian-dome	4,4,4	(7,20,14)	61	**0.75**
chair	4,4,4	(4,9,7)	31	**3.05**
hunt	6,6,6	(3,2,3)	15	**1.21**
star	6,6,6	(1,1,2)	5	**3.61**
spiky	6,9,6	(1,8,8)	13	**1.43**
C8	8,8,8	(40,48,26)	496	**30.95**
random-3	3,3,3	(2,3,3)	15	**0.17**
random-4	4,4,4	(7,14,8)	64	**4.50**
random-5	5,5,5	(16,24,10)	154	**236.40**
interpolated-3	3,3,3	(4,6,3)	23	**0.34**
interpolated-4	4,4,4	(12,18,9)	82	**31.41**
projection-4d	4,4,4	(4,12,9)	34	**10.33**

Table 5.2. Complexity and running times (in seconds) for the stratification of a selection of surfaces. Defining polynomials are reported in Appendix A.

in non-wrapping mode. Observe that our software currently does not benefit from having several processors, although many steps of the algorithm are well-suited for parallel computations, such as the lifting or adjacency computation.

Table 5.2 reports for a selection of tested surfaces the size of the computed (d,k)-arrangement $\mathcal{A}_{\{S\}}$, the total number of cells in Ω_S, and the obtained running times (in seconds). It is also expected, that (some) surfaces do not show any (d,k)-vertex (e. g.,`tangle-cube`), or (d,k)-edge (e. g., xy-functional surfaces) at all. Concerning the running times, we observed that about 90% is spent to construct $\mathcal{A}_{\{S\}}$. This is no surprise, as we have to analyze plane algebraic curves of degree up to $D(D-1)$. The remaining 10% are consummated for the computation of the lifts and adjacencies. The success of the m-k-filter depends on the surface. For most of the tested surfaces, it fails in less than 10% of the non-square-free liftings, while for the highly-degenerate "C8" example no execution is successfull. Concerning running time, if $\deg_z(f)$ is low (≤ 3), computing the square-free part with subresultants is not expensive. However, with increasing $\deg_z(f)$, the m-k-filter shows its power. A drastic example is the "star"-surface that only requires two critical lifts. For one, the filter is successful and only needs a fraction of a second. If switching off the filter, the total running time increases from less than 4 seconds to more than 25 seconds.

We finally can conclude that especially the lifting and adjacency steps benefit from chosen approximative and combinatorial methods, such as the bitstream Descartes method and its m-k-variant, interval arithmetic, propagations of available information, and a careful selection of sample points required for the adjacency computations. A naive approach would result in real root isolations along ℓ_p with a very bad separation, which typically increases running times tremendously.

5.7. Conclusion and outlook

Achievements: We presented a generic realization of surface stratifications with full adjacency information. Our C++-implementation is supported by CGAL's `Arrangement_2` package. Its design is kept simple, the interface intuitive, and the approach taken does not enforce to assume generic position. We decoupled combinatorial from geometric tasks. A new family of surfaces can be used by implementing a small set of tasks defined by the newly introduced *SurfaceTraits_3* concept. We provide models for this concept: One for algebraic surfaces of any degree, and one for quadrics. This second implements degree-specific combinatorial filters.

Our work demonstrates that surface analysis is practically feasible for moderate degrees. The experiments show promising results thanks to our circumspectly cell decomposition and the consequent application of approximate methods. However, as the number of cells in our decomposition still grows fast, we see the main application of this tool in providing information for a small set of surfaces, that is, to compute the topology (and geometry) of a single surface, a single space-curve, or to serve as a key ingredient for high-level algorithms like the computation of envelopes, or three-dimensional arrangements. Some of them are already presented and implemented, others require further work.

Future directions: As a first step, we want to generalize further, that is, to remove the last algebraic terminology. In particular, most of the tasks are already expressed in the favored generic language. A strategy to achieve this goal could be to abstract the concept while developing models for other kinds of surfaces, for example, Bézier patches. A straightforward model that we have in mind, is to support rotated surfaces — similar to the ideas for conics in the plane; see [BCW07].

We also want to elaborate further utilizations of the computed data. For example, there is on-going work to extract an isotopic triangulation from an enhanced cell decomposition [BKS]. Another showcase is the computation of a single Voronoi cell of a set of planes, spheres, and cylinders. The current implementation provided by [HE08] relies on non-certified analyses of low-degree algebraic surfaces (i. e., $D \leq 4$). We consider our contribution as perfectly suited to easily certify this subproblem, which finally results in a fully certified algorithm — in C++. Additionally, it should be checked in how far our analyses of surfaces support, for a given set of algebraic surfaces, to compute the Voronoi cell for each of them.

We finally consider the provided decompositions as an important building block for full three-dimensional arrangements of algebraic surfaces, and boolean operations on the induced cells. Having this, we are able to robustly compute instances describing the configuration space for a rotational robot whose movements are restricted by polygonal obstacles [Lat93]. This task is also known as the Piano Mover's problem; see [SSH87]. If we finally manage to combine fast subdivision approaches with our exact and certified analyses, the approach is expected to be reasonable efficient — knowing that the obtained result is ultimately correct.

Bibliography

[Abb06] John Abbott. Quadratic Interval Refinement for Real Roots. Poster presented at the 2006 International Symposium on Symbolic and Algebraic Computation (ISSAC 2006), 2006.

[ACM84] Dennis S. Arnon, George E. Collins, and Scott McCallum. Cylindrical Algebraic Decomposition II: An Adjacency Algorithm for the Plane. *S*IAM Journal on Computing, 13:878–889, 1984. Reprinted in [CJ98], pp. 152–165.

[ACM88] Dennis S. Arnon, George E. Collins, and Scott McCallum. An Adjacency Algorithm for Cylindrical Algebraic Decompositions of Three-Dimensional Space. *J*ournal of Symbolic Computation, 5:163–187, 1988.

[AHW07] Lars Arge, Michael Hoffmann, and Emo Welzl, editors. *A*lgorithms - ESA 2007, 15th Annual European Symposium on Algorithms, Eilat, Israel, October 8-10, 2007, Proceedings, volume 4698 of *L*NCS. Springer, 2007.

[Arn88] Dennis S. Arnon. A Cluster-Based Cylindrical Algebraic Decomposition Algorithm. *J*ournal of Symbolic Computation, 5:189–212, 1988.

[AS00] Pankaj K. Agarwal and Micha Sharir. Arrangements and Their Applications. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *H*andbook of Computational Geometry, pages 49–119. Elsevier Science Publishers B.V., 2000.

[AS01] Marcus Vinícius A. Andrade and Jorge Stolfi. Exact Algorithms for Circles on the Sphere. *J*ournal of Computational Geometry & Applications, 11(3):267–290, 2001.

[AS05] Juan Gerardo Alcázar and Juan Rafael Sendra. Computation of the Topology of Real Algebraic Space Curves. *J*ournal of Symbolic Computation, 39:719–744, 2005.

[ASS96] Pankaj K. Agarwal, Otfried Schwarzkopf, and Micha Sharir. The Overlay of Lower Envelopes and its Applications. *D*iscrete Computational Geometry, 15:1–13, 1996.

[ASS07] Juan Gerardo Alcázar, Josef Schicho, and Juan Rafael Sendra. A Delineability-based Method for Computing Critical Sets of Algebraic Surfaces. *J*ournal of Symbolic Computation, 42:678–691, 2007.

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

[BCSM$^+$] Jean-Daniel Boissonnat, David Cohen-Steiner, Bernard Mourrain, Günter Rote, and Gert Vegter. Meshing of Surfaces. In [BT06], pp. 181–229.

[BCW07] Eric Berberich, Manuel Caroli, and Nicola Wolpert. Exact Computation of Arrangements of Rotated Conics. In *P*roceedings of 23rd European Workshop on Computational Geometry, pages 231–234, Graz, Austria, March 2007. Technische Universität Graz.

[BD96] Jean-Daniel Boissonnat and Katrin T. G. Dobrindt. On-line Construction of the Upper Envelope of Triangles and Surface Patches in Three Dimensions. *C*omputational Geometry: Theory and Applications, 5(6):303–320, 1996.

[BE08] Eric Berberich and Pavel Emeliyanenko. CGAL's Curved Kernel via Analysis. Technical Report of [1] with number ACS-TR-123203-04, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2008.

[BEH+02] Eric Berberich, Arno Eigenwillig, Michael Hemmer, Susan Hert, Kurt Mehlhorn, and Elmar Schömer. A Computational Basis for Conic Arcs and Boolean Operations on Conic Polygons. In *P*roceedings of 10th European Symposium on Algorithms, volume 2461 of *L*ecture Notes in Computer Science, pages 174–186. Springer-Verlag, 2002.

[BEH+05] Eric Berberich, Arno Eigenwillig, Michael Hemmer, Susan Hert, Lutz Kettner, Kurt Mehlhorn, Joachim Reichel, Susanne Schmitt, Elmar Schömer, and Nicola Wolpert. EXACUS: Efficient and Exact Algorithms for Curves and Surfaces. In *P*roceedings of 13th Annual European Symposium on Algorithms (ESA'05), volume 3669 of *L*ecture Notes in Computer Science, pages 155–166, October 2005.

[BEPP97] Hervé Brönnimann, Ioannis Z. Emiris, Victor Y. Pan, and Sylvain Pion. Computing Exact Geometric Predicates Using Modular Arithmetic with Single Precision. In *P*roceedings of 13th Symposium on Computational Geometry, pages 174–182, 1997.

[Ber04] Eric Berberich. *Exact Arrangements of Quadric Intersection Curves*. Universität des Saarlandes, Saarbrücken, Germany, 2004. Diplomarbeit.

[Bez07] Helmut E. Bez. Rational Maximal Parametrisations of Dupin Cyclides. In Ralph R. Martin, Malcolm A. Sabin, and Joab R. Winkler, editors, *M*athematics of Surfaces XII, volume 4647 of *L*NCS, pages 78–92. Springer, 2007.

[BFH+07] Eric Berberich, Efi Fogel, Dan Halperin, Kurt Mehlhorn, and Ron Wein. Sweeping and Maintaining Two-Dimensional Arrangements on Surfaces: A First Step. In Arge et al. [AHW07], pages 645–656.

[BFM+01] Christoph Burnikel, Stefan Funke, Kurt Mehlhorn, Stefan Schirra, and Susanne Schmitt. A Separation Bound for Real Algebraic Expressions. In Friedhelm Meyer auf der Heide, editor, *A*lgorithms - ESA 2001 (ESA-01) : Proceedings of the 9th Annual European Symposium,, volume 2161 of *L*ecture Notes in Computer Science, pages 254–265, Aarhus, Denmark, August 2001. Springer.

[BFWZ07] Eric Berberich, Efi Fogel, Ron Wein, and Baruch Zukerman. Sweeping Curves and Maintaining 2D Arrangements on Surfaces. Unpublished manuscript, 2007.

[BHK+05] Eric Berberich, Michael Hemmer, Lutz Kettner, Elmar Schömer, and Nicola Wolpert. An Exact, Complete and Efficient Implementation for Computing Planar Maps of Quadric Intersection Curves. In *P*roceedings of the 21st Annual Symposium on Computational Geometry (SCG 2005), pages 99–106, 2005.

[BHK+06a] Eric Berberich, Michael Hemmer, Menelaos Karavelas, Sylvain Pion, Monique Teillaud, and Elias Tsigaridas. Interface Specification of Algebraic Kernel. Technical Report of [1] with number ACS-TR-123101-01, INRIA Sophia-Antipolis, Max-Planck-Institut für Informatik, National University of Athens, 2006.

[BHK+06b] Eric Berberich, Michael Hemmer, Menelaos Karavelas, Sylvain Pion, Monique Teillaud, and Elias Tsigaridas. Prototype Implementation of the Algebraic Kernel. Technical Report of [1] with number ACS-TR-121202-01, INRIA Sophia-Antipolis, Max-Planck-Institut für Informatik, National University of Athens, 2006.

[BHKT07] Eric Berberich, Michael Hemmer, Menelaos I. Karavelas, and Monique Teillaud. Revision of the Interface Specification of Algebraic Kernel. Technical Report of [1] with number ACS-TR-243301-01, INRIA Sophia-Antipolis, Max-Planck-Institut für Informatik, National University of Athens, 2007.

[BHKT08] Eric Berberich, Michael Hemmer, Menelaos I. Karavelas, and Monique Teillaud. Algebraic_kernel_d. In CGAL Editorial Board, editor, *C*GAL User and Reference Manual. 3.4 edition, 2008. Internal Version.

[BK07] Eric Berberich and Lutz Kettner. Linear-Time Reordering in a Sweep-line Algorithm for Algebraic Curves Intersecting in a Common Point. Research Report MPI-I-2007-1-001, Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany, July 2007.

[BK08] Eric Berberich and Michael Kerber. Exact Arrangements on Tori and Dupin Cyclides. In Haines and McGuire [HM08], pages 59–66.

[BKS] Eric Berberich, Michael Kerber, and Michael Sagraloff. An Efficient Algorithm for the Stratification and Triangulation of an Algebraic Surface. *C*omputational Geometry: Theory and Applications. Submitted to special issue on SCG 2008.

[BKS08] Eric Berberich, Michael Kerber, and Michael Sagraloff. Exact Geometric-Topological Analysis of Algebraic Surfaces. In Monique Teillaud, editor, *P*roceedings of the 24th Annual Symposium on Computational Geometry, College Park, MD, USA, June 9-11, 2008, pages 164–173. ACM, June 2008.

[BKSV98] Hervé Brönnimann, Lutz Kettner, Stefan Schirra, and Remco C. Veltkamp. Applications of the Generic Programming Paradigm in the Design of CGAL. In Mehdi Jazayeri, Rüdiger Loos, and David R. Musser, editors, *G*eneric Programming, volume 1766 of *L*ecture Notes in Computer Science, pages 206–217. Springer, 1998.

[BM07] Eric Berberich and Michal Meyerovitch. Computing Envelopes of Quadrics. In *P*roceedings of 23rd European Workshop on Computational Geometry, pages 235–238, Graz, Austria, March 2007. Technische Universität Graz.

[BO79] Jon Louis Bentley and Thomas Ottmann. Algorithms for Reporting and Counting Geometric Intersections. *I*EEE Transactions on Computers, 28(9):643–647, 1979.

[Boe90] Wolfgang Boehm. On Cyclides in Geometric Modeling. *C*omputer Aided Geometric Design, 7:243–255, 1990.

[Bos06] Siegfried Bosch. *A*lgebra. Springer Lehrbuch. Springer-Verlag, Berlin, Heidelberg, sechste auflage edition, 2006.

[BPR06] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry*, volume 10 of *A*lgorithms and Computation in Mathematics. Springer, 2nd edition, 2006.

[Bre95] Glen E. Bredon. *Topology and geometry*, volume 139 of *G*raduate Texts in Mathematics. Springer, 1995.

[Brö01a] Hervé Brönnimann. Designing and Implementing a General Purpose Halfedge Data Structure. In *WAE* '01: Proceedings of the 5th International Workshop on Algorithm Engineering, pages 51–66, London, UK, 2001. Springer-Verlag.

[Bro01b] Christopher W. Brown. Improved Projection for Cylindrical Algebraic Decomposition. *J*ournal of Symbolic Computation, 32:447–465, 2001.

[Bro02] Christopher W. Brown. Constructing Cylindrical Algebraic Decompositions of the Plane Quickly. Unpublished, 2002.

[Bru81] J. W. Bruce. An Upper Bound for the Number of Singularities on a Projective Hypersurface. *B*ull. London Math. Soc., 13(1):47–50, 1981.

[BS08] Eric Berberich and Michael Sagraloff. A generic and flexible framework for the geometrical and topological analysis of (algebraic) surfaces. In Haines and McGuire [HM08], pages 171–182.

[BT06] Jean-Daniel Boissonnat and Monique Teillaud, editors. *E*ffective Computational Geometry for Curves and Surfaces. Springer, 2006.

[BT08] Antoine Bru and Monique Teillaud. Generic Implementation of a Data Structure for 3D Regular Complexes. In *A*bstracts of 24th European Workshop on Computational Geometry, pages 95–98. LORIA, Nancy, France, 2008.

[Büh95] Katja Bühler. Rationale algebraische Kurven auf Dupinschen Zykliden. Master's thesis, Universität Karlsruhe, August 1995. in German.

[CA76] George E. Collins and Alkiviadis G. Akritas. Polynomial Real Root Isolation Using Descartes' Rule of Signs. In Richard D. Jenks, editor, *S*YMSAC, pages 272–275, Yorktown Heights, NY, 1976. ACM Press.

[CDH89] Vijaya Chandru, Debasish Dutta, and Christoph M. Hoffmann. On the Geometry of Dupin Cyclides. *T*he Visual Computer, 5(5):277–290, 1989.

[CGA07] Editorial Board CGAL. *CGAL User and Reference Manual*, 3.3 edition, 2007.

[CGL05] Jin-San Cheng, Xiao-Shan Gao, and Ming Li. Determining the Topology of Real Algebraic Surfaces. In Ralph Martin, Helmut Bez, and Malcolm Sabin, editors, *11*. IMA Conference on the Mathematics of Surfaces, volume 3604 of *L*NCS, pages 121–146, 2005.

[CGV07] Jorge Caravantes and Laureano González-Vega. Computing the Topology of an Arrangement of Quartics. In Ralph R. Martin, Malcolm A. Sabin, and Joab R. Winkler, editors, *I*MA Conference on the Mathematics of Surfaces, volume 4647 of *L*ecture Notes in Computer Science, pages 104–120. Springer, 2007.

[CGV08] Jorge Caravantes and Laureano González-Vega. Improving the Topology Computation of an Arrangement of Cubics. *C*omputational Geometry, 41(3):206–218, 2008.

[CJ98] Bob F. Caviness and Jeremy R. Johnson, editors. *Quantifier Elimination and Cylindrical Algebraic Decomposition, Texts and Monographs in Symbolic Computation*. Springer, 1998.

[CJK02] George E. Collins, Jeremy R. Johnson, and Werner Krandick. Interval Arithmetic in Cylindrical Algebraic Decomposition. *J*ournal of Symbolic Computation, 34(2):145–157, 2002.

[CL07] Frédéric Cazals and Sébastien Loriot. Computing the Exact Arrangement of Circles on a Sphere, with Applications in Structural Biology. Technical Report 6049, INRIA Sophia-Antipolis, 2007.

[CLO97] David A. Cox, John B. Little, and Donal O'Shea. *I*deals, varieties, and algorithms. Undergraduate Texts in Mathematics. Springer, 2nd edition, 1997.

[CLO05] David A. Cox, John B. Little, and Donal O'Shea. *U*sing algebraic geometry, volume 185 of *U*ndergraduate Texts in Mathematics. Springer, New York, NY, 2nd edition, 2005.

[Col75] George E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In *S*econd GI Conference on Automata Theory and Formal Languages, volume 33 of *L*NCS, pages 134–183, 1975. Reprinted in [CJ98], pp. 85–121.

[dBHO+94] Mark de Berg, Dan Halperin, Mark Overmars, Jack Snoeyink, and Mark van Kreveld. Efficient Ray Shooting and Hidden Surface Removal. *A*lgorithmica, 12:30–53, 1994.

[dBvKOS00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, Germany, 2nd edition, 2000.

[dCPT07] Pedro M. M. de Castro, Sylvain Pion, and Monique Teillaud. Exact and Efficient Computations on Circles in CGAL and Applications to VLSI Design. Research Report 6091, INRIA Sophia-Antipoli, 01 2007.

[DFMT02] Olivier Devillers, Alexandra Fronville, Bernard Mourrain, and Monique Teillaud. Algebraic Methods and Arithmetic Filtering for Exact Predicates on Circle Arcs. *C*omputational Geometry: Theory and Applications, 22:119–142, 2002.

[DHPS07] Laurent Dupont, Michael Hemmer, Sylvain Petitjean, and Elmar Schömer. Complete, Exact and Efficient Implementation for Computing the Adjacency Graph of an Arrangement of Quadrics. In Arge et al. [AHW07], pages 633–644.

[DLLP08a] Laurent Dupont, Daniel Lazard, Sylvain Lazard, and Sylvain Petitjean. Near-optimal Parameterization of the Intersection of Quadrics: I. The Generic Algorithm. *J.* Symb. Comput., 43(3):168–191, 2008.

[DLLP08b] Laurent Dupont, Daniel Lazard, Sylvain Lazard, and Sylvain Petitjean. Near-optimal Parameterization of the Intersection of Quadrics: II. A Classification of Pencils. *J.* Symb. Comput., 43(3):192–215, 2008.

[DLLP08c] Laurent Dupont, Daniel Lazard, Sylvain Lazard, and Sylvain Petitjean. Near-optimal Parameterization of the Intersection of Quadrics: III. Parameterizing Singular Intersections. *J.* Symb. Comput., 43(3):216–232, 2008.

[DMR08] Daouda Niang Diatta, Bernard Mourrain, and Olivier Ruatta. On the Computation of the Topology of a Non-reduced Implicit Space Curve. In Juan Rafael Sendra and Laureano González-Vega, editors, *I*SSAC, pages 47–54. ACM, 2008.

[Dup22] Charles Dupin. *A*pplications de Géométrie et de Méchanique. Bachelier, Paris, 1822.

[Ede87] Herbert Edelsbrunner. *Algorithmus in Combinational Geometry*. EATCS Monographs on Theoretical Computer Science. Springer, Berlin - Heidelberg - New York, 1987.

[EH05] Eran Eyal and Dan Halperin. Dynamic Maintenance of Molecular Surfaces under Conformational Changes. In *P*roceedings of 21st Annual Symposium on Computational Geometry, pages 45–54, 2005.

[EHK+08] Ioannis Emiris, Michael Hemmer, Menelaos Karavelas, Michael Kerber, Bernard Mourrain, Elias P. Tsigaridas, and Zafeirakis Zafeirakopoulos. Cross-benchmarks of Univariate Algebraic Kernels. Technical Report of [1] with number ACS-TR-363602-02, INRIA Sophia-Antipolis, Max-Planck-Institut für Informatik, National University of Athens, 2008.

[Eig03] Arno Eigenwillig. *E*xact Arrangement Computation of Cubic Curves. Universität des Saarlandes, Saarbrücken, Germany, 2003. Diplomarbeit.

[Eig07] Arno Eigenwillig. On Multiple Roots in Descartes' Rule and Their Distance to Roots of Higher Derivatives'. *J*ournal of Computational and Applied Mathematics, 200:226–230, 2007.

[Eig08] Arno Eigenwillig. *Real Root Isolation for Exact and Approximate Polynomials Using Descartes' Rule of Signs*. PhD thesis, Universität des Saarlandes, Germany, 2008.

[EK08a] Arno Eigenwillig and Michael Kerber. Exact and Efficient 2D-Arrangements of Arbitrary Algebraic Curves. In *P*roceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA08), 2008. 122–131.

[EK08b] Pavel Emeliyanenko and Michael Kerber. An Implementation for the 2D Algebraic Kernel. Technical Report of [1] with number ACS-TR-363602-01, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2008.

[EK08c] Pavel Emeliyanenko and Michael Kerber. Visualizing and Exploring Planar Algebraic Arrangements – a Web Application. Technical Report of [1] with number ACS-TR-363608-02, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2008.

[EKK+05] Arno Eigenwillig, Lutz Kettner, Werner Krandick, Kurt Mehlhorn, Susanne Schmitt, and Nicola Wolpert. A Descartes Algorithm for Polynomials with Bit-Stream Coefficients. In *8*th International Workshop on Computer Algebra in Scientific Computing (CASC 2005), volume 3718 of *L*NCS, pages 138–149, 2005.

[EKP+04] Ioannis Z. Emiris, Athanasios Kakargias, Sylvain Pion, Monique Teillaud, and Elias P. Tsigaridas. Towards an Open Curved Kernel. In *P*roceedings of the 20th Annual Symposium on Computational Geometry (SCG 2004), pages 438–446. ACM, 2004.

[EKSW06] Arno Eigenwillig, Lutz Kettner, Elmar Schömer, and Nicola Wolpert. Exact, Efficient and Complete Arrangement Computation for Cubic Curves. *C*omputational Geometry: Theory and Applications, 35(1-2):36–73, August 2006.

[EKW07] Arno Eigenwillig, Michael Kerber, and Nicola Wolpert. Fast and Exact Geometric Analysis of Real Algebraic Plane Curves. In Christopher W. Brown, editor, *P*roocedings of the 2007 International Symposium on Symbolic and Algebraic Computation (ISSAC 2007), pages 151–158, 2007.

[Eme07] Pavel Emeliyanenko. Visualization of Points and Segments of Real Algebraic Plane Curves. Master's thesis, Universität des Saarlandes, February 2007.

[ES86] Herbert Edelsbrunner and Raimund Seidel. Voronoi Diagrams and Arrangements. *D*iscrete & Computational Geometry, 1:25–44, 1986.

[ES94] Herbert Edelsbrunner and Nimish R. Shah. Triangulating Topological Spaces. In *S*CG '94: Proceedings of the 10th Annual Symposium on Computational Geometry, pages 285–292, New York, NY, USA, 1994. ACM.

[ET03a] Ioannis Z. Emiris and Elias P. Tsigaridas. Comparison of Fourth-Degree Algebraic Numbers and Applications to Geometric Predicates. Technical Report of [5] with number ECG-TR-302206-03, National University of Athens, Athens, Greece, 2003.

[ET03b] Ioannis Z. Emiris and Elias P. Tsigaridas. Methods to Compare Real Roots of Polynomials of Small Degree. Technical Report of [5] with number ECG-TR-242200-01, National University of Athens, Athens, Greece, 2003.

[Exa04] The Exacus Team. Exacus: Strategy 2004–2008, 2004. edited by Lutz Kettner and Nicola Wolpert, for members:
`http://www.mpi-inf.mpg.de/projects/EXACUS/Members/strategy.pdf`.

[Fab07] Andreas Fabri. CGAL Editorial Board Review for Interface of the Algebraic Kernel. Technical Report of [1] with number ACS-TR-363502-01, GeometryFactory, 2007.

[FGL04] Elisabetta Fortuna, Patrizia M. Gianni, and Domenico Luminati. Algorithmical Determination of the Topology of a Real Algebraic Surface. *J*ournal of Symbolic Computation, 38:1551–1567, 2004.

[FGPT03] Elisabetta Fortuna, Patrizia M. Gianni, Paola Parenti, and Carlo Traverso. Algorithms to Compute the Topology of Orientable Real Algebraic Surfaces. *J*ournal of Symbolic Computation, 36:343–364, 2003.

[FH00] Eyal Flato and Dan Halperin. Robust and Efficient Construction of Planar Minkowski Sums. In *A*bstracts of 16th European Workshop on Computational Geometry, pages 85–88. Ben-Gurion University of the Negev, 2000.

[FH07] Efi Fogel and Dan Halperin. Exact and Efficient Construction of Minkowski Sums of Convex Polyhedra with Applications. *C*omputer-Aided Design, 39(11):929–940, 2007.

[FHH+00] Eyal Flato, Dan Halperin, Iddo Hanniel, Oren Nechushtan, and Eti Ezra. The Design and Implementation of Planar Maps in CGAL. *T*he ACM Journal of Experimental Algorithmics, 5:1–23, 2000.

[FHK+] Efi Fogel, Dan Halperin, Lutz Kettner, Monique Teillaud, Ron Wein, and Nicola Wolpert. Arrangements. In [BT06], pp. 1–66.

[FHS08] Efi Fogel, Dan Halperin, and Ophir Setter. Exact Implementation of Arrangements of Geodesic Arcs on the Sphere with Applications. In *A*bstracts of 24th European Workshop on Computational Geometry, pages 83–86. LORIA, Nancy, France, 2008.

[FKMS05] Stefan Funke, Christian Klein, Kurt Mehlhorn, and Susanne Schmitt. Controlled Perturbations for Delaunay Triangulations. In *P*roceedings of 16th ACM-SIAM Symposisum on Discrete Algorithms (SODA), pages 1047–1056, 2005.

[FM02] Stefan Funke and Kurt Mehlhorn. LOOK, a Lazy Object-Oriented Kernel for Geometric Computations. *C*omputational Geometry: Theory and Applications, 22:99–118, 2002.

[For12] Andrew Russel Forsyth. *Lectures on the Differential Geometry of Curves and Surfaces*. Cambridge University Press, 1912.

[FSH08] Efi Fogel, Ophir Setter, and Dan Halperin. Arrangements of Geodesic Arcs on the Sphere. In Monique Teillaud, editor, *P*roceedings of the 24th Annual Symposium on Computational Geometry, College Park, MD, USA, June 9-11, 2008, pages 218–219. ACM, 2008.

[FV96] Steven Fortune and Christopher J. Van Wyk. Static Analysis Yields Efficient Exact Integer Arithmetic for Computational Geometry. *A*CM Transactions on Graphics, 15(3):223–248, July 1996.

[FWH04] Efi Fogel, Ron Wein, and Dan Halperin. Code Flexibility and Program Efficiency by Genericity: Improving CGAL's Arrangements. In *P*roceedings of 12th Annual European Symposium on Algorithms, pages 664–676. Springer-Verlag, 2004.

[FWZH07] Efi Fogel, Ron Wein, Baruch Zukerman, and Dan Halperin. 2D Regularized Boolean Set-Operations. In CGAL Editorial Board, editor, *C*GAL User and Reference Manual. 3.3 edition, 2007.

[Gal01] Jean Gallier. Internet Supplement to 'Geometric Methods and Applications for Computer Science and Engineering', Chapter 23: Rational Surfaces. http://www.cis.upenn.edu/~jean/gbooks/geom2.html, 2001.

[GCL92] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *A*lgorithms for Computer Algebra. Kluwer, 1992.

[GHJV99] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *D*esign Patterns. Professional computing series. Addison-Wesley, Reading, Mass., second international student edition repr. edition, 1999.

[Gib98] Christopher G. Gibson. *E*lementary geometry of algebraic curves. Cambridge University Press, 1998.

[GLMT05] G. Gatellier, A. Labrouzy, B. Mourrain, and J.-P. Técourt. Computing the Topology of 3-dimensional Algebraic Curves. In *C*omputational Methods for Algebraic Spline Surfaces, pages 27–44. Springer, 2005.

[GVEK96] Laureano González-Vega and M'hammed El Kahoui. An Improved Upper Complexity Bound for the Topology Computation of a Real Algebraic Plane Curve. *J*ournal of Complexity, 12(4):527–544, 1996.

[GVN02] Laureano González-Vega and Ioana Necula. Efficient Topology Determination of Implicitly Defined Algebraic Plane Curves. *C*omputer Aided Geometric Design, 19:719–743, 2002.

[GVRLR] Laureano González-Vega, Tomás Recio, Henri Lombardi, and Marie-Françoise Roy. Sturm-Habicht Sequences, Determinants and Real Roots of Univariate Polynomials. In [CJ98], pp. 300–316.

[Hac07] Peter Hachenberger. Exact Minkowksi Sums of Polyhedra and Exact and Efficient Decomposition of Polyhedra in Convex Pieces. In Arge et al. [AHW07], pages 669–680.

[Hal04] Dan Halperin. Arrangements. In Jacob E. Goodman and Joseph O'Rourke, editors, *H*andbook of Discrete and Computational Geometry, chapter 24, pages 529–562. Chapman & Hall/CRC, 2nd edition, 2004.

[Hat02] Allen Hatcher. *A*lgebraic topology. Cambridge University Press, 2002.

[HE08] Iddo Hanniel and Gershon Elber. Computing the Voronoi Cells of Planes, Spheres and Cylinders in $\mathbb{R}^3$. In Haines and McGuire [HM08], pages 47–58.

[Hem02] Michael Hemmer. *Realiable Computation of Planar and Spatial Quadric Arrangements*. Universität des Saarlandes, Saarbrücken, Germany, 2002. Diplomarbeit.

[Hem07a] Michael Hemmer. Algebraic Foundations. In CGAL Editorial Board, editor, *C*GAL User and Reference Manual. 3.3 edition, 2007.

[Hem07b] Michael Hemmer. CGAL Package for Modular Arithmetic Operations. Technical Report of [1] with number ACS-TR-243406-01, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2007.

[Hem07c] Michael Hemmer. Polynomial. In CGAL Editorial Board, editor, *C*GAL User and Reference Manual. 3.4 (internal) edition, 2007.

[Hem08] Michael Hemmer. *Exact Computation of the Adjacency Graph of an Arrangements of Quadrics*. PhD thesis, Johannes-Gutenberg-Universität, Mainz, Germany, 2008.

[HH07] Michael Hemmer and Dominik Hülse. Traits Classes for Polynomial GCD Computation over Algebraic Extensions. Technical Report of [1] with number ACS-TR-241405-03, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2007.

[HK07a] Peter Hachenberger and Lutz Kettner. 2D Boolean Operations on Nef Polygons Embedded on the Sphere. In CGAL Editorial Board, editor, *C*GAL User and Reference Manual. 3.3 edition, 2007.

[HK07b] Peter Hachenberger and Lutz Kettner. 3D Boolean Operations on Nef Polyhedra. In CGAL Editorial Board, editor, *C*GAL User and Reference Manual. 3.3 edition, 2007.

[HKM07] Peter Hachenberger, Lutz Kettner, and Kurt Mehlhorn. Boolean Operations on 3D Selective Nef Complexes: Data Structure, Algorithms, Optimized Implementation and Experiments. *C*omputational Geometry: Theory and Applications, 38(1-2):64–99, 2007.

[HL93] Josef Hoschek and Dieter Lasser. *Fundations of Computer Aided Geometric Design*. A.K. Peters, 1993.

[HL04] Dan Halperin and Eran Leiserowitz. Controlled Perturbation for Arrangements of Circles. *I*nternational Journal of Computational Geometry and Applications, 14(4 & 5):277–310, 2004.

[HL07] Michael Hemmer and Sebastian Limbach. Benchmarks on a Generic Univariate Algebraic Kernel. Technical Report of [1] with number ACS-TR-243306-03, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2007.

[HL08] Michael Hemmer and Sebastian Limbach. Arrangements of Quadrics in 3D: Continued Work on Experimental Implementation. Technical Report of [1] with number ACS-TR-363606-01, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2008.

[HM08] Eric Haines and Morgan McGuire, editors. *P*roceedings of the 2008 ACM Symposium on Solid and Physical Modeling, Stony Brook, New York, USA, June 2-4, 2008. ACM, 2008.

[HS94] Dan Halperin and Micha Sharir. New Bounds for Lower Envelopes in Three Dimensions, with Applications to Visibility in Terrains. *D*iscrete Computational Geometry, 12:313–326, 1994.

[HS98] Dan Halperin and Christian R. Shelton. A Perturbation Scheme for Spherical Arrangements with Application to Molecular Modeling. *C*omputational Geometry: Theory and Applications, 10:273–287, 1998.

[HSS08] Dan Halperin, Ophir Setter, and Micha Sharir. Constructing Two-Dimensional Voronoi Diagrams via Divide-and-Conquer of Envelopes in Space. Technical Report of [1] with number ACS-TR-361601-01, Tel-Aviv University, Tel-Aviv, Israel, 2008.

[HW07] Iddo Hanniel and Ron Wein. An Exact, Complete and Efficient Computation of Arrangements of Bézier Curves. In *S*PM '07: Proceedings of the 2007 ACM Symposium on Solid and Physical Modeling, pages 253–263, New York, NY, USA, 2007. ACM.

[IEE85] *I*EEE Standard for binary floating point arithmetic, ANSI/IEEE Std 754 – 1985. New York, NY, 1985. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987.

[Joh93] John K. Johnstone. A New Intersection Algorithm for Cyclides and Swept Surfaces Using Cycle Decomposition. *C*omputer Aided Geometric Design, 10:1–24, 1993.

[Kah08] Mhammed El Kahoui. Topology of Real Algebraic Space Curves. *J*ournal of Symbolic Computation, 43(4):235–258, 2008.

[Kar06] Björn Karlsson. *Beyond the* C++ *Standard Library*. Addison-Wesley, Upper Saddle River, NJ [u.a.], 2006.

[KC08] Lutz Kettner and Fernando Cacciola. Halfedge Data Structures. In CGAL Editorial Board, editor, *C*GAL User and Reference Manual. 3.4 edition, 2008. Internal Version.

[KCF+04] John Keyser, Tim Culver, Mark Foskey, Shankar Krishnan, and Dinesh Manocha. ESOLID - a System for Exact Boundary Evaluation. *C*omputer-Aided Design, 36(2):175–193, 2004.

[KCMK00] John Keyser, Tim Culver, Dinesh Manocha, and Shankar Krishnan. Efficient and Exact Manipulation of Algebraic Points and Curves. *C*omputer-Aided Design, 32:649–662, 2000.

[Ker] Michael Kerber. PhD thesis, Universität des Saarlandes, Germany. In preparation.

[Ker06] Michael Kerber. *Analysis of Real Algebraic Plane Curves*. Universität des Saarlandes, Saarbrücken, Germany, 2006. Diplomarbeit.

[Ker08] Michael Kerber. On Filter Methods in CGAL's 2D Curved Kernel. Technical Report of [1] with number ACS-TR-243404-03, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2008.

[Ket06] Lutz Kettner. Reference Counting in Library Design—Optionally and with Union-Find Optimization. In David Musser and Jeremy Siek, editors, *P*roceedings of the First International Workshop on Library-Centric Software Design, LCSD'05, volume 06-12 of *T*echnical Report, pages 34–43, San Diego, CA, USA, 2006. Rensselaer Polytechnic Institute, Computer Science Department.

[Ket07] Lutz Kettner. Halfedge Data Structures. In CGAL Editorial Board, editor, *C*GAL User and Reference Manual. 3.3 edition, 2007.

[KLPY99] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core Library for Robust Numeric and Geometric Computation. In *P*roceedings of the 15th Annual ACM Symposium of Computational Geometry (SCG), pages 351–359, 1999.

[KMP+04] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee-Keng Yap. Classroom Examples of Robustness Problems in Geometric Computations. In Susanne Albers and Tomasz Radzik, editors, *E*SA, volume 3221 of *L*ecture Notes in Computer Science, pages 702–713. Springer, 2004.

[KN04] Lutz Kettner and Stefan Näher. Two Computational Geometry Libraries: LEDA and CGAL. In Jacob E. Goodman and Joseph O'Rourke, editors, *H*andbook of Discrete and Computational Geometry, chapter 65, pages 1435–1463. CRC Press LLC, Boca Raton, FL, second edition, 2004.

[Kön93] Konrad Königsberger. *A*nalysis 2. Springer, Berlin - New York, 1993.

[KOS92] Matthew J. Katz, Mark H. Overmars, and Micha Sharir. Efficient Hidden Surface Removal for Objects With Small Union Size. *C*omputational Geometry: Theory and Applications, 2:223–234, 1992.

[KP02] Steven George Krantz and Harold R. Parks. *The Implicit Function Theorem: History, Theory, and Application.* Birkhäuser, 2002.

[KY07] Menelaos Karavelas and Mariette Yvinec. 2D Apollonius Graphs (Delaunay Graphs of Disks). In CGAL Editorial Board, editor, *C*GAL User and Reference Manual. 3.3 edition, 2007.

[Lan02] Serge Lang. *A*lgebra. Addison-Wesley, rev. 3. edition, 2002.

[Lat93] Jean-Claude Latombe. *Robot Motion Planning*, volume 0124; of *K*luwer international series in engineering and computer science; SECS. Kluwer, 1993.

[Lev79] Joshua Zev Levin. Mathematical Models for Determining the Intersections of Quadric Surfaces. *C*ompututer Graphics and Image Processing, 11:73–87, 1979.

[Loo82a] Rüdiger Loos. Computing in Algebraic Extensions. In Bruno Buchberger, George E. Collins, and Rüdiger Loos, editors, *C*omputer Algebra – Symbolic and Algebraic Computation, pages 173–188. Springer, 1982.

[Loo82b] Rüdiger Loos. Generalized Polynomial Remainder Sequences. In Bruno Buchberger, George E. Collins, and Rüdiger Loos, editors, *C*omputer Algebra – Symbolic and Algebraic Computation, pages 115–138. Springer, 1982.

[LPP06] Sylvain Lazard, Luis Mariano Peñaranda, and Sylvain Petitjean. Intersecting Quadrics: An Efficient and Exact Implementation. *C*omputational Geometry, 35(1-2):74–99, 2006.

[LPT08] Sylvain Lazard, Luis Peñaranda, and Elias Tsigaridas. A CGAL-Based Univariate Algebraic Kernel and Applications to Arrangements. In *A*bstracts of 24th European Workshop on Computational Geometry, pages 91–94. LORIA, Nancy, France, 2008.

[LY01] Chen Li and Chee-Keng Yap. A New Constructive Root Bound for Algebraic Expressions. In *S*ODA, pages 496–505, 2001.

[Mas67] William S. Massey. *A*lgebraic Topology: An Introduction. Springer, 1967.

[MC02] Scott McCallum and George E. Collins. Local Box Adjacency Algorithms for Cylindrical Algebraic Decompositions. *J*ournal of Symbolic Computation, 33:321–342, 2002.

[McC] Scott McCallum. An Improved Projection Operation for Cylindrical Algebraic Decomposition. In [CJ98], pp. 242–268.

[Meh01] Kurt Mehlhorn. Circle Points and Predicates on Circle Points. Lectures on November 15 and 20., 2001. Lecture Notes for "Effective Computational Geometry: Theory and Practice of Implementing Geometric Algorithms" (WS 2001/02, Saarland University) http://www.mpi-inf.mpg.de/~mehlhorn/ECG/CirclePointsandPredicates.ps.

[Mey06a] Michal Meyerovitch. Robust, Generic and Efficient Construction of Envelopes of Surfaces in Three-Dimensional Space. M.Sc. thesis, School of Computer Science, Tel-Aviv University, Tel-Aviv, Israel, July 2006.

[Mey06b] Michal Meyerovitch. Robust, Generic and Efficient Construction of Envelopes of Surfaces in Three-Dimensional Spaces. In Yossi Azar and Thomas Erlebach, editors, *E*SA, volume 4168 of *L*ecture Notes in Computer Science, pages 792–803. Springer, 2006.

[MN00] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[MOS06] Kurt Mehlhorn, Ralf Osbild, and Michael Sagraloff. Reliable and Efficient Computational Geometry via Controlled Perturbation. In Michele Bugliesi, Bart Preneel, Vladimir Sassone, and Ingo Wegener, editors, *A*utomata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Part I, volume 4051 of *L*ecture Notes in Computer Science, pages 299–310, Venice, Italy, 2006. Springer.

[MPS+] Bernard Mourrain, Sylvain Pion, Susanne Schmitt, Jean-Pierre Técourt, Elias Tsigaridas, and Nicola Wolpert. Algebraic Issues in Computational Geometry. In [BT06], pp. 117-155.

[MRR05] Bernard Mourrain, Fabrice Rouillier, and Marie-Françoise Roy. Bernstein's Basis and Real Root Isolation. In J. E. Goodman, J. Pach, and E. Welzl, editors, *C*ombinatorial and Computational Geometry, Mathematical Sciences Research Institute Publications, pages 459–478. Cambridge University Press, 2005.

[MS88] David A. Musser and Alexander A. Steanov. Generic Programming. In *P*roceedings of International Symposium on Symbolic and Algebraic Computation, LNCS 358, pages 13–25. Springer-Verlag, 1988.

[MS03] Kurt Mehlhorn and Michael Seel. Infimaximal Frames: A Technique for Making Lines Look Like Segments. *J*ournal of Computational Geometry & Applications, 13(3):241–255, 2003.

[MT05] Bernard Mourrain and Jean-Pierre Técourt. Isotopic Meshing of a Real Algebraic Surface. Technical Report 5508, INRIA Sophia-Antipolis, 2005.

[MTT05] Bernard Mourrain, Jean-Pierre Técourt, and Monique Teillaud. On the Computation of an Arrangement of Quadrics in 3D. *C*omputational Geometry, 30(2):145–164, 2005.

[Mul89] Ketan Mulmuley. An Efficient Algorithm for Hidden Surface Removal. In *P*roceedings SIGGRAPH 1989, pages 379–388, New York, NY, USA, 1989. ACM Press.

[MWZ07] Michal Meyerovitch, Ron Wein, and Baruch Zukerman. 3D Envelopes. In CGAL Editorial Board, editor, *C*GAL User and Reference Manual. 3.3 edition, 2007.

[Mye95] Nathan C. Myers. Traits: A New and Useful Template Technique. *C++* Report, 7(5):32–35, 1995.

[Nef78] Walter Nef. *Beiträge zur Theorie der Polyeder*. Herbert Lang, Bern, 1978.

[Pra90] Michael J. Pratt. Cyclides in Computer Aided Geometric Design. *C*omputer Aided Geometric Design, 7:221–242, 1990.

[Pra95] Michael J. Pratt. Cyclides in Computer Aided Geometric Design II. *C*omputer Aided Geometric Design, 12:131–152, 1995.

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

[PT07] Sylvain Pion and Monique Teillaud. 2D Circular Geometry Kernel. In CGAL Editorial Board, editor, *C*GAL User and Reference Manual. 3.3 edition, 2007.

[PTT06] Sylvain Pion, Monique Teillaud, and Constantinos P. Tsirogiannis. Geometric Filtering of Primitives on Circular Arcs. Technical Report of [1] with number ACS-TR-121105-01, INRIA Sophia-Antipolis, France, 2006.

[PV07] Simon Plantinga and Gert Vegter. Isotopic Meshing of Implicit Surfaces. *T*he Visual Computer, 23:45–58, 2007.

[Raa99] Sigal Raab. Controlled Perturbation for Arrangements of Polyhedral Surfaces with Application to Swept Volumes. In *P*roceedings of 15th Annual Symposium on Computational Geometry, pages 163–172, 1999.

[Rei08] Tobias Reithmann. Topological Correct Intersection Curves of Tori and Natural Quadrics. Technical Report of [1] with number ACS-TR-361502-01, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2008.

[RZ03] Fabrice Rouillier and Paul Zimmermann. Efficient Isolation of a Polynomial's Real Roots. *J*ournal of Computational and Applied Mathematics, 162(1):33–50, 2003.

[SA95] Micha Sharir and Pankaj K. Agarwal. *Davenport-Schinzel sequences and heir geometric applications*. Cambridge University Press, 1st ed. edition, 1995.

[Sch96] Stefan Schirra. Precision and Robustness in Geometric Computations. In Marc J. van Kreveld, Jürg Nievergelt, Thomas Roos, and Peter Widmayer, editors, *A*lgorithmic Foundations of Geographic Information Systems, volume 1340 of *L*ecture Notes in Computer Science, pages 255–287. Springer, 1996.

[Sch08] Stefan Schirra. How Reliable Are Practical Point-in-Polygon Strategies? In Dan Halperin and Kurt Mehlhorn, editors, *A*lgorithms - ESA 2008, 16th Annual European Symposium on Algorithms, Karlsruhe, Deutschland, September 15-17, 2008, Proceedings, volume 5193 of *L*NCS, pages 744–755. Springer, 2008.

[SH89] Jack Snoeyink and John Hershberger. Sweeping Arrangements of Curves. In *S*ymposium on Computational Geometry, pages 354–363, 1989.

[Sha94] Micha Sharir. Almost Tight Upper Bounds for Lower Envelopes in Higher Dimensions. *D*iscrete Computational Geometry, 12:327–345, 1994.

[She96] Jonathan Richard Shewchuk. Adaptive Precision Floating Point Arithmetic and Fast Robust Reometric Predicates. Technical Report CMU-CS-96-140, Carnegie Mellon University, Pittsburgh, PA, 1996.

[SS83] Jacob T. Schwartz and Micha Sharir. On the Piano Mover's Problem II: General Techniques for Computing Topological Properties of Algebraic Manifolds. *A*dvances in Applied Mathematics, 4:298–351, 1983.

[SSH87] Jacob T. Schwartz, Micha Sharir, and John E. Hopcroft, editors. *Planning, Geometry, and Complexity of Robot Motion*. Ablex series in artificial intelligence. Ablex Publ., 1987.

[Str06] Adam W. Strzebonski. Cylindrical Algebraic Decomposition using Validated Numerics. *J*ournal of Symbolic Computation, 41:1021–1038, 2006.

[SW05] Raimund Seidel and Nicola Wolpert. On the Exact Computation of the Topology of Real Algebraic Curves. In *P*roceedings of the 21st Annual Symposium on Computational Geometry (SCG 2005), pages 107–115, 2005.

[TE08] Elias P. Tsigaridas and Ioannis Z. Emiris. On the Complexity of Real Root Isolation Using Continued Fractions. *T*heoretical Computer Science, 392(1-3):158–173, 2008.

[vdW71] Bartel L. van der Waerden. *A*lgebra I [früher u.d.T.: Moderne Algebra], volume 12 of *H*eidelberger Taschenbücher. Springer, 8. auflage edition, 1971.

[vzGG99] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[Wal50] Robert J. Walker. *A*lgebraic Curves. Princeton University Press, 1950.

[Wei02] Ron Wein. High-Level Filtering for Arrangements of Conic Arcs. In *P*roceedings of the 10th European Symposium on Algorithms (ESA 02), pages 884–895, 2002.

[Wei07a] Ron Wein. 2D Envelopes. In CGAL Editorial Board, editor, *C*GAL User and Reference Manual. 3.3 edition, 2007.

[Wei07b] Ron Wein. 2D Minkowski Sums. In CGAL Editorial Board, editor, *C*GAL User and Reference Manual. 3.3 edition, 2007.

[WFZH05] Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin. Advanced Programming Techniques Applied to CGAL's Arrangements. In *P*roceedings of the First International Workshop on Library-Centric Software Design (LCSD '05), at the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) Conference., October 2005.

[WFZH07a] Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin. 2D Arrangements. In CGAL Editorial Board, editor, *C*GAL User and Reference Manual. 3.3 edition, 2007.

[WFZH07b] Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin. Advanced Programming Techniques Applied to CGAL's Arrangement Package. *C*omputational Geometry: Theory and Applications, 38(1-2):37–63, 2007.

[Whi49] J. H. C. Whitehead. Combinatorial Homotopy. I. *B*ull. Amer. Math. Soc., 55(3):213–245, 1949.

[Wol02] Nicola Wolpert. *A*n Exact and Efficient Approach for Computing a Cell in an Arrangement of Quadrics. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2002.

[Wol03] Nicola Wolpert. Jacobi Curves: Computing the Exact Topology of Arrangements of Non-Singular Algebraic Curves. In *P*roceedings of the 11th Annual European Symposium on Algorithms (ESA 2003), volume 2832 of *L*NCS, pages 532–543. Springer, 2003.

[WZ06] Ron Wein and Baruch Zukerman. Exact and Efficient Construction of Planar Arrangements of Circular Arcs and Line Segments with Applications. Technical Report of [1] with number ACS-TR-121200-01, Tel-Aviv University, Tel-Aviv, Israel, 2006.

[Yap00] Chee Keng Yap. *Fundamental Problems in Algorithmic Algebra*. Oxford University Press, 2000.

[Yap04] Chee Yap. Robust Geometric Computation. In Jacob E. Goodman and Joseph O'Rourke, editors, *H*andbook of Discrete and Computational Geometry, chapter 41, pages 927–952. Chapman & Hall/CRC, 2nd edition, 2004.

[Yun76] David Y.Y. Yun. On Square-free Decomposition Algorithms. In *S*YMSAC '76: Proceedings of the Third ACM Symposium on Symbolic and Algebraic Computation, pages 26–35, New York, NY, USA, 1976. ACM.

All cyclide pictures are produced with `xsurface` that is based on CGAL's planar curve renderer [Eme07]. The author thanks Pavel Emeliyanenko for his contribution.

Pictures: Most figures in this thesis are created by the author using `xfig` [18], or screenshots of EXACUS-demos `xquadri` and `xsurface`. We give details on the copyright next to each exception: Figure 1.3 (c) on page 16, Figure 3.14 on page 112, Figure 4.4 on page 122, Figure 4.23 on page 175, Figure 4.24 on page 179, and Figure 5.8 on page 231.

Links

[1] ACS, Algorithms for Complex Shapes with certified topology and numerics.
http://acs.cs.rug.nl/.

[2] BOOST, C++ Libraries.
http://www.boost.org/.

[3] CGAL, Computational Geometry Algorithms Library.
http://www.cgal.org/.

[4] CORE Number Library.
http://cs.nyu.edu/exact/core_pages/.

[5] ECG, Effective Computational Geometry for Curves and Surfaces.
http://www-sop.inria.fr/prisme/ECG/.

[6] EXACUS, Efficient and Exact Algorithms for Curves and Surfaces.
http://www.mpi-inf.mpg.de/projects/EXACUS.

[7] EXACUS Webdemo, Computing and Visualizing Arrangements of Algebraic Curves.
http://exacus.mpi-inf.mpg.de.

[8] GEOMETRYFACTORY.
http://www.geometryfactory.com.

[9] GMP, GNU Multiple Precision Arithmetic Library.
http://www.swox.com/gmp.

[10] LEDA, Library for Efficient Data Types and Algorithms.
http://www.algorithmic-solutions.com/leda/index.htm.

[11] MPFI, Multiple Precision Interval Arithmetic Library.
http://perso.ens-lyon.fr/nathalie.revol/software.html.

[12] MPFR, Multiple-Precision Floating-Point Computations.
http://www.mpfr.org.

[13] NTL, A Library for doing Number Theory.
http://www.shoup.net/ntl/.

[14] RS: Real roots of systems with a finite number of complex solutions.
http://fgbrs.lip6.fr.

[15] STL, C++ Standard Template Library.
http://www.sgi.com/tech/stl.

[16] SYNAPS, A Library for Symbolic and Numeric Computation.
http://www-sop.inria.fr/galaad/logiciels/synaps.

[17] Wolfram Mathworld. Cyclide.
http://mathworld.wolfram.com/Cyclide.html.

[18] Xfig Drawing Program for the X Windows System.
http://www.xfig.org.

Index

A List of Algebraic Surfaces

This appendix gives the defining polynomials of the example surfaces analyzed in §5.6.2, which allows to rerun experiments or to play around with the surfaces.

steiner-roman
$f = (y^2 + (x^2)) \cdot z^2 + (((1) \cdot x) \cdot y) \cdot z + ((x^2) \cdot y^2)$

cayley-cubic
$f = (5 \cdot y + (5 \cdot x)) \cdot z^2 + (5 \cdot y^2 + (-2) \cdot y + (5 \cdot x^2 + (-2) \cdot x)) \cdot z + ((5 \cdot x) \cdot y^2 + (5 \cdot x^2 + (-2) \cdot x) \cdot y)$

dupin-cyclic
$f = 447279 \cdot z^4 + (894558 \cdot y^2 + (894558 \cdot x^2 + (-1155200) \cdot x + 1155200)) \cdot z^2 + (447279 \cdot y^4 + (894558 \cdot x^2 + (-1155200) \cdot x + (-1155200)) \cdot y^2 + (447279 \cdot x^4 + (-1155200) \cdot x^3 + (-1404800) \cdot x^2 + 5120000 \cdot x + (-2560000)))$

tangle-cube
$f = z^4 + (-5) \cdot z^2 + (y^4 + (-5) \cdot y^2 + (x^4 + (-5) \cdot x^2 + 10))$

bohemian-dome
$f = z^4 + (2 \cdot y^2 + ((-2) \cdot x^2)) \cdot z^2 + ((-1) \cdot y^4 + (2 \cdot x^2 + (-4)) \cdot y^2 + (x^4))$

chair $f = 16 \cdot z^4 + (288 \cdot y^2 + (288 \cdot x^2 + (-600))) \cdot z^2 + ((-1280) \cdot y^2 + (1280 \cdot x^2)) \cdot z + (80 \cdot y^4 + ((-96) \cdot x^2 + (-600)) \cdot y^2 + (80 \cdot x^4 + (-600) \cdot x^2 + 5125))$

hunt
$f = 4 \cdot z^6 + (12 \cdot y^2 + (12 \cdot x^2 + 276)) \cdot z^4 + (12 \cdot y^4 + (24 \cdot x^2 + (-528)) \cdot y^2 + (12 \cdot x^4 + (-960) \cdot x^2 + 4620)) \cdot z^2 + (4 \cdot y^6 + (12 \cdot x^2 + (-129)) \cdot y^4 + (12 \cdot x^4 + (-150) \cdot x^2 + 1380) \cdot y^2 + (4 \cdot x^6 + 87 \cdot x^4 + 84 \cdot x^2 + (-4900)))$

star
$f = 100 \cdot z^6 + (300 \cdot y^2 + (300 \cdot x^2 + (-300))) \cdot z^4 + (300 \cdot y^4 + (600 \cdot x^2 + (-599)) \cdot y^2 + (300 \cdot x^4 + (-599) \cdot x^2 + 300)) \cdot z^2 + (100 \cdot y^6 + (300 \cdot x^2 + (-300)) \cdot y^4 + (300 \cdot x^4 + (-599) \cdot x^2 + 300) \cdot y^2 + (100 \cdot x^6 + (-300) \cdot x^4 + 300 \cdot x^2 + (-100)))$

spiky
$f = z^6 + ((-3) \cdot y^3 + (3 \cdot x^2)) \cdot z^4 + (3 \cdot y^6 + (21 \cdot x^2) \cdot y^3 + (3 \cdot x^4)) \cdot z^2 + ((-1) \cdot y^9 + (3 \cdot x^2) \cdot y^6 + ((-3) \cdot x^4) \cdot y^3 + (x^6))$

C8
$f = 32 \cdot z^8 + (-64) \cdot z^6 + 40 \cdot z^4 + (-8) \cdot z^2 + (32 \cdot y^8 + (-64) \cdot y^6 + 40 \cdot y^4 + (-8) \cdot y^2 + (32 \cdot x^8 + (-64) \cdot x^6 + 40 \cdot x^4 + (-8) \cdot x^2 + 1))$

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Saarbrücken,

Eric Berberich